Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik VI

Algorithmen und Datenstrukturen

Vorlesungsmitschrift zur Vorlesung im SS'98

Prof. Dr.-Ing. H. Ney

Ausgearbeitet von Christian Fuß
in Zusammenarbeit mit
Sonja Nießen und Jörg Dahmen

Letzte Überarbeitung: 05.02.1999

Inhaltsverzeichnis

Literatur

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

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

[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 Bd 1: 'Sortieren und Suchen'), Teubner 1988. (Bde 2+3 vergriffen, Neuauflage ?)

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

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

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

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

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

Hinweise:

	[Sedgewick 88, Sedgewick 93]	: konkrete Programme
	[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:	a) Schülerduden INFORMATIK
		b) INFORMATIK (fast identisch)

1 Grundlagen

1.1 Einführung

1.1.1 Konkrete 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. In diesem Zusammenhang werden folgende Problemstellungen der Informatik betrachtet:

- *Operationen auf Mengen*, z.B. die gleichzeitige Bestimmung des Maximums und Minimums einer Menge.
- *Sortieren* von Folgen reeller Zahlen und Bestimmung ihres Medians.
- *Suchen* in Datensätzen (Datenbanken)
  Beispiele (aus [Mehlhorn, Seite 97]):
  - Symboltabellen für Compiler (enthalten vordefinierte Wörter, Variablen, etc.),
  - Autorenverzeichnis einer Bibliothek,
  - Kontenverzeichnis einer Bank,
  - allgemeine Datenbanken.
- *Pfade in einem Graphen*:
  - kürzester Pfad von A nach B,
  - kürzester Pfad, der alle Knoten miteinander verbindet
    (Minimalbaum, Minimalgerüst, Minimum Spanning Tree (MST)),
  - kürzeste Rundreise, bei der alle Knoten genau einmal besucht werden
    (Traveling Salesman Problem, TSP).
- Suche nach Character-Folgen („Mustern“) im Text.

1.1.2 Aktualität des Themas

Das Thema der guten Algorithmen und ihrer effizienten Implementierung ist stets aktuell gewesen. Auch in Zeiten der immer schneller wachsenden Ressourcen (Speicher- und Rechenkapazität) ist es ökonomisch unerläßlich, sie möglichst effizient zu nutzen, da man sonst schnell auch derzeitig verfügbare Ressourcen erschöpft, wie wir später sehen werden.

Auch in der Forschung gibt es viele aktuelle Aspekte. So existiert bisher keine „gute“ Lösung für das Traveling Salesman Problem. Neben diesen theoretischen Fragestellungen existieren viele Fragestellungen in der angewandten Informatik, die nach extrem effizienten Algorithmen verlangen, so z.B. bei der *Erkennung gesprochener Sprache*, unserem eigenen Forschungsgebiet am Lehrstuhl für Informatik VI. Denn dabei müssen große Mengen von Hypothesen bearbeitet werden, diese werden u.a. mit der Methode des besten Pfades durchsucht.

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 (CPU-Zeit, Speicherplatz) eines Verfahrens definiert? Wie kann man diesen Aufwand in Komplexitätsmaße fassen, und wie analysiere ich einen Algorithmus überhaupt?
- Praktische Fähigkeiten: Wie analysiere ich ein Problem/eine Aufgabenstellung, und wie bilde ich dieses dann auf ein Verfahren = Datenstruktur + Algorithmus ab? Wie implementiere ich ein lauffähiges Programm aus dem erarbeiteten Verfahren, und wie teste ich es dann?

1.1.4 Hinweis auf das didaktische Problem der Vorlesung

Es ist wesentlich schwieriger klarzumachen, 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 fertigen 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 die Details wie z.B. Index-Grenzen auch 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,
- Binomialkoeffizienten,
- Rekursionsgleichungen und
- Wahrscheinlichkeitsrechnung.

1.1.5 Vorgehensweise

Bei der Behandlung der Algorithmen und Datenstrukturen werden zwei Ebenen unterschieden [Güting, Seite 3]:

- *Die algorithmische Ebene*: Sie umfaßt eine möglichst allgemeine und maschinenunabhängige Beschreibung der Objekte (zu manipulierende Daten) und des Algorithmus. Dies hat den Vorteil, daß man sich auf das Wesentliche konzentriert, und die Lösung portabel ist.
- *Die programmiersprachliche Ebene*: Sie umfaßt die konkrete Implementierung, wobei „konkret" nicht ganz streng aufzufassen ist, denn der Übersichtlichkeit wegen werden in der Vorlesung nicht immer vollständig lauffähige Implementationen vorgestellt. Zum Verständnis der Programmcodes ist die grundlegende Kenntnis einer imperativen Programmiersprache Voraussetzung. Programme und Algorithmen sind in folgenden imperativen Sprachen formuliert:
  - Pascal/Modula,
  - Pascal/Modula-naher Pseudocode und
  - allgemeiner Pseudo-Code.

1.1.6 Datenstrukturen und Algorithmen

Datenstrukturen und Algorithmen sind unmittelbar miteinander verknüpft und kennzeichnen ein Verfahren. Meist ergeben sich die Datenstrukturen direkt aus der Problemstellung und können grob unterteilt werden in die folgenden Kategorien:

- Sequenzen (Folgen, Listen),
- Mengen (speziell Dictionaries (Lexika)) und
- Graphen (speziell Bäume).

1.2 Algorithmen und Komplexität

Wenn wir uns für die Komplexitätsordnung eines Algorithmus interessieren, so liegt das vordergründig daran, daß sie maßgeblich die Laufzeit und den Platzbedarf der konkreten Implementierung im Programm bestimmt. Diese hängt aber, neben dem Algorithmus selbst, u.a. noch von den folgenden Größen ab:

- den Eingabedaten,
- der Qualität des Compilers,
- der Rechner-Hardware und
- dem Betriebssystem.

Die letzten drei Punkte sind mehr oder minder maschinenabhängig und sollen uns deswegen nicht interessieren, so daß wir nur die abstrakte Laufzeit in Abhängigkeit von den Eingabedaten betrachten. Für diese schreibt man $T(n)$. Dabei drückt der Parameter n die Größenordnung der Eingabedaten aus, die je nach Problemstellung auch verschieden aufgefaßt werden kann, z.B.:

- Beim *Sortieren* drückt n die Anzahl der zu sortierenden Werte $a_1, \ldots, a_n$ aus,
- während beim *Finden aller Primzahlen* $\leq n$ diese Schranke durch n ausgedrückt wird.

Zur Veranschaulichung betrachten wir das Beispiel eines der einfachsten Sortierverfahren, *Sortieren durch Auswählen*. Es besteht aus zwei verschachtelten Schleifen, und kann wie folgt implementiert werden:

```
procedure SelectionSort(var A: array of ItemType) =
  var min, i, j: cardinal;
      t: ItemType;
  begin
    for i:= FIRST(A) to (LAST(A)-1) do
      min:=i;
      for j:=i+1 to LAST(A) do
        if A[j]<A[min] then min:=j;
      end;
      t:=A[min]; A[min]:=A[i]; A[i]:=t;
    end;
  end SelectionSort;
```

Die Rechenzeit $T(n)$ wird hier bestimmt durch die elementaren Operationen *vergleichen* (**c**ompare) und *vertauschen* (**e**xchange). Deren Anzahl wollen wir jetzt in Abhängigkeit von n, der Anzahl der zu sortierenden Elemente ermitteln:

- Anzahl der Vergleiche:

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

- Anzahl der Vertauschungen:

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

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

- Bezeichnet man mit α_1 und α_2 den relativen Aufwand für Vergleiche und Vertauschungen, dann ergibt sich folgender Gesamtaufwand:

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

Diese Aussage gilt für große n und wir können sagen, daß das *asymptotische Verhalten* von $T(n)$ dem von n^2 entspricht, oder man sagt auch: $T(n)$ hat quadratische Komplexität.

1.2.1 Komplexitätsklassen

Um Algorithmen bezüglich ihrer Komplexität besser vergleichen zu können teilt man diese in Komplexitätsklassen ein. In der nachfolgenden Tabelle sind typische Zeitkomplexitätsklassen aufgeführt. Sortiert sind sie nach ihrem Komplexitätsgrad:

Formel	**Name**	**Beispiel**
1	konstant	elementarer Befehl
$\log(\log n)$	doppelt logarithmisch	
$\log n$	logarithmisch	Binärsuche
n	linear	lineare Suche
$n \log n$	überlinear	Divide-and-Conquer-Strategien
n^2	quadratisch	einfache Sortierverfahren
n^3	kubisch	Matrizen-Inversion
n^k	polynomiell vom Grad k	lineare Programmierung
2^n	exponentiell	Exhaustive Search
$n!$	Fakultät	Traveling Salesman Problem
n^n		

Tabelle: Typische Zeitkomplexitätsklassen.

Bei den logarithmischen Komplexitätsklassen spielt die Basis in der Regel keine Rolle, denn ein Basiswechsel bedeutet nur die Multiplikation mit einem konstanten Faktor. Wenn keine Basis explizit angegeben wird, dann geht diese für $\log(n)$ meist aus dem Kontext hervor. Sonst steht ld für Logarithmus dualis ($\log_2$), und lg für den dekadischen Logarithmus ($\log_{10}$).

Zur Veranschaulichung der Komplexitätsklassen hier eine vergleichende Tabelle. Sie geht von hypothetischen Algorithmen mit den angegebenden Komplexitäten aus. Es wird angenommen, daß der Rechner 0,1 μsec (1 μsec = 10^{-6} Sekunden) für eine elementare Operation benötigt.

T(n)	**n = 10**	**n = 20**	**n = 50**	**n = 100**	**n = 300**
$\log n$	0,23 μsec	0,30 μsec	0,39 μsec	0,46 μsec	0,57 μsec
n	1 μsec	2 μsec	5 μsec	10 μsec	30 μsec
n^2	10 μsec	40 μsec	250 μsec	1 msec	9 msec
n^5	10 msec	320 msec	31,25 sec	2,78 h	28 Tage
2^n	102 μsec	104 msec	35,7 Jahre	$4 \cdot 10^{16}$ Jahre	...
$n!$	3,6 sec	77147 Jahre	$9,6 \cdot 10^{50}$ Jahre	...	...

Tabelle: Verhalten einiger Zeitkomplexitätsklassen.

Zum Vergleich: Der Urknall war vor etwa 15 Milliarden Jahren.

Aus den Vergleichen der Laufzeit der hypothetischen Algorithmen folgt für die *Komplexität* von Programmen, daß in der Praxis in der überwiegenden Mehrzahl der Fälle nur solche Algorithmen eingesetzt werden können, die maximal polynomielle oder geringere Komplexität haben, da die exponentiellen Algorithmen zu „beliebig“ langen Laufzeiten führen.

1.2.2 O-Notation (Ω, Θ) für asymptotische Aussagen

Beschreibt eine Funktion die Laufzeit eines Algorithmus, so ist es oft ausreichend nur ihr asymptotisches Verhalten zu untersuchen, um Aussagen über die Komplexität eines Programms zu machen. Das bietet den Vorteil, daß man dann konstante Faktoren vernachlässigen, und sich auf die *Betrachtung einer einfacheren Funktion* beschränken kann. Neben diesem Ziel sollte aber auf *möglichst enge Schranken* geachtet werden.
Um asymptotische Aussagen mathematisch konkreter zu fassen, definiert man die *O-Notation* wie folgt:

Definition *O-Notation*: Sei $f : \mathbb{N} \to \mathbb{R}^+$ eine Funktion

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

Bedeutung: Seien f und g zwei Funktionen von n, dann verwendet man folgende Sprechweisen:
$g \in O(f)$: 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
Bei Laufzeitabschätzungen wird meist $O(f)$ verwendet. Statt $g \in O(f)$ schreibt man oft auch $g = O(f)$.

Allgemeine Aussagen und Rechenregeln zur O-Notation

Für das Rechnen mit obigen Definitionen gelten einige einfache Regeln. Seien f und g Funktionen von n, dann gilt:

1. Falls $\exists \lim\limits_{n \to \infty} \dfrac{g(n)}{f(n)} \Longrightarrow g \in O(f)$ [Güting]

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

Beweis: laut Definition muß es also $c > 0$, $n_0 > 0$ geben, so daß

$$g(n) = \alpha \cdot f(n) + \beta \leq c \cdot f(n) \;\; \forall\, n \geq n_0$$

Wegen $f \in \Omega(1)$ $\exists\, c' > 0,\, n' > 0$, mit $f(n) \geq c' \cdot 1 \;\; \forall\, n \geq n'$.

Man wählt $c = \alpha + \dfrac{\beta}{c'}$ und $n_0 = n'$.

Mit diesen Werten für c und n_0 ist obige Ungleichung erfüllt, wie man leicht nachvollziehen kann.

3. „**Addition**“: Setzt sich ein Programm aus mehreren Teilen zusammen, die unterschiedliche Laufzeiten haben, so wird das asymptotische Verhalten allein von dem Teil bestimmt, der den größten Aufwand hat.

$$f(n) \;+\; g(n) \;\in\; O(\max\{f(n),\, g(n)\}) \;=\; \begin{cases} O(g) & \text{falls } \; f \;\in\; O(g) \\ O(f) & \text{falls } \; g \;\in\; O(f) \end{cases}$$

4. „**Multiplikation**“: Laufzeiten müssen „multipliziert“ werden, falls z.B. eine Schleife, deren Kern eine Laufzeit in $O(f)$ hat, von einer Prozedur k-mal aufgerufen wird, mit $k \in O(g)$.

   Für die Gesamtlaufzeit $T(n)$ gilt dann:

$$T(n) \in \; O(f \cdot g)$$

   Anmerkung: Falls die Anzahl der Aufrufe k nicht von n abhängt, gilt natürlich $k \in O(1)$ und somit $T(n) \in O(f \cdot 1) = O(f)$.

Beispiele:

1. 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 $g \in O(f)$.

2. Sei $f(n) = 7n + 3$. Dann gilt:

$$7n + 3 \leq c \cdot n \quad \forall\, n \geq n_0 \text{ mit } c = 8 \text{ und } n_0 = 3$$

   und damit $f(n) \in O(n)$

3. 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

$$3^n \;\leq\; c \;\cdot\; 2^n \;\;\; \forall\, n \geq n_0$$

4. 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:

$$T_1(n) \;+\; T_2(n) \;+\; T_3(n) \;\in\; O(n^3)$$

1.2.3 Regeln für die Laufzeitanalyse

1. *Elementare Anweisungen* sind in $O(1)$.
   Anmerkung: Falls ein elementarer Befehl k-mal aufgerufen wird und die Anzahl der Aufrufe k konstant ist, gilt natürlich $k \in O(1)$ und somit auch die k-malige Hintereinanderausführung der elementaren Anweisung. Anders ist dies, falls die Anzahl der Aufrufe k von n abhängt (vgl. *Multiplikation*).

2. *Folgen von Anweisungen*: Betrachte nur die Anweisung mit der größten Laufzeit (vgl. *Addition*).

3. *Schleifendurchläufe*: vgl. *Multiplikation*

4. *Bedingte Anweisungen*: S_1 und S_2 seien zwei Subprozeduren mit Laufzeiten $T_{S_1} \in O(f_1)$ und $T_{S_2} \in O(f_2)$, dann hat die Anweisung

   if (Bedingung) **then** S_1 **else** S_2

   eine Laufzeit in $O(1) + O(f_1 + f_2)$, falls die Bedingung in konstanter Zeit ausgewertet wird. Vergleiche Addition für die Berechnung von $O(f_1 + f_2)$.

5. *Prozeduraufrufe* müssen unterschieden werden nach nicht-rekursiv und rekursiv.

   - nicht-rekursiv: Jede Prozedur kann separat analysiert werden.
   - rekursiv: Eine direkte Analyse ist nicht möglich, man muß auf Rekursionsgleichungen zurückgreifen (siehe Abschnitt 1.4.2, Seite 39).

1.2.4 Fibonacci-Zahlen

Am Beispiel verschiedener Programme zur Berechnung der Fibonacci-Zahlen wollen wir Laufzeitanalysen durchführen.

Definition *Fibonacci-Zahlen*:

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

Damit ergibt sich die Folge: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...
Wir wollen zunächst die Fibonacci-Zahlen selbst, bzw. ihr Wachstum betrachten.

Behauptung:

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

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

Beweis:

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

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

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

$$\begin{array}{rcl} f(n) & = & f(n-1) + f(n-2) \\ & < & 2 \cdot f(n-1) \\ & < & 4 \cdot f(n-2) \\ & \vdots & \\ & < & 2^{n-1} \cdot f(1) \end{array}$$

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

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

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

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

Versuchen Sie, ebenfalls durch elementare Abschätzungen, engere Schranken für folgende Beispiele anzugeben:

(a)

$$\begin{array}{rcl} f(n) & = & f(n-1) + f(n-2) \\ & < & f(n-3) + 2f(n-2) \\ & < & 3 \cdot f(n-2) \\ & < & 9 \cdot f(n-4) \\ & < & ... \\ & < & \begin{cases} 3^{\frac{n-1}{2}} \cdot f(1) & \text{, für } n \text{ ungerade} \\ 3^{\frac{n}{2}-1} \cdot f(2) & \text{, für } n \text{ gerade} \end{cases} \\ & \Longrightarrow & f(n) \in O(3^{n/2}) \end{array}$$

(b)

$$\begin{aligned}
\frac{f(n)}{f(n-1)} &= 1+\frac{f(n-2)}{f(n-1)} \\
&= 1+\frac{f(n-2)}{f(n-2)+f(n-3)} \\
&= 1+\frac{1}{1+\frac{f(n-3)}{f(n-2)}} \\
&= 1+\frac{1}{1+\frac{f(n-3)}{f(n-3)+f(n-4)}} \\
&= 1+\frac{1}{1+\frac{1}{1+\frac{f(n-4)}{f(n-3)}}}
\end{aligned}$$

$$\begin{aligned}
\text{Mit } \frac{1}{2} &< \frac{f(n-4)}{f(n-3)} < 1 \\
\text{folgt: } \frac{8}{5} &< \frac{f(n)}{f(n-1)} < \frac{5}{3} \\
\Longrightarrow (\frac{8}{5})^{n-1} &< f(n) < (\frac{5}{3})^{n-1}
\end{aligned}$$

Zwei Algorithmen zur Berechnung der Fibonacci-Zahlen

Zur Berechnung der Fibonacci-Zahlen gibt es nun zwei grundlegend verschiedene Algorithmen, einen *naiven, rekursiven Algorithmus* und einen *iterativen.*

Naiver rekursiver Algorithmus

Hier wird die Definition der Fibonacci-Zahlen einfach übernommen und zu einem Programm umgeschrieben.

```
procedure Fibonacci (n : cardinal): cardinal =
begin
  if n = 0
    then return 0
    elsif n = 1
      return 1
    else return Fibonacci(n-1)+Fibonacci(n-2);
  end;
end Fibonacci;
```

Sei $a > 0$ die Dauer für eine simple **return** `0` - bzw. **return** `1` - Anweisung, dann gilt für die Laufzeit $T(n)$ dieses Algorithmus:

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

Der Zusammenhang von $T(n)$ und $f(n)$ ist offensichtlich, und macht verständlich, warum gilt

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

Folgender Baum, der die rekursiven Aufrufe des Algorithmus für $f(5)$ darstellt, macht auch anschaulich klar, warum dieser Algorithmus so schlecht abschneidet. Man betrachte nur die Wiederholung der Aufrufe $f(0)$ und $f(1)$.

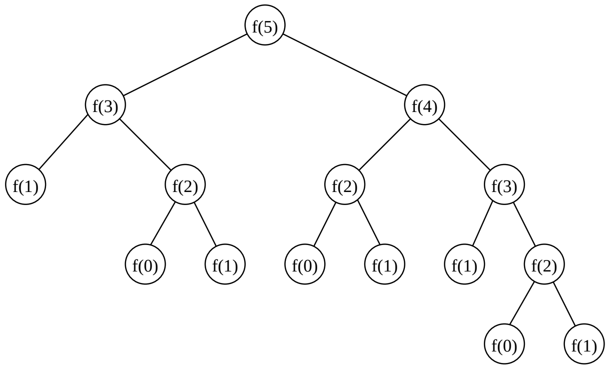


Abbildung 1: Baumdarstellung der Aufrufe des rekursiven Fibonacci-Programms

Iterativer Algorithmus

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

```
procedure Fibonacci( n : cardinal ) : cardinal =
  var fib_new, fib_old, t, i : cardinal
begin
  fib_new := 1; fib_old := 0;
  for i := 1 to (n-1) do
    begin
      t := fib_new;
      fib_new := fib_new + fib_old;
      fib_old := t;
    end;
  return fib_new;
end Fibonacci;
```

Die Laufzeit dieses Algorithmus ist offensichtlich $O(n)$.

Das bedeutet eine erhebliche Verbesserung, ganz davon abgesehen, daß man sich den Rekursions-Overhead (Aufwand zum Verwalten eines Stacks, auf dem rekursive Aufrufe abgelegt werden) der naiven Lösung spart.

Betrachtet man einmal die Laufzeiten für die Berechnung von $f(45)$ (dies ist die größte unter MODULA-3 auf einem 32-Bit-Rechner darstellbare Fibonacci-Zahl), so ergibt sich schon ein beachtlicher Unterschied:

- Iterativ: 45 Zeiteinheiten
- Rekursiv: $2^{23} = 8.400.000$ Zeiteinheiten.

D.h. wenn der iterative Algorithmus die Lösung nach einigen Mikrosekunden liefert, braucht die rekursive Lösung ca. 8,4 Sekunden, was für ein so kleines Problem schon beachtlich ist. Bei Laufzeiten für $n = 100$ wäre der Unterschied noch viel gravierender, der iterative Algorithmus benötigte wieder nur einige Millisekunden, der rekursive hingegen 2^{50} Zeiteinheiten. Das wären also etwa 35702 Jahre für die einfachen Rechenschritte, plus ein paar Jahrtausende zur Abarbeitung des Rekursions-Overheads.

Beispiel: Fakultät

Wir wollen nun noch am Beispiel der Fakultät zeigen, daß in manchen Fällen auch der rekursive Algorithmus, abgesehen vom Rekursions-Overhead, nicht allzuschlecht abschneidet.

Definition der Fakultät:

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

Sei $T(n)$ die Laufzeit des entsprechenden rekursiven Programms:

$$T(n) = \begin{cases} a & n = 0, 1 \\ b + T(n-1) & n > 1 \end{cases} \qquad \text{mit geeigneten } a, b > 0.$$

$$\begin{aligned} T(n) &= b + T(n-1) \\ &= 2 \cdot b + T(n-2) \\ &\vdots \\ &= (n-1) \cdot b + T(1) \\ &= (n-1) \cdot b + a \\ &\in O(n) \end{aligned}$$

$T(n)$ ist linear und damit hat auch die rekursive Implementation „gutartiges“ Verhalten.

1.3 Datenstrukturen

Algorithmen und Datenstrukturen sind unmittelbar miteinander verknüpft und können nicht getrennt voneinander betrachtet werden, da ein Algorithmus mit den Methoden arbeiten muß, die auf einer Datenstruktur vorgegeben sind. Die Datenstruktur ergibt sich meist aus der Problemstellung. Zunächst betrachten wir einige einfache Datentypen.

1.3.1 Datentypen

Man unterscheidet einige grundlegende Datentypen, die man unterteilt in die folgenden funktionalen Gruppen [Güting, Seite 36]:

1. *Elementare (atomare) Datentypen*:
   - **integer** : Ganze Zahlen
   - **cardinal** : Natürliche Zahlen
   - **character** : Zeichen (Buchstaben etc.)
   - **real** : Dezimalzahlen
   - **boolean** : Wahrheitswerte (0 und 1 bzw. TRUE und FALSE)
   - zusätzlich existiert in PASCAL und MODULA-3 noch die Deklaration **type** für
     - Aufzählungstypen (z.B. **type** `Wochentag=(Mo,Di,Mi,Do,Fr,Sa,So);`)
     - Unterbereichstypen (z.B. **type** `Jahr=[1980..1999];`)
2. *Typkonstruktoren*:
   - **array** : Feld
   - **record** (in C: structure): Datensatz, Verbund
   - **set** : existiert zusätzlich in PASCAL für Mengen. Funktioniert aber meist nur bei kleineren Mengen mit weniger als 1024 Elementen.

   Diese Typkonstruktoren lassen einen direkten Speicherzugriff zu, d.h. der Zugriffsaufwand ist konstant, und nicht, wie z.B. bei Listen, abhängig von der Größe des Datensatzes.
3. *Zeigertypen (Pointer)*: Dieser Datentyp ist de facto eine Speicheradresse, unter der die Daten zu finden sind, auf die der Pointer zeigt. Die Verwendung dieses Typs erlaubt *dynamische Datenstrukturen*, also Speicher-Allokation auf Anforderung.

   Warnung: Pointer erfordern sehr viel Disziplin beim Programmieren.

Mit Hilfe der Konstruktoren **array** und **record**, sowie der *Pointer* lassen sich weitere und komplexere Datentypen konstruieren. Einige wichtige davon, die wir in den nächsten Abschnitten besprechen wollen, sind die folgenden:

- **list**: Liste, Sequenz, Folge
- **stack**: LIFO-Memory (Last In First Out), Keller
- **queue**: FIFO-Memory (First In First Out), Schlange
- **tree**: Baum
- **graph**: Graph oder Netzwerk. Sie beschreiben paarweise Beziehungen (Kanten) zwischen Elementen (Knoten) einer Menge.

1.3.2 Listen

Listen werden oft als Verkettung von Pointern (linked lists) implementiert und speichern eine Sequenz oder Folge. Wenn eine Liste nicht verzweigt ist, nennt man sie auch lineare Liste.

Formal definiert man eine Liste wie folgt:

Eine Liste ist eine Folge von Elementen $a(1), \ldots, a(i)$, $a(i+1), \ldots, a(n)$. i identifiziert man mit der Position. Die folgenden Operationen sind auf einer Liste erlaubt:

- **insert**: Einfügen an beliebiger Position.
- **delete**: Entfernen an beliebiger Position.
- **read**: Zugreifen auf beliebige Position.

Es gibt einige Varianten von Listen, die hier nur kurz aufgeführt sind:

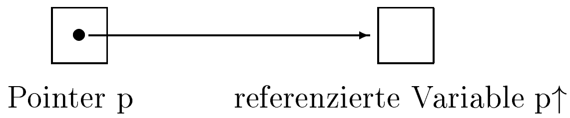
- Zyklische Liste
- Doppelt verkettete Liste
- Beliebig verkettetes „Zeigergeflecht“ .

Bei der Implementation kann man je nach Zielsetzung zwei verschiedene Wege beschreiten: Man kann mit Pointern arbeiten oder mit Arrays.

Pointer-Implementierung von Listen

Das Pointer-Konzept: Ein *Pointer* ist eine Variable, deren Inhalt eine Speicherplatzadresse einer anderen Variablen ist. Die Variable, auf die der Pointer zeigt heißt *referenzierte Variable*. Der Typ der referenzierten Variablen heißt *Bezugstyp*.

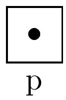
Pointer und referenzierte Variable sind zwei unterschiedliche Objekte. Daraus folgt, daß nicht zu jedem Zeitpunkt eine referenzierte Variable zu einem Pointer existieren muß. Es kann auch sein, daß kein Pointer mehr auf eine referenzierte Variable zeigt. Letzteres sollte man aber vermeiden.



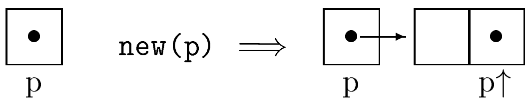
Die elementaren Operatoren auf Pointern:

- **ref** - *Erzeugung eines Pointers*: Eine Pointervariable wird erzeugt, indem man eine Variable als Pointer auf einen Bezugstyp deklariert.

  In PASCAL dient dazu die Funktion ↑ `Bezugstyp` und in MODULA-2 **POINTER TO** `Bezugstyp` .

  var `p:` **ref** `Bezugstyp` $\Longrightarrow$ 

- **new**`(p)` - *Erzeugung einer Variablen vom Bezugstyp*: Diese Funktion allociert zur Laufzeit, also dynamisch, Speicherplatz für eine Variable vom Bezugstyp und weist dem Pointer die entsprechende Adresse zu. Der Wert der referenzierten Variablen p↑ ist wohlgemerkt noch unbestimmt.



- **dispose**`(p)` - *Freigabe einer Variablen vom Bezugstyp*: Diese Funktion arbeitet wie **new** ebenfalls dynamisch mit der Wirkung, daß der Speicherplatz von p↑ freigegeben wird und der Pointer p den Wert NIL erhält.

  Anmerkung: Diese Aufgabe übergibt MODULA-3 dem Betriebssystem, so daß man den Speicher nur noch per **dispose** freigeben muß, falls man zuvor für die entsprechende Variable explizit die systemeigene Garbage-Collection deaktiviert hat.

- Wertzuweisungen an einen Pointer: Außer mit den beiden oben genannten Funktionen kann p wie folgt direkt ein Wert zugewiesen werden:

  Zuweisung: `p := NIL`
  Zuweisung: `p := q` dazu müssen p und q denselben Bezugstyp besitzen.

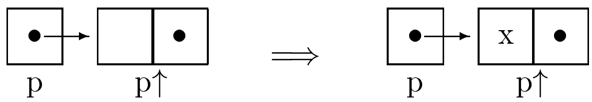
Zur Darstellung einer Liste in Pointer-Implementierung vereinbart man einen Bezugstyp, der sich zusammensetzt aus einem Feld *key*, das den Inhalt des Listenelements trägt, und einem Pointer *next* auf das nächste Element der Liste.

```
type link = ref node;
     node = record
```

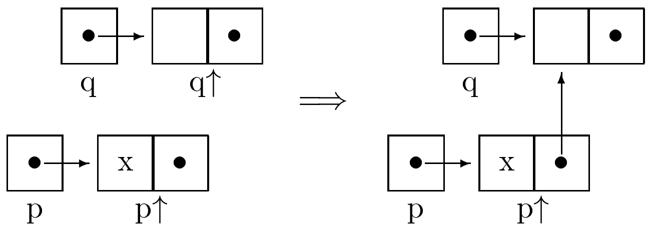
```
            key  : integer;
            next: link;
          end;
var p,q : link;
```

Zuweisungen können dann folgendermaßen dargestellt werden:

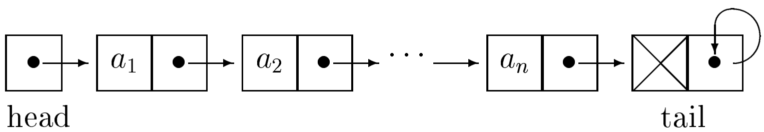
1. Zuweisung p↑.key := x



2. Zuweisung p↑.next := q , wobei q den gleichen Typ wie p↑.next haben muß.

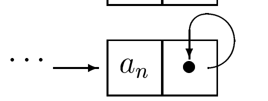


Eine einfach verkettete Liste sieht dann so aus:

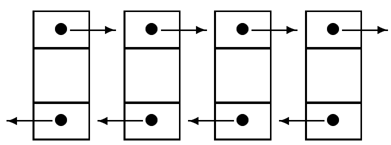


Bei der Darstellung der head- und tail-Elemente gibt es einige Varianten. So benötigt man nicht unbedingt ein explizites head-Element, und bei den tail-Elementen sind, außer der obigen „Dummy“-Variante noch folgende Varianten gängig:

NIL-Pointer:



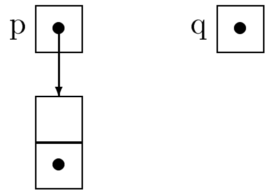
Eine doppelt verkettete Liste lässt sich so darstellen:



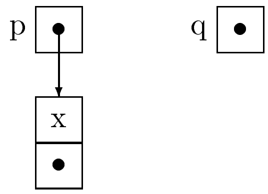
Beispiel:

var p,q: link;

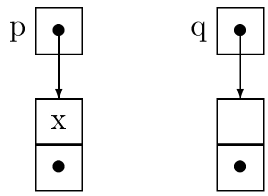
new (p);



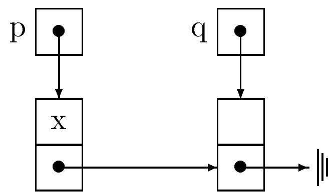
p↑.key:=x;



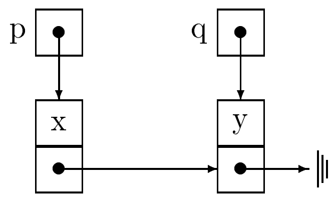
new (q);



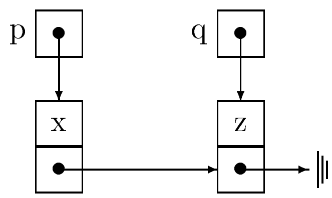
p↑.next:=q;
q↑.next:=NIL;



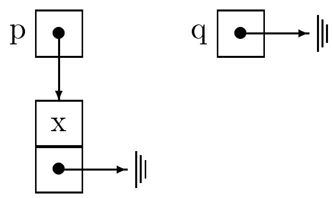
p↑.next↑.key:=y;



q↑.key:=z;



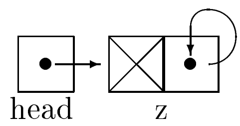
p↑.next:=NIL;
dispose(q);



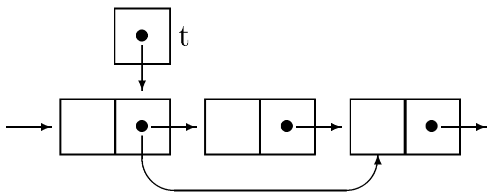
Programme [Sedgewick 88]:
Die Listenmethoden *InsertAfter* und *DeleteNext* lassen sich dann wie folgt implementieren. Dabei ist der Zeiger z die Position in der Liste:

```
type link = ref node;
     node = record
              key : integer;
              next: link;
            end;
var head,z : link;
```

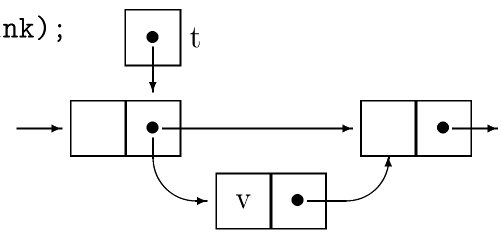
```
procedure ListInitialize();
begin
  new(head);
  new(z);
  head↑.next := z;
  z↑.next := z;
end ListInitialize;
```




```
procedure DeleteNext(t : link);
begin
  t↑.next := t↑.next↑.next;
end DeleteNext;
```




```
procedure InsertAfter(v : integer; t : link);
  var x : link;
begin
  new(x);
  x↑.key := v;
  x↑.next := t↑.next;
  t↑.next := x;
end InsertAfter;
```



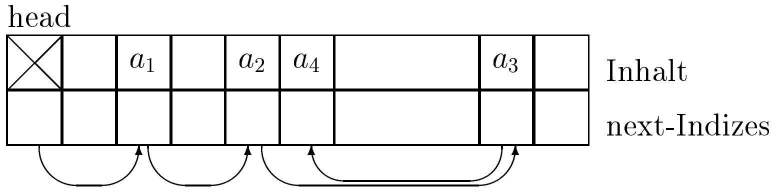
Array-Implementierung von Listen

Bei der Array-Implementierung kommen wiederum zwei verschiedene Lösungen in Betracht: Ein sequentielles Array, das sich aber als viel zu starr erweist, und eine Cursor-Darstellung, die weitgehend an die Pointer-Implementierung angelehnt ist, und ein zweites Array für die Indizes benutzt.

1. sequentiell

a_1	a_2	a_3	a_4						

2. Cursor-Darstellung



Die Listenmethoden *InsertAfter* und *DeleteNext* lassen sich in der Cursor-Darstellung dann wie folgt implementieren. Dabei ist z wieder die Position in der Liste, und x die Position im Array:

```
var key,next : array [0..N] of integer;
    x,head,z : integer;


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: integer; t: integer);
begin
  x := x + 1;
  key[x] := v; next[x] := next[t];
  next[t] := x;
end InsertAfter;
```

Beachte:

- Overflow und Underflow der Arrays/Listen müssen abgefangen werden.

- Mit Hilfe einer Freispeicherliste können bereits benutzte und wieder freigegebene Feldpositionen wiederverwendet werden.

Je nach Zielsetzung ist abzuwägen, welcher Implementierung man den Vorrang gibt, denn jede Implementierung hat unterschiedliche Vor- und Nachteile:

1. sequentielles Array
   (−) sehr starr: Delete und Insert sehr umständlich
   (−) nur bei „statischer“ Objektmenge (also ohne Operationen) zu empfehlen
2. „Cursor-Darstellung“
   (−) vorgegebene maximale Array-Größe
   (−) Freispeicherverwaltung erforderlich
   (+) kein Overhead
   (+) Fehlerkontrolle leichter
3. Pointer-Implementierung:
   (+) maximale Anzahl der Elemente offen
   (+) keine Freispeicherverwaltung
   (−) Overhead durch Systemaufrufe
   (−) Fehlerkontrolle schwierig

1.3.3 Stacks und Queues

Stacks (Stapel) und Queues (Schlangen) sind Listen mit eingeschränkten Operationen. Es kann nicht mehr an beliebiger Stelle in der Liste eingefügt und entfernt werden, sondern nur noch am Anfang bzw. Ende.

Stack

Ein Stack wird auch LIFO-Memory (Last In First Out) genannt. *Top* bezeichnet die erste freie Stelle über dem obersten Element. Beim Stack ist das einfügen und entfernen nur wie folgt möglich:

- einfügen (*push*) nur am Ende (Top)
- entfernen (*pop*) ebenfalls nur am Ende

pop liefert den Wert des obersten Elementes zurück und entfernt dieses zugleich aus dem Stack. Geläufige Namen für einen Stack sind auch: pushdown stack, Keller-Speicher und Stapelkeller.

Nun wollen wir einen Stack in einer Programmiersprache realisieren (**Programme** [Sedgewick 93]):

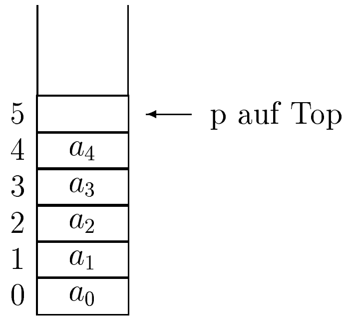


Abbildung 2: Stack

1. **Implementation mit Pointern**: Ein Stack hat die gleiche Datenstruktur wie eine Liste, nur die Operationen unterscheiden sich folgendermaßen:

```
procedure push(v : integer) =
  var t : link;
  begin
    new(t);
    t↑.key := v; t↑.next := head↑.next;
    head↑.next := t;
  end push;

procedure pop() : integer =
  begin
    return head↑.next↑.key;
    head↑.next := head↑.next↑.next;
  end pop;

procedure stackempty() : boolean =
  begin return (head↑.next = z); end stackempty;
```

2. **Implementation mit Arrays** (Cursor-Darstellung)

```
const maxP = 100;
var stack : array[0..maxP] of integer;
        p : integer;

procedure stackinit() =
  begin p := 0; end stackinit;

procedure push (v : integer) =
  begin
    stack[p] := v; p := p + 1;
  end push;

procedure pop() : integer =
```

```
    begin
      p := p - 1; return stack[p];
    end pop;

  procedure stackempty() : boolean =
    begin return (p <= 0); end stackempty;

  procedure stackfull() : boolean =
    begin return (p > maxP) end stackfull;
```

Beachte: Over- und Underflow der Indexgrenzen müssen abgefangen werden.

Queue

Eine Queue wird auch FIFO-Memory (First In First Out) genannt. Bei der Queue ist das einfügen und entfernen nur wie folgt möglich:

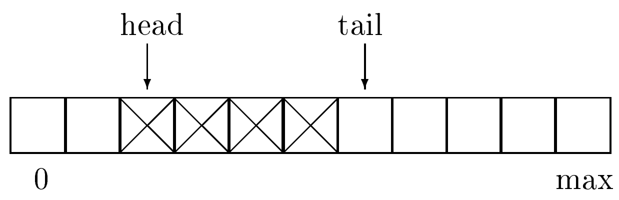
- einfügen (*put*) nur am Ende
- entfernen (*get*) nur am Anfang

get liefert den Wert des ersten Elementes zurück und entfernt dieses Zugleich aus der Schlange. Geläufige Namen für eine Queue sind auch: Warteschlange und Ringspeicher.

Implementation einer Queue als Array (**Programme** [Sedgewick 93]):

```
const Nmax = 100
var head,tail : integer;
        queue : array[0..Nmax] of integer;

procedure queueinitialize() =
  begin
    head := 0; tail := 0;
  end queueinitialize;

procedure put (v : integer) =
  begin
    queue[tail] := v; tail := tail + 1;
    if tail > max then tail := 0;
  end put;

procedure get() : integer =
  var buffer : integer;
  begin
    buffer := queue[head];
```

```
    head := head + 1;
    if head > max then head := 0;
    return buffer;
  end get;

procedure queueempty() : boolean =
  begin return (head = tail); end queueempty;

procedure queuefull() : boolean =
  begin return (head = (tail+1)mod(max+1)); end queuefull;
```



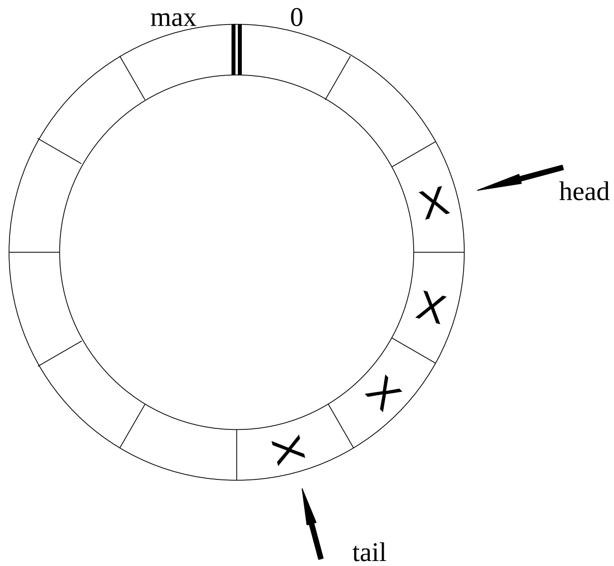


Abbildung 3: Ringpuffer

1.3.4 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. Daraus folgt, daß außer den Blattknoten jeder Knoten mindestens einen unmittelbaren Nachfolger hat.
- *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, d.h. 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.

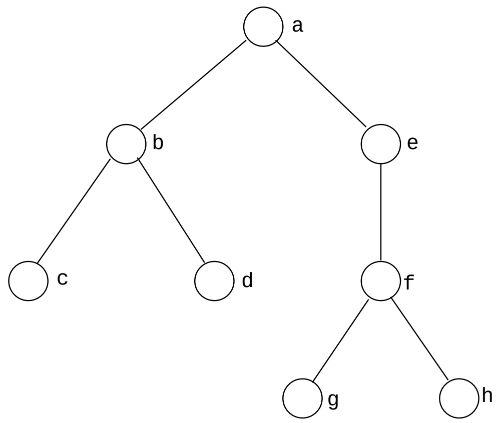


Abbildung 4: Beispiel für einen Baum

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)

Bäume können unterschiedlich dargestellt werden, die folgenden vier Darstellungen sind gängige Beschreibungen derselben hierarchischen Struktur:

1. Baum

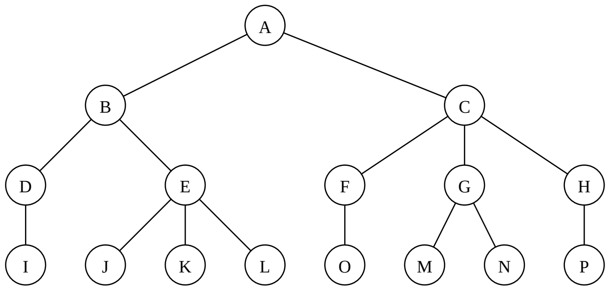


Abbildung 5: Graph

2. geschachtelte Mengen

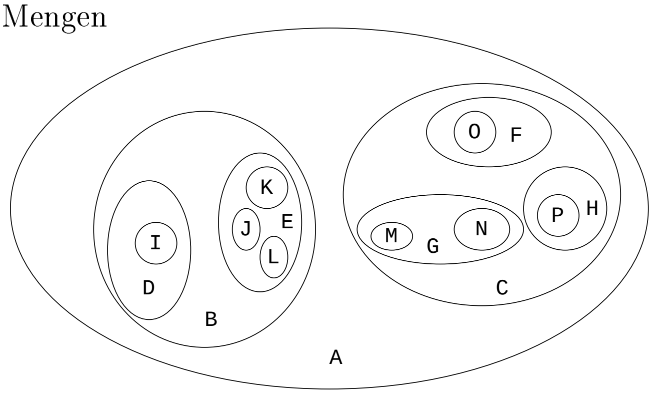


Abbildung 6: geschachtelte Mengen

3. Einrückung

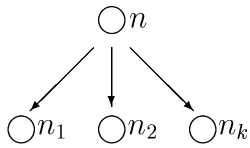
A
    B
        D
            I
        E
            J
            K
            L
    C
        F
            O
        G
            M
            N
        H
            P

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

Induktive Definition eines Baumes

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

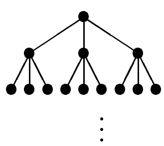
Sei T ein Baum mit Grad $d \geq 2$ und n Knoten, dann gilt:

- maximale Höhe $= n - 1$
- minimale Höhe $= \lceil \log_d (n(d-1)+1) \rceil - 1 \leq \lceil \log_d n \rceil$,
  mit $\lceil x \rceil =$ nächst höhere ganze Zahl $\geq x$ für $x \in \mathbb{R}$:

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 entartet in eine lineare Liste.
- 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
  $\vdots$

  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“ außer der letzten alle 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 $h-1$.

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

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

$$h = \lceil \operatorname{ld}(n+1) \rceil - 1$$

Implementierung eines Binärbaumes

Für allgemeine Bäume sind die Darstellungen im Gegensatz zu Binärbäumen wenig standardisiert, deswegen widmen wir uns zunächst letzteren.

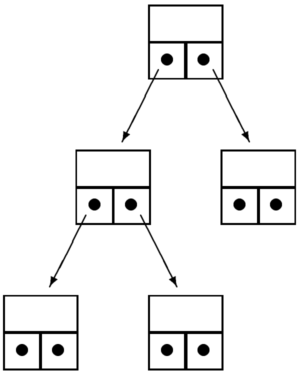
Bei den Binärbäumen existieren, ähnlich zu den Listen-Implementationen, die Array-Einbettung und die Pointer-Implementation. Die Wahl der Implementierung ist nach den gleichen Kriterien zu treffen, wie schon bei den Listen.

1. **Pointer-Realisierung:**
   Umsetzung in Programm-Code:

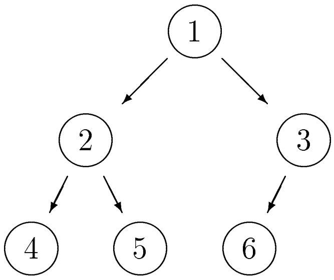
```
type node = record
              key : elem;
              left, right : ref node;
            end;
```

Ähnlich den unterschiedlichen Varianten zur Behandlung eines Listenendes, muß auch hier noch eine Vereinbarung zur Behandlung der Blätter getroffen werden.

Ein Binärbaum stellt sich dann so dar:



2. **Array-Einbettung:** nur bei vollständigen Bäumen empfehlenswert. Diese können dann später auch zu „Heaps“ erweitert werden. Ein vollständiger Baum ist dabei folgendermaßen definiert:
   Definition: 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).

   Nachfolger und Vorgänger lassen sich dann auf folgende Weise einfach „berechnen“:

   $$\text{Predecececessor}: \; Pred(n) = \frac{n}{2}$$

   $$\text{Successor}: \quad Succ(n) = \begin{cases} 2n & \text{left son} \\ 2n+1 & \text{right son} \end{cases} \quad \text{(falls diese existieren)}$$

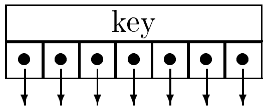
   Man prüft die Existenz der Nachfolger über die Arraygrenzen.

Implementierung eines allgemeinen Baumes

Wie schon erwähnt sind Darstellungen für allgemeine Bäume wenig standardisiert. Einige davon sind:

- Darstellung durch Knoten, die ein **Array von Pointern** auf ihre Nachfolgeknoten enthalten. Diese Darstellung empfiehlt sich bei fest vorgegebenem Grad.

type `node` **=record**
`key: elem;`
`sons :` **array** `[1..d]` **of ref** `node;`
end;



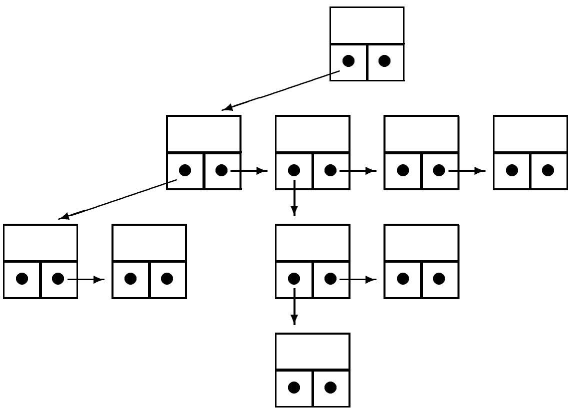
Nachteile:

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

- Darstellung durch **Binärbaum**: „LeftMostChild, RightSibling"

type `node` = **record**
`key: elem;`
`leftmostchild :` **ref** `node;`
`rightsibling :` **ref** `node;`
end;

  - $leftmostchild$ = head der verketteten Liste der Geschwister mit „Next-Pointer" *rightsibling*.



  - Platzbedarf:
    #Knoten · (2· sizeof(↑node) + sizeof(elem)),
    wobei sizeof(x) der für x notwendige Speicherplatz ist.

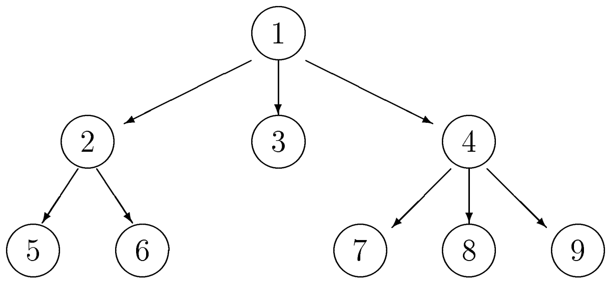
- **Reine Array-Implementierungen:**

  1. `array` : Parent[1..N] (Spezialfall: Wurzel)

2. Knoten mit „breadth first“ durchnumeriert:

```
array
  LeftMostChild [1..N],
  RightMostChild [1..N];
```

Zur Veranschaulichung:



Konvention sei, daß für „nicht vorhanden“ -1 in das Array eingetragen wird. Dann gestaltet sich ein solcher Baum so:

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

In einem solchen Array sind die Durchlauf-Funktionen sehr einfach:

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

1.3.5 Tree Traversal (Baum-Durchlauf)

Für viele komplexere Anwendungen in Bäumen ist ein kompletter Durchlauf aller Knoten eines Baumes notwendig. Wie wir schon gesehen haben, ist bei Array-Implementierung oft schon ein Baumdurchlauf mittels einfacher Schleife über die Knotennummern gegeben. Anders ist das bei den Pointer-Implementierungen, dort ist kein festes Schema vorgegeben, aber es gibt einige Standard-Durchläufe:

- **Tiefendurchlauf** (Depth-First): Zu einem Knoten n werden rekursiv seine Teilbäume $n_1...n_k$ durchlaufen. Je nach Reihenfolge in der die Teilbäume durchlaufen 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 in der Regel nur für Binärbäume gebräuchlich.

- **Breitendurchlauf** (Breadth-First): Knoten werden ihrer Tiefe nach gelistet, und zwar bei gleicher Tiefe von links nach rechts.

Rekursiver Durchlauf für einen Binärbaum

Wir wollen obige Tiefendurchläufe für einen Binärbaum implementieren, der folgendermaßen dargestellt ist:

```
type node = record
               key : elem;
               left, right : ref node;
            end;
```

1. **Preorder-Durchlauf**:

```
procedure traverse(t : node) =
begin
  if t<>z then             (* Abfrage, ob t kein Blatt *)
    visit(t);
    traverse(t↑.left);
    traverse(t↑.right);
  end;
end traverse;
```

2. **Postorder-Durchlauf**:

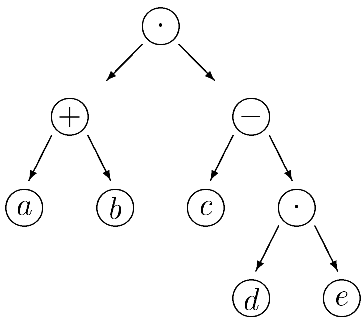
```
procedure traverse(t : node) =
begin
  if t<>z then
    traverse(t↑.left);
    traverse(t↑.right);
    visit(t);
  end;
end traverse;
```

3. **Inorder-Durchlauf**:

```
procedure traverse(t : node) =
begin
  if t<>z then
    traverse(t↑.left);
    visit(t);
    traverse(t↑.right);
  end;
end traverse;
```

Traversieren eines Operatorbaumes

Bei Anwendung auf Operatorbäume realisieren die Traversierungsarten eine Darstellung von algebraischen Ausdrücken in unterschiedlichen Notationen (die für rechnerinterne Darstellung besonders vorteilhaft sind). Betrachten wir den zu dem Ausdruck $(a+b)\cdot(c-d\cdot e)$ gehörenden Operatorbaum.



Dann liefert eine Ausgabe aller Knoten mittels der

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

Nicht-rekursiver Durchlauf für einen Binärbaum

1. **Preorder Traversal** (mit Stack)

```
procedure traverse (t : node) =
  begin
    push (t);
    repeat
      t := pop;
      visit (t);
      if t<>z then push (t↑.right);
      if t<>z then push (t↑.left);
    until stackempty;
  end traverse;
```

2. **Level-order Traversal** auch „Breadth First“ (mit Queue)

```
procedure traverse (t : node) =
  begin
    put(t);
    repeat
      t := get();
      visit (t);
      if t<>z then put (t↑.left);
      if t<>z then put (t↑.right);
    until queueempty;
  end traverse;
```

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. Hier nun einige typische Methoden:

1. **Divide and Conquer** [Aho et al. 74]
2. **Rekursion**: Sollte bekanntlich mit Vorsicht eingesetzt werden. Ein gutes Beispiel sind die Baum-Durchläufe.
3. **Rekursionsgleichungen** [Aho et al. 74, Sedgewick 88]
4. **Dynamische Programmierung** [Aho et al. 74]

1.4.1 „Divide and Conquer“-Strategie

Die „Divide and Conquer“-Strategie wird sehr häufig benutzt, weil sie vielseitig ist, und sich an fast jedem Problem, welches sich in kleinere Teilprobleme zerlegen läßt anwenden läßt.
Die Strategie besagt:

1. **Divide**: Zerlege das Problem in Teilprobleme, wenn es nicht trivial ist.
2. **Conquer**: Löse Teilprobleme auf dieselbe Art, und setze die Teillösungen zur Gesamtlösung zusammen (Merge).

Beispiel: Gleichzeitige Bestimmung von MAX und MIN einer Menge oder Folge [Aho et al. 74, Seite 61]

```
0:procedure MAXMIN (S);
1:if ||S||=2 then
2:  begin
3:    let S={a,b};
4:    return (MAX (a, b), MIN (a, b));
5:  end
6:  else
7:    begin
8:      divide S into two subsets S1 and S2, each with half the elements;
9:      (max1,min1) ← MAXMIN(S1);
10:     (max2,min2) ← MAXMIN(S2);
11:     return (MAX (max1, max2), MIN (min1, min2));
12:   end;
13: end MAXMIN;
```

Sei $T(n)$ die Zahl der Vergleiche, dann gilt für die gewählte Implementierung:

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

Erklärung: $n = 2$: klar
$n > 2$: 2 Aufrufe von MAXMIN mit $\frac{n}{2}$- elementigen Mengen (Zeilen 9 und 10)
und 2 Vergleiche (Zeile 11), die aber in $O(1)$ ablaufen.

Behauptung: $T(n) = \frac{3}{2}n - 1$

Verifikation:

$$T(2) \;=\; \frac{3}{2} \cdot 2 - 1 = 2 \qquad \checkmark$$

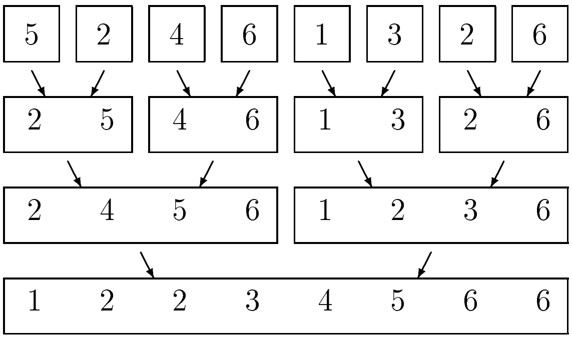
$$
\begin{aligned}
2 \cdot T\left(\frac{n}{2}\right) + 1 &= 2 \cdot \left[\frac{3}{2} \cdot \frac{n}{2} - 1\right] + 1 \\
&= \frac{3}{2} n - 1 = T(n) \quad \checkmark
\end{aligned}
$$

Beispiel: MergeSort

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

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

Das „bottom up“ oder auch „2-way-straight“ MergeSort-Verfahren, überspringt den Divide-Schritt, und versteht die Eingabefolge direkt als Menge von einelementigen Teilfolgen. Diese MergeSort-Variante wollen wir nun betrachten:



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 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:

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

Exakte Analyse von MergeSort

Sei $T(n)$ die exakte 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 Rekursionsgleichung gilt [Mehlhorn, Seite 57]:

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

1.4.2 Rekursionsgleichungen

Rekursionsgleichungen sind oft einfacher zu lösen für $n = 2^k$, $k \in \mathbb{N}$. Denn dann sind viele Rekursionsgleichungen durch *Sukzessives Einsetzen* zu lösen:

1.

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

2.

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

3.

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

4.

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

5.

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

Verifiziere: $2n-1 = 2\cdot(n-1)+1 \quad \surd$

Allgemeiner Typ von Rekursionsgleichungen [Aho et al. 83]:

$$T(n)=\begin{cases} 1 & n=1\\ a\cdot T\left(\frac{n}{b}\right)+d(n) & n>1\end{cases}$$

mit Konstanten $a>0$ und $b>0$.

Ist $n \to T(n)$ eine stetige Funktion, dann können wir vor allem die Werte für $n=b^k$, $k\in\mathbb{N}$ betrachten:

$$T\left(\frac{n}{b^i}\right) = aT\left(\frac{n}{b^{i+1}}\right)+d\left(\frac{n}{b^i}\right) \text{ klar mit } \frac{n}{b^i} \text{ statt } n.$$

Sukzessives Einsetzen liefert dann:

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

Nun wählen wir für k einen von n abhängigen Wert k_n: Damit folgt aus $n=b^{k_n}$, daß $k_n=\log_b n$ und somit:

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

$$
\begin{aligned}
&= a^{\log_b n} + \sum_{j=0}^{k_n-1} a^j d(b^{k_n-j}) \\
&= (b^{\log_b a})^{\log_b n} + \sum_{j=0}^{k_n-1} a^j d(b^{k_n-j}) \\
&= (b^{\log_b n})^{\log_b a} + \sum_{j=0}^{k_n-1} a^j d(b^{k_n-j}) \\
&= n^{\log_b a} + \sum_{j=0}^{k_n-1} a^j d(b^{k_n-j})
\end{aligned}
$$

Der Fall einer allgemeinen Funktion $n \to d(n)$ ist umständlicher, aber nach [Cormen et al.] auch lösbar. Dazu wird die allgemeine Funktion $n \to d(n)$ auf den Fall der speziellen Funktion $n \to n^\gamma$ zurückgeführt, indem man das Wachstum dieser beiden Funktionen miteinander vergleicht („Master Theorem").

Im folgenden wollen wir nur noch den speziellen Fall $n \to n^\gamma$ betrachten, an dem sich das wesentliche Ergebnis zeigen läßt. So erhalten wir eine Summe, die sich wie folgt geschlossen berechnen läßt:

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

Beachte: $a^{k_n} = n^{\log_b a}$
$b^{k_n} = b^{\log_b n} = n$

Nun nehmen wir folgende Fallunterscheidung vor:

- $a < b^\gamma$: geometrische Reihe konvergiert für $k \to \infty$:

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

Wegen $\log_b a < \gamma$ folgt somit: $T(n) \in O(n^\gamma)$

- $a = b^\gamma$:

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

  Und somit folgt: $T(n) \in O(n^\gamma \log_b n)$

- $a > b^\gamma$:

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

  Wegen $a^{k_n} > (b^\gamma)^{k_n}$ folgt: $T(n) \in O(n^\gamma)$

1.4.3 Dynamische Programmierung

Grundvoraussetzung für die dynamische Programmierung ist ein Optimierungsproblem, das sich in Teilprobleme zerlegen läßt, denn das Grundkonzept der DP ist eine Divide-and-Conquer-Strategie, die folgendes Vorgehen beinhaltet:

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

Charakteristisch ist also eine **sukzessive Vorgehensweise**. Ausgehend von Elementarproblemen werden nach und nach größere Teilprobleme bearbeitet, bis schließlich die Gesamtlösung betrachtet werden kann.

Terminologie (Richard Bellman 1957):

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

Beispiel: Kette von Matrizenprodukten

Nach dem Assoziativgesetz für Matrizenprodukte ist das Ergebnis der Berechnung nicht abhängig von der Klammerung. Sehr wohl aber der Rechenaufwand. Nun ist also die Frage: Welche Klammerung ist am günstigsten, d.h. erfordert den geringsten Rechenaufwand?

Zunächst zum Aufwand der Berechnung eines Matrizenproduktes A zweier Matrizen B und C.

$B[n, i]$ beschreibe B als $n \times i$-Matrix.

Da für jedes Element a_{uv} der Matrix $A[n,m] = B[n,i] \cdot C[i,m]$ gilt

$$a_{uv} = \sum_j b_{uj} \cdot c_{jv}$$

ergibt sich als Anzahl der zur Berechnung nötigen Operationen (Additionen und Multiplikationen): $n \cdot i \cdot m$.

Seien M_i, $i = 1, \dots, 4$ reelle Matrizen, dann interessiert uns der Rechenaufwand zur Berechnung des Matrizenproduktes

$A = M_1[10,20] \cdot M_2[20,50] \cdot M_3[50,1] \cdot M_4[1,100]$

für zwei verschiedene Klammerungen:

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

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

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

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

Das Problem besteht nun in der Suche nach der besten Klammerung. Eine vollständige Suche (exhaustive search) hat eine **Komplexität, die exponentiell zur Zahl der Matrizen n wächst**.

Ansatz der dynamischen Programmierung

Definiere zunächst eine Hilfsgröße m_{ij}. Sie beschreibt die minimale Zahl von Operationen zur Berechnung von

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

$$\begin{array}{ccccc} \underbrace{(M_i \cdot M_{i+1} \cdot \ldots \cdot M_k)} & \cdot & \underbrace{(M_{k+1} \cdot M_{k+2} \cdot \ldots \cdot M_j)} & \text{mit} & M_i \in \mathbb{R}^{r_{i-1} \times r_i} \\ m_{ik} & + & m_{(k+1)j} & + & r_{i-1} \cdot r_k \cdot r_j \end{array}$$

Dann gilt aufgrund der Definition von m_{ij} und der Rekursionsgleichung (recurrence relation; DP equation):

$$m_{ij} = \begin{cases} 0 & j = i \\ \min\limits_{i \leq k < j} \{m_{ik} + m_{(k+1)j} + r_{i-1} \cdot r_k \cdot r_j\} & j > i \end{cases}$$

Die Reihenfolge der Auswertung wird bestimmt von einem Längenindex l:

$$\begin{aligned}
l &= 1 : m[i,i] \qquad \forall\, i = 1,\ldots,N \\
l &= 2 : m[i,i+1] \qquad \forall\, i = 1,\ldots,N \\
l &= 3 : m[i,i+2] \qquad \forall\, i = 1,\ldots,N \\
&\vdots \\
l &= N-1 : m[i,i+N-2] \qquad \forall\, i = 1,\ldots,N \\
l &= N : m[i,i+N-1] \qquad \forall\, i = 1,\ldots,N
\end{aligned}$$

Dies läßt sich in einem Programm (Pseudo-Code) folgendermaßen realisieren:

```
const N = 100;
var r : array [0..N] of integer;
    m,s : array [1..N],[1..N] of integer;
begin
  for i := 1 to N
    m[i,i] = 0;
  for l := 2 to (N)            (* Laengenindex *)
    for i := 1 to (N-l+1)      (*Positionsindex*)
      begin
        j := i+l-1;
        m[i,j] :=     min      { m[i,k] + m[k+1,j] + r[i-1] · r[k] · r[j] };
                    i≤k<j
        s[i,j] := arg min      { m[i,k] + m[k+1,j] + r[i-1] · r[k] · r[j] };
                    i≤k<j
      end;
  end;
```

Ergebnis:

- m[1,N]: Unter diesem Eintrag in der Matrix m findet man die Anzahl der minimal nötigen Operationen.
- s[1,N]: Unter diesem Eintrag der Matrix 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 in der Matrix nachschauen.
- Die Komplextät ist dann für die iterative Bottom-up Methode $O(N^3)$ und für die rekursive Top-down Methode $O(2^N)$.

1.4.4 Memoization: Tabellierung von Zwischenwerten

Die mögliche Ineffizienz direkter rekursiver Implementierung kann oft durch Memoization vermieden werden, indem Zwischenwerte in Tabellen (Arrays) gespeichert werden.
Beispiele:

- Matrixprodukt-Problem
- Fibonacci-Zahlen:

```
const Nmax = 100;
type  table := array [0..Nmax] of integer;
var   F: table{-1, ..}; /* initialisiert alle Werte in Matrix F mit -1*/

procedure fib(n : integer) : integer =
  begin
    if F[n] > -1 then return F[n]; end;
    if n = 0 then F[n] := 0; end;
    if n = 1 then F[n] := 1; end;
    if n > 1 then F[n] := fib(n-1) + fib(n-2) end;
    return F[n];
  end;

  Zur Berechnung aller Fibonacci-Zahlen ruft man die Funktion
  in einer Schleife auf:

  for i=1 to Nmax fib(i);
```

Übung: Entwickeln Sie ein Programm für die Matrixprodukte, das auf Memoization basiert.

1.5 RAM (Random Access Machine)

Um genaue und maschinenunabhängige Aussagen zur Komplexität eines Programms zu machen und dieses dann mit anderen vergleichen zu können, benötigt man einen Rechner als Maßstab.
Dazu eignet sich am besten ein axiomatisch definiertes Rechnermodell, wie es die "verallgemeinerte Registermaschine" (RAM) darstellt.
Eine RAM besteht aus:

- adressierbaren Speicherzellen (random access registers)
- Ein- und Ausgabeband, deren Inhalt nur Integer-Werte sind und die beide nur von links nach rechts bewegt werden können.

- zentraler Recheneinheit mit:
  - Akkumulator
  - Program Counter (Befehlszähler)
- Programm, das nicht im Speicher steht und sich nicht selbst verändern kann.
- Befehlssatz, in abstraktem Assembler verfaßt. Er ist sehr rudimentär und akzeptiert meist nur Integer-Datentypen.
  - LOAD: immediate/direct/indirect
  - ADD, SUB
  - (MUL, DIV): nicht immer
  - JUMP on (Accu=0), JUMP on (Accu>0)
  - JUMP
  - READ, WRITE

Einzelheiten können variieren, so ist oft der Akkumulator kein eigener Speicher, sondern einfach Speicherzelle Nr. 0. Variationen gibt es auch beim Befehlssatz u.v.m.

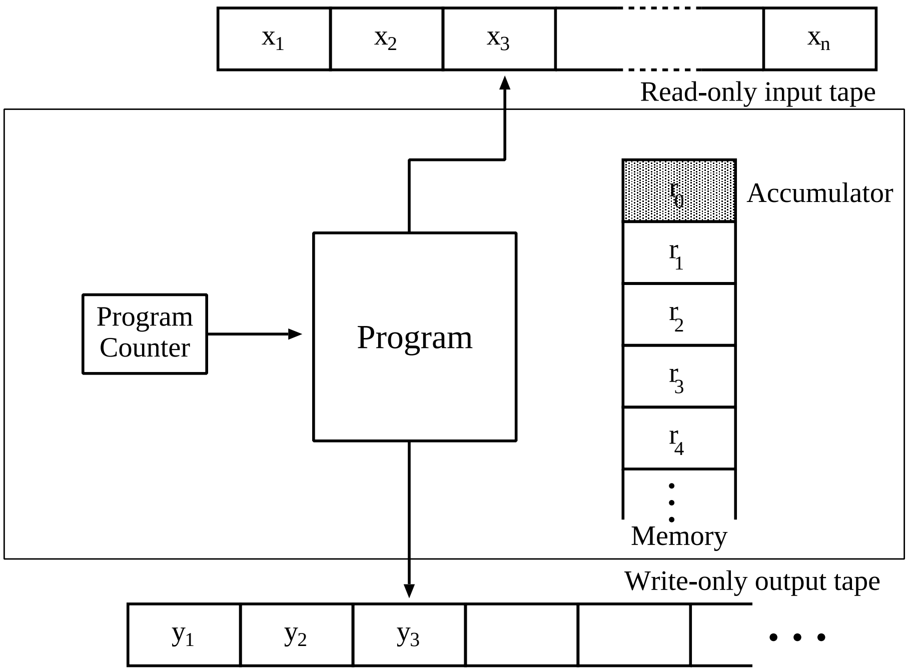


Abbildung 7: **RAM** (Random Access Machine)

Anmerkung: Die modernen RISC (Reduced Instruction Set Computer)-Prozessoren sind in ihrer Architektur einer RAM deutlich ähnlicher als die älteren CISC (Complex Instruction Set Computer)- Prozessoren.

1.5.1 RAM: Kostenabschätzung

Je nach Zielsetzung, RAM und angestrebter Präzision der Aussage kann man zwischen zwei Arten des Kostenmaßes wählen:

- Einheitskostenmaß: Es ist unabhängig von der Wortlänge des Operanden, sondern nur von der Art des Befehls (s.u.).
- Logarithmisches Kostenmaß: Es macht die Kosten abhängig von der Länge des Operanden, d.h. der Wortlänge des Rechners = Anzahl der Bits.

  Das logarithmische Kostenmaß sollte verwendet werden, wenn ein Operand mehr als eine Speicherzelle belegt (bei unserer RAM nicht der Fall).

Die Kosten, die einem Befehl im Einheitskostenmaß zugeordnet werden, sind wenig standardisiert. Man versucht dabei eine möglichst realistische Schätzung zu finden. Grob kann man die Befehle in unterschiedliche Kostenklassen unterteilen:

1. Befehl ohne Memory Access
2. Befehl mit Memory-Access (ADD, MOV, ...)
3. Befehl mit indirektem Memory-Access
4. MUL, DIV: können teurer sein
5. Loop(repeat, while,...): n-mal
   n mal die Kosten für die Befehle innerhalb der Schleife

Die RAM erfaßt (ziemlich) genau die im realen Rechner anfallenden Memory-Operationen, dabei werden externe Speicher wie Platten (zunächst) nicht betrachtet.

1.5.2 Rekursion

Auf der RAM muß eine Rekursion durch eine nicht-rekursive Implementierung ersetzt werden. Dies geschieht mittels eines Stacks. Dazu wendet man folgende Methode an:

1. Jeder Aufruf (Inkarnation, Aktivierung) produziert einen sogenannten Aktivierungsblock (Versorgungsblock) als einen Speicherbereich auf dem Stack. Er besteht aus:

   - aktuellen Parametern (werden vom rufenden Programm bereitgestellt)
   - lokalen Variablen
   - Rücksprungadresse

   Beachte:

   - Der Zugriff auf die aktuellen Parameter (call by value) und die lokalen Variablen erfolgt über den Stack.

- Der Aktivierungsblock wird angelegt für jeden individuellen Aufruf.

2. Nach Beendigung der Prozedur wird der zugehörige Aktivierungsblock entfernt und die Kontrolle an das rufende Programm zurückgegeben.

Beachte: Es wird angenommen, daß globale Variable nur aus dem Hauptprogramm kommen, d.h. es gibt keine globalen Variablen, die über Prozedur-Schachtelung definiert werden (wie bei MODULA-3 möglich). Bei Sprachen wie C ist diese Prozedur-Schachtelung ohnehin nicht möglich.

Ein „intelligenter“ Compiler ersetzt „Endrekursion“ (tail recursion) durch Iteration (Beispiel: Fakultät).

1.6 Abstrakte Datentypen (ADT)

[Sedgewick 88, Seite 31]
Wenn man Datenstrukturen, wie die bisher besprochenen Liste, Stack, etc, nur durch die auf ihnen zugelassenen Operationen definiert, und die spezielle Implementierung ignoriert, dann gelangt man zu den sogenannten „Abstrakten Datentypen“.

Ein „Abstrakter Datentyp“ (ADT) besteht dann aus:

1. einer Menge von Objekten (Elementen)
2. einer Menge von zulässigen Operationen auf diesen Elementen

Die Absicht, die hinter dieser Abstraktion steht ist die, die tatsächliche Implementation vor dem Benutzer zu verbergen. Das hat den Vorteil, daß man die Implementation einer Datenstruktur nach Bedarf ändern kann, ohne daß der Benutzer sein Programm ändern muß. Dies ist gerade bei umfangreichen Programmpaketen sehr nützlich.

Die Idee des ADT findet auch Unterstützung durch MODULA-3:

- sichtbare Schnittstelle: Definitionsmodul
- verborgene Realisierung: Implementierungsmodul

Weiterführende Literatur: [Güting, Seiten 21–23]

Beispiel: Stack als ADT

Algebraische Spezifikation eines Stacks:

- Sorten (Datentypen):
  - Stack (hier zu definieren)
  - Element („ElementTyp“)
- Operationen (sollen die Syntax eines Datentyps festlegen):

  - stackinit: $\rightarrow$ Stack (Konstante)
  - stackempty: Stack $\rightarrow$ Boolean
  - push: Element $\times$ Stack $\rightarrow$ Stack
  - pop: Stack $\rightarrow$ Element $\times$ Stack
- Axiome (sollen die Semantik des Datentyps definieren):

  x: Element („ElementTyp“)
  s: Stack

  pop(push(x,s)) = (x,s)
  push(pop(s)) = s (für nichtleeren Stack s)
  stackempty (stackinit) = TRUE
  stackempty (push(x,s)) = FALSE

Realisierung: [Sedgewick 88, Seite 27]

Anmerkung: Fehlersituationen erfordern Aufmerksamkeit und eine Fehlerroutine, z.B. falls pop(stackinit) aufgerufen wird.

Potentielle Probleme von ADT's

Allgemein:

- Bei komplexen Fällen wird die Anzahl der Axiome sehr groß.
- Das Verständnis der Spezifikation wird u. U. erschwert.
- Es ist schwierig zu prüfen, ob die Gesetze vollständig und widerspruchsfrei sind

Hier:

- Relativ kleine und kompakte Programme
- Höhere Abstraktion: Problemstellung hat Priorität gegenüber der Schnittstelle, durch die sich die Realisierung des ADT verbergen läßt.
- Effizienz spielt für uns eine entscheidende Rolle.

2 Sortieren

2.1 Einführung

In diesem Kapitel wollen wir uns mit verschiedenen Sortierverfahren beschäftigen. Das Sortieren ist ein enorm wichtiges Feld in der Informatik - laut IBM werden etwa 25% der Ressourcen kommerzieller Rechenanlagen allein für diese Aufgabenstellung benötigt. Daher werden an diese Algorithmen sehr hohe Effizienzansprüche gestellt.

Um einen Einstieg in die Problematik zu erhalten widmen wir uns zunächst den einfach zu verstehenden Sortierverfahren.

Elementare Sortierverfahren:

- SelectionSort (Sortieren durch Auswahl),
- InsertionSort (Sortieren durch Einfügen),
- BubbleSort,
- Indirektes Sortieren (Abwandlung bekannter Sortierverfahren mit dem Ziel, die Anzahl der Bewegungen zu minimieren),
- BucketSort.

Sobald anhand dieser Verfahren Begriffe und Problematik geläufig sind, wenden wir uns den komplizierteren und effizienteren Sortierverfahren zu, speziell:

- QuickSort
- HeapSort

Nachdem wir dann diese effizienten Verfahren kennengelernt haben, wollen wir noch untersuchen, wie effizient ein Sortierverfahren maximal sein kann, und ermitteln eine *untere Schranke* für das Sortierproblem.

Definition *Sortierproblem*: Gegeben sei eine Folge $a[1], \ldots, a[n]$ von Datensätzen (**record**). Diese können aus mehreren Informationseinheiten bestehen, z.B. Name, Adresse, PLZ, etc. und einer *key* genannten Schlüsselkomponente.

Das Sortierproblem besteht nun darin, eine Permutation $k = k_1 \ldots k_n$ der ursprünglichen Folge zu finden, so daß gilt

$$a[k_1].\text{key} \leq a[k_2].\text{key} \leq \ldots \leq a[k_{n-1}].\text{key} \leq a[k_n].\text{key}$$

Es handelt sich also um eine Sortierung nach der Schlüsselkomponente.

Voraussetzung ist demnach eine *Ordnung* $\leq$ auf der Menge der Schlüsselkomponenten. Auf Zahlen $\in \mathbb{R}$ oder $\mathbb{N}$ ist dies die gängige „kleiner gleich"-Ordnung und bei Zeichenketten meint $\leq$ die lexikographische („alphabetische") Ordnung.

Da für die Sortierung außer der Schlüsselkomponente der restliche Datensatz uninteressant ist, betrachten wir nachfolgend nur noch die Schlüsselkomponente. Das bedeutet keinen Unterschied für den Algorithmus und erhöht die Übersichtlichkeit.

Statt bei sehr großen Records diese komplett zu bewegen, werden Zeiger auf die Daten sortiert („indirektes Sortieren").

Beachte: Im allgemeinen Fall kann es Duplikate auf der Ebene der Schlüssel oder ganzer Datensätze geben. Bei einer Menge ist dies per definitionem nicht möglich.

Unterscheidungskriterien

Die von uns behandelten Sortierverfahren unterscheiden sich in einigen Punkten, u.a. sind dies

- Effizienz: $O(n^2)$ für elementare Sortierverfahren und $O(n \log n)$ für ausgeklügeltere,
- intern vs. extern (alle Records im Arbeitsspeicher vs. Records ausgelagert im Massenspeicher),
- direkt vs. indirekt (bei letzterem werden zunächst nur Pointer umsortiert),
- in situ vs. nicht in situ (erstere Verfahren kommen ohne zusätzlichen Speicherplatz aus),
- allgemeine vs. spezielle (spezielle Sortierverfahren setzen weitere Eigenschaften der Schlüsselkomponente voraus, z.B. BucketSort),
- Stabilität: Die Reihenfolge von Records mit gleichem Schlüssel sollte erhalten bleiben.

Mit dem Sortieren sind einige Aufgaben verwandt und lassen sich darauf zurückführen, so z.B.:

- Ermittlung des *Median*,
- Ausgabe der k kleinsten Elemente einer Menge.

Außer den im Anschluß behandelten Sortierverfahren gibt es u.a. noch folgende wichtige Verfahren:

- BinaryInsertionSort,
- ShellSort (Verzahnung immer größer werdender sortierter Teilfolgen),
- ShakerSort (interessante Variante von BubbleSort),
- Natural 2-Way-MergeSort (Variante von MergeSort, die die Vorsortierung einer Folge ausnutzen kann).

Ein weiteres wichtiges, und im Worst-Case hervorragend abschneidendes Sortierverfahren ist *MergeSort*, welches wir schon in Abschnitt 1.4.1 (Seite 38) behandelt haben.

2.2 Elementare Sortierverfahren

2.2.1 SelectionSort

Sortieren durch Auswahl.

Gegeben sei eine Folge $a[1], \ldots, a[n]$ natürlicher Zahlen.
Prinzip: Wir betrachten den i-ten Schritt des Verfahrens

1. wähle kleinsten Schlüssel aus der noch nicht sortierten Teilfolge $a[i], ..., a[n]$ aus,
2. vertausche dieses Minimum mit $a[i]$, dem ersten Element der unsortierten Teilfolge.

Nun enthält die Teilfolge $a[1], \ldots, a[i]$ die i kleinsten Schlüssel der Gesamtfolge.

Vorteil: Ausgesprochene Einfachheit des Algorithmus und relativ geringe Zahl von Bewegungen ($\in O(n)$).

Programm [Sedgewick 93]:

```
procedure SelectionSort (var A:array of integer) =
  var i, j, min, t : cardinal;
begin
  for i := FIRST(A) to LAST(A)-1 do
    begin
      min := i;
      for j := i + 1 to LAST(A) do
        if A[j] < A[min] then min := j;
      t := A[min]; A[min] := A[i]; A[i] := t;
    end;
end SelectionSort;
```

Beispiel: Markiert ist das kleinste Element der unsortierten Teilfolge, die jeweils bei der linken Pfeilspitze beginnt.

3	7	8	6	4	2
2	7	8	6	4	3
2	3	8	6	4	7
2	3	4	6	8	7
2	3	4	6	8	7
2	3	4	6	7	8

Bild aus [Sedgewick 88, Seiten 101–102]: Gegeben ist eine Permutation der Zahlen von 1 bis n, so daß am Ende der Sortierung für alle i gilt: $a[i] = i$. Die drei Bilder stellen den Fortschritt der Sortierung nach 25%, 50% und 75% der Laufzeit dar:

Abbildung 8: SelectionSort einer zufälligen Permutation

Komplexitätsanalyse: Zum Sortieren der ganzen Folge $a[1], \ldots, a[n]$ benötigt man $n-1$ Durchläufe. Pro Durchlauf gibt es eine Vertauschung, die sich aus drei Bewegungen zusammensetzt, und $n-i$ Vergleiche ($n-i$ ist die Anzahl der noch nicht sortierten Elemente).
Also gilt:

- $3 \cdot (n-1)$ Bewegungen ($\in O(n)$), und
- $(n-1) + (n-2) + \ldots + 2 + 1 = \dfrac{n(n-1)}{2}$ Vergleiche.

Sind die einzelnen Datensätze sehr groß, eignet sich SelectionSort gut, da die Anzahl der Bewegungen nur linear zur Anzahl der Datensätze wächst. Auch für eine sehr geringe Anzahl von Datensätzen eignet sich SelectionSort, da dort die Einfachheit des Algorithmus konzeptbedingte Zeitverluste wieder kompensiert.

2.2.2 InsertionSort

Sortieren durch Einfügen.

Gegeben sei wieder eine Folge $a[1], \ldots, a[n]$.

Prinzip: Wir betrachten den i-ten Schritt des Verfahrens. Zu diesem Zeitpunkt ist die i-elementige Anfangsfolge schon sortiert.

Füge das Element $a[i+1]$ (erstes Element der unsortierten Teilfolge $a[i+1], \ldots, a[n]$) an korrekter Position in die schon sortierte Teilfolge $a[1], \ldots, a[i]$ ein.

Auch dieser Algorithmus besticht durch seine Einfachheit, benötigt aber wesentlich mehr Bewegungen ($\in O(n^2)$), da bei einer Einfügeoperation alle nachfolgenden Elemente der schon sortierten Folge nach hinten geschoben werden müssen.

Programm [Sedgewick 93]: Sentinel-Element a[0]=$-\infty$.

```
procedure InsertionSort (var A:array of integer) =
  var i, j, v: cardinal;
begin
  for i := FIRST(A)+1 to LAST(A) do
    begin
      v := A[i]; j := i;
      while A[j-1]>v do    (* wegen Sentinel-Element *)
        begin
          A[j] := A[j-1];
          dec(j);
        end;
    end;
end InsertionSort;
```

Anmerkung: In dieser Implementation wird ein sog. *Sentinel-Element* (Wärter, Anfangsmarkierung) verwendet, um an der oben markierten Stelle eine zusätzliche Abfrage zu ersparen, die Rechenzeit kostet und u.U. zu Bereichsverletzungen führen kann.

Dafür verwendet man ein Feld mit Indizes 0 bis n und trägt für den Schlüssel $a[0]$ den kleinstmöglichen Schlüsselwert ein, in unserem Fall (Folge $\in \mathbb{N}$) also 0.

Ohne dieses Element müßte die kommentierte Zeile wie folgt lauten:

```
while j>FIRST(A) and A[j-1]>v do
```

Dieses könnte aber zu Bereichsverletzungen führen, wenn der Compiler **nicht** *faul auswertet.*

Bei „lazy evaluation“ wird bei einem **and** -Ausdruck der zweite Ausdruck, nach dem **and** gar nicht mehr ausgewertet, falls der erste schon **false** war. Wird dieser Ausdruck aber ausgewertet, dann führt obige Zeile für $j = 0$ zu einer Bereichsverletzung.

Daher wird ein solches Sentinel-Element in unterschiedlichen Varianten in vielen Algorithmen verwendet, um eine Abfrage der Array-Grenzen zu sparen.

Beispiel:

$-\infty \quad 3 \quad 7 \quad 8 \quad 6 \quad 4 \quad 2$

$-\infty \quad 3 \quad 7 \quad 8 \quad 6 \quad 4 \quad 2$

$-\infty \quad 3 \quad 7 \rightarrow 8 \rightarrow 6 \quad 4 \quad 2$

$-\infty \quad 3 \quad 6 \rightarrow 7 \rightarrow 8 \rightarrow 4 \quad 2$

$-\infty \quad 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 2$

$-\infty \quad 2 \quad 3 \quad 4 \quad 6 \quad 7 \quad 8$

Sentinel-Element $a[0] = -\infty$

Bild aus [Sedgewick 88, Seiten 101–102]: Gegeben ist wieder eine Permutation der Zahlen von 1 bis n, so daß am Ende der Sortierung für alle i gilt: $a[i] = i$. Die drei Bilder stellen den Fortschritt der Sortierung nach 25%, 50% und 75% der Laufzeit dar:

Abbildung 9: InsertionSort einer zufälligen Permutation

InsertionSort durchläuft die Folge von links nach rechts. Die Teilfolge am Anfang des Feldes ist zwar in sich sortiert, jedoch liegen die Elemente noch nicht an ihrer endgültigen Position.

Komplexitätsanalyse:
Vergleiche: Zum Einfügen des i-ten Elements wird mindestens ein Vergleich benötigt und höchstens i.

Im Mittel sind das $i/2$, da bei zufälliger Verteilung die Hälfte der bereits eingefügten Elemente größer ist als das i-te.

Im *Best Case*, also bei kompletter Vorsortierung, $n-1$ Vergleiche.

Im *Worst Case*, das ist der Fall bei komplett invertierter Vorsortierung, resultiert für die Vergleiche

$$\begin{aligned} \sum_{i=2}^{n} i &= \sum_{i=1}^{n} i - 1 = \frac{n(n+1)}{2} - 1 \\ &= \frac{n^2}{2} + \frac{n}{2} - 1 \\ &\in O(n^2) \end{aligned}$$

Im *Average Case* ist die Anzahl der Vergleiche etwa $\frac{n^2}{4}$, also ebenfalls $\in O(n^2)$.

Bewegungen: Bei der Analyse der Bewegungen kommt es stark auf die Implementierung an. In unserer Implementierung wird im i-ten Schritt das einzufügende Element ($a[i+1]$) zunächst in einen Puffer geschrieben (eine Bewegung), anschließend wird es mit maximal $i+1$ ($+1$ wegen Sentinel-Element), minimal 1 und im Mittel mit $i/2$ Elementen verglichen. Bis auf das letzte Vergleichselement müssen alle um eine Position nach hinten verschoben werden (je eine Bewegung) und im Anschluß wird das betrachtete Element an der gefundenen Stelle wieder eingetragen (eine Bewegung). Im i-ten Schritt werden also bestenfalls zwei Bewegungen vorgenommen, schlechtestenfalls $i+2$ und im Mittel $i/2+2$.

Für den kompletten Durchlauf resultieren demnach für die Zahl der Bewegungen M

im *Best Case*, also bei kompletter Vorsortierung: $M_{BC} = 2(n-1)$.

Im *Worst Case* ist die Zahl der Bewegungen $M_{WC} \approx \frac{n^2}{2}$ also $O(n^2)$, und

im *Average Case* benötigt man nur die Hälfte dieser Bewegungen, also $M_{AC} \approx \frac{n^2}{4}$, aber daher immer noch $O(n^2)$.

Bei gut vorsortierten Folgen zeigt InsertionSort eine gute Effizienz. So würde sich InsertionSort gut eignen einige neue Adressen in ein Telefonbuch aufzunehemen, oder eine wichtige Anwendung ist der Übergang zu InsertionSort ab einem gewissen Vorsortierungsgrad, der durch kompliziertere Verfahren erreicht wurde.

2.2.3 BubbleSort

Sortieren durch wiederholte Vertauschung von Nachbarn.
Gegeben sei wieder eine Folge $a[1], \ldots, a[n]$ natürlicher Zahlen.

Prinzip: Wir betrachten den i-ten Schritt des Verfahrens.

1. Prüfe, ob das erste Element und das nachfolgende in korrekter Ordnung stehen, falls nicht vertausche die beiden.
2. Fahre genauso für das zweite und alle weiteren Elemente fort.

Wie wir am Beispiel sehen werden, wandern dabei die großen Elemente ans Ende der Liste wie Luftblasen an die Wasseroberfläche, daher auch der hübsche Name BubbleSort, dem dieses Verfahren wohl seine Popularität verdankt, denn wie wir in der Analyse sehen werden ist BubbleSort gnadenlos ineffizient.
Programm [Sedgewick 93]:

```
procedure BubbleSort (var A:array of integer) =
  var i, j, t: cardinal;
begin
  for i := LAST(A) to FIRST(A)+1 do
    begin
      for j := FIRST(A)+1 to i do
        begin
          if A[j-1] > A[j] then
            begin
              t := A[j-1]; A[j-1] := A[j]; A[j] := t;
            end;
        end;
    end;
end BubbleSort;
```

Beispiel:

3	7	8	6	4	2
3	7	6	4	2	8
3	6	4	2	7	8
3	4	2	6	7	8
3	2	4	6	7	8
2	3	4	6	7	8

Bild aus [Sedgewick 88, Seiten 101–102]: Gegeben ist wieder eine Permutation der Zahlen von 1 bis n, so daß am Ende der Sortierung für alle i gilt: $a[i] = i$. Die drei Bilder stellen den Fortschritt der Sortierung nach 25%, 50% und 75% der Laufzeit dar:

Abbildung 10: BubbleSort einer zufälligen Permutation

BubbleSort verhält sich ähnlich SelectionSort und wählt das größte Element der Folge um es an die richtige Position zu setzen, so daß der letzte Teil der Folge die größten Schlüssel in sortierter Reihenfolge enthält.

Dabei wirkt es sehr elegant, daß BubbleSort gleichzeitig auf der noch unsortierten Teilfolge die Ordnung erhöht. Dies kann dieses Verfahren aber im nachhinein nicht nutzen, und so stellt sich in der Analyse heraus, daß ein solches Verhalten Zeitverschwendung ist.

Komplexitätsanalyse: Augenscheinlich hängt die Anzahl der Vergleiche nicht von der Vorsortierung der Folge ab, daher sind Worst Case, Average Case und Best Case identisch, es werden immer alle Elemente der noch nicht sortierten Teilfolge miteinander verglichen. Im i-ten Schritt enthält diese $n - i + 1$ Elemente, also benötigt man $n - i$ Vergleiche.

Um die ganze Folge zu sortieren benötigt man $n - 1$ Schritte, daraus folgt für die

Gesamtzahl der Vergleiche:

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

Man sieht also, daß BubbleSort bei der Anzahl der Vergleiche (immer $\in O(n^2)$) sehr schlecht abschneidet.

Bei der Analyse der Bewegungen resultieren für den kompletten Durchlauf im *Best Case*, also bei kompletter Vorsortierung, keine Bewegungen $M_{BC} = 0$
Im *Worst Case* ist die Zahl der Bewegungen $M_{WC} \approx \frac{3n^2}{2}$, und somit $\in O(n^2)$, und im *Average Case* ist die Anzahl der Bewegungen $M_{AC} \approx \frac{3n^2}{4}$, also auch $\in O(n^2)$

Vergleich der elementaren Sortierverfahren

	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	$3n^2/4$	$3n^2/2$

Tabelle: Anzahl der Bewegungen der elementaren Sortierverfahren.

	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	$n^2/2$	$n^2/2$	$n^2/2$

Tabelle: Anzahl der Vergleiche der elementaren Sortierverfahren.

Abschließend bleibt zu sagen, daß die elementaren Sortierverfahren allenfalls für kleine $n \leq 50$ einzusetzen sind, sofern die Folgen nicht vorsortiert sind. Für größere n sollte man auf alle Fälle QuickSort oder HeapSort verwenden.

2.2.4 Indirektes Sortieren

Handelt es sich bei den Datensätzen um sehr große Records, dann kann das Vertauschen dieser den Rechenaufwand für das Sortieren dominieren. In solchen Fällen ist es sinnvoll, statt der Records selbst nur Verweise auf diese zu sortieren.

Die drei elementaren Verfahren InsertionSort, SelectionSort und BubbleSort können dahingehend modifiziert werden, daß bei einer n-elementigen Folge nicht mehr als n

Vertauschungen der eigentlichen Datensätze nötig sind. Dazu geht man folgendermaßen vor:

1. Man verwendet ein Array $p[1..N]$ zur Verwaltung der Record-Indizes.
   Dieses initialisiert man mit $p[i] = i \ \forall\, i = 1, ..., n$.

2. Für die Vergleiche muß auf den Record $a[p[i]]$ zugegriffen werden.

3. Vertauscht werden dann aber nur die Indizes $p[i]$.

4. Optional werden nach dem Sortieren dann noch die Records selbst umsortiert (geschieht in $O(n)$).

Beispiel: Indirektes Sortieren

Vor dem Sortieren:

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

Nach dem indirekten Sortieren:

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

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

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

Auf die genaue Implementierung wollen wir hier nicht eingehen, es sei auf die in der Einführung erwähnte Literatur hingewiesen.

Wir wollen noch kurz auf die *Permutation des Arrays* mittels p[k] eingehen. Benutzt man ein zweites Array, um dort einfach die Datensätze in neuer Reihenfolge einzutragen, so gestaltet sich die Lösung trivial.

Etwas komplizierter wird es, wenn man das Array effizient *in situ* permutieren will. Dies ist sinnvoll, wenn es sich bei den Datensätzen um sehr große Datensätze handelt, so daß z.B. nicht die gesamte Folge ein zweites mal in den Arbeitspeicher paßt.

Ziel ist es, das Array so zu permutieren, daß am Ende $p[i] = i$ ist, und nicht mehr als ein Record zusätzlich zum Array im Speicher untergebracht werden muß. Falls $p[i] = i$ schon erfüllt ist, muß man nichts tun, ansonsten nimmt man folgenden verketteten Ringtausch vor:

1. Lege den aktuell betrachteten Record in einem Puffer ab $t := a[i]$, um Platz für die folgenden Tauschaktionen zu schaffen.

2. Führe Ringtausch auf diesem Ring (Zyklus, Orbit, Bahn vgl. Vorlesung Lineare Algebra zu Permutationen) aus:
   Beispiel (vgl. großes Beispiel oben für i=2): $t = a[2]$; $a[2] = a[11]$; $a[11] = a[13]$; $a[13] = t$;

$$\begin{tikzcd}
 & t \arrow[dr, bend left] & \\
a[2] \arrow[ur, bend left] & a[11] \arrow[l] & a[13] \arrow[l]
\end{tikzcd}$$

Programm: In situ-Permutation [Sedgewick 88]

```
procedure InSitu;
  var i, j, k, t : integer;
begin
  for i := 1 to N do
    if p[i] <> i then
      begin
        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;
```

Versuchen Sie, anhand dieses Codes das obige kleine Beispiel nachzuvollziehen.

2.2.5 BucketSort

BucketSort ist noch unter etlichen anderen Namen bekannt, u.a. Bin Sorting, Distribution Counting, Sortieren durch Fachverteilen oder Sortieren mittels Histogramm.

Vorraussetzung ist, daß die Schlüssel als Integer-Werte im Bereich $0...m-1$ dargestellt werden können, so daß man sie als Index für ein Array verwenden kann:

$$a[i] \in \{0, ..., m-1\} \;\; \forall i = 1, \ldots, n$$

Prinzip:

1. Erstelle ein Histogramm, d.h. zähle für jeden möglichen Schlüsselwert, wie häufig er vorkommt,

2. und berechne anhand dieser Information die Position für jeden Record.

3. Bewege die Records (mit rückläufigem Index) an ihre errechnete Position.

Wegen des rückläufigen Index hat BucketSort die Eigenschaft *stabil* zu sein.
Die Zeit- und Platzkomplexität $T(n)$ liegt in $O(n+m) = O(\max(n,m))$
Programm: [Sedgewick 88]

```
/* a[i] ∈ {0,..,M-1} */
procedure BucketSort;
  var i, j : integer;
begin
  for j := 0 to M − 1 do count[j] := 0;
  for i := 1 to N do
    count[a[i]] := count[a[i]] + 1;
  for j := 1 to M − 1 do
    count[j] := count[j−1] + count[j];
  for i := N downto 1 do
    begin
      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;
```

Beispiel: ABBACADABBADDA

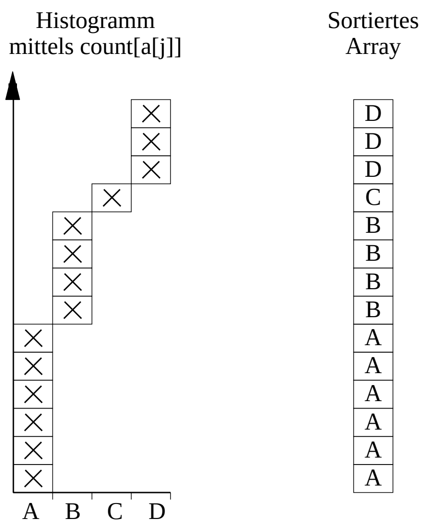


Abbildung 11: Sortiervorgang mit BucketSort

2.3 QuickSort

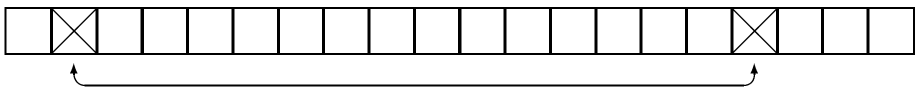
QuickSort, wohl das bekannteste Sortierverfahren, wurde 1962 von C.A.R. Hoare konzipiert.

Das **Prinzip** folgt dem Divide-and-Conquer-Ansatz: Gegeben sei eine Folge F als Array.

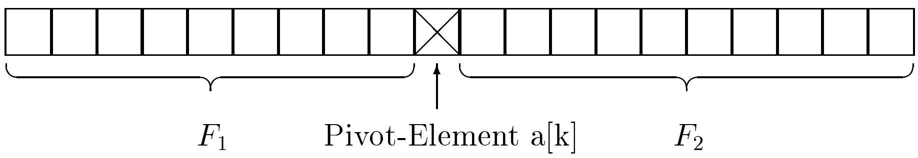
1. Zerlege F anhand eines partitionierenden Elementes (Pivot (engl.= Drehpunkt)) $p \in F$ in zwei Teilfolgen F_1 und F_2 dermaßen, daß für alle Elemente $x_1 \in F_1$ gilt $x_1 \leq p$ und entsprechend für alle $x_2 \in F_2$ gilt $x_2 > p$. Dabei sind die Teilfolgen entsprechend der Größe ihrer Elemente im Array anzuordnen (F_1, p, F_2).
2. Sortiere die Teilfolgen nach demselben Schema bis zum trivialen Fall, daß nämlich diese nur noch aus einem Element bestehen.

Um Rechenzeit zu sparen, verwirklicht QuickSort also folgende Ideen:

- Schlüsseltausch über größere Distanzen im Array:

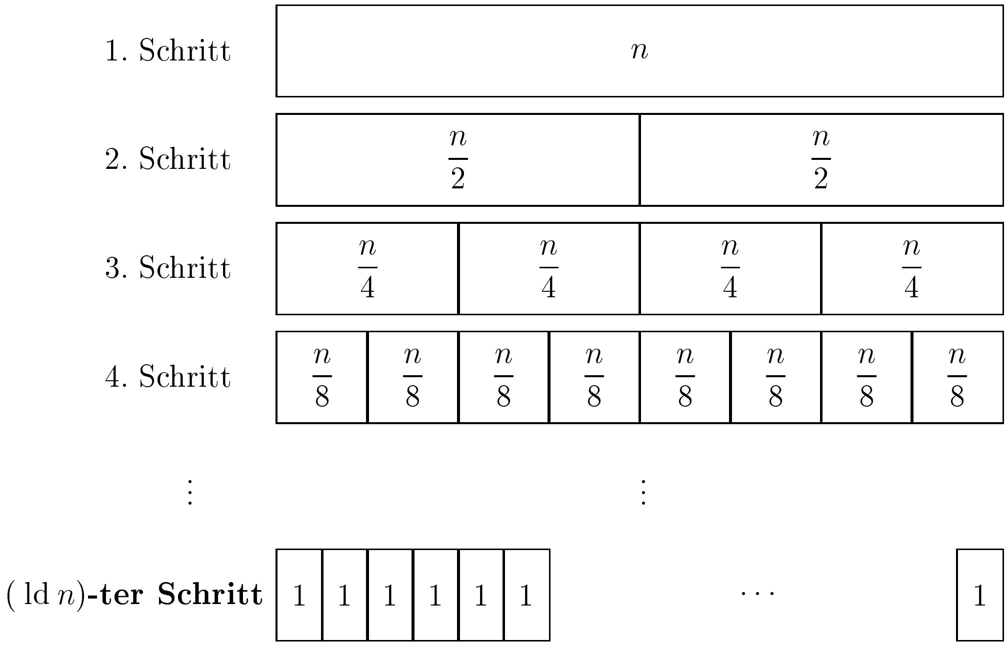


- Zerlegung (Partitionierung) des Arrays in zwei Teilarrays anhand eines Pivot-Elementes a[k] und rekursive Abarbeitung der Teilarrays, für die dann gilt a[i]≤a[k]≤a[j] mit a[i]∈ F_1 und a[j]∈ F_2:



Um zu sehen, warum man mit so einem Vorgehen Rechenzeit gewinnt, wollen wir eine *Komplexitätsabschätzung* für den idealen Fall, daß die Folgen immer genau halbiert werden, durchführen:

Wir betrachten die Zahl der Vergleiche bei einer n-elementigen Folge.



Man benötigt also $\operatorname{ld} n$ Schritte um die Folge in ein-elementige Teilfolgen zu zerlegen, und in jedem Schritt n Vergleiche.

Demnach gilt im idealen Fall (d.h. bei exakter Halbierung in jedem Schritt):

$$T(n) = n \cdot \operatorname{ld} n.$$

Wir wollen nun eine Implementation vorstellen, die das Grundgerüst für QuickSort darstellt. Die Prozedur erhält zwei Parameter l und r, die den linken und rechten Rand des Teilarrays angeben, das bearbeitet werden soll:

```
procedure QuickSort (l, r : integer);
  var i: integer;
begin
  if (r > l) then
    begin
      i := Partition (l, r);
      QuickSort (l, i-1);
      QuickSort (i+1, r);
    end;
end;
```

Der eigentlich wichtige Schritt, der Divide- oder Partitions-Schritt, bleibt hier noch außen vor, er wird in der Funktion `Partition (l, r)` vorgenommen.

Die Funktion **Partition** muß folgende Eigenschaften besitzen:

1. Sie muß dem Pivot-Element a[i] seinen endgültigen Platz im Array zuweisen, und diese Position i zurückgegeben, und

2. alle Elemente des Arrays so umsortieren, daß
   $\forall\, k = l, \ldots, i-1$ gilt $a[k] \le a[i]$ und
   $\forall\, k = i+1, \ldots, r$ gilt $a[k] > a[i]$.

Eine Realisierung, die diese Kriterien erfüllt, und Vertauschungen über große Distanzen durchführt ist folgende, bei der man zwei Zeiger (Indizes i und j) auf dem Array benutzt:

1. Lasse den ersten Zeiger (Index i) von links über das Array wandern, bis er auf ein Element zeigt, für das gilt: $a[i] \ge a[k]$.
2. Lasse dann den zweiten Zeiger (Index j) von rechts über das Array wandern, bis er auf ein Element zeigt, das kleiner als das Pivot-Element ist, also: $a[j] \le a[k]$.
3. Vertausche beide Elemente und fahre entsprechend fort, bis die Zeiger sich kreuzen, also $j \le i$. Abschließend muß noch das Pivot-Element positioniert werden.

```
function Partition (l, r:integer):integer;
  var v,t,i,j : integer;
begin
  i := l-1; j := r;
  v := a[r]; (* wähle Pivot-Element *)
  repeat
    repeat i := i+1 until a[i]≥v;
    repeat j := j-1 until a[j]≤v;
    t := a[i]; a[i] := a[j]; a[j] := t;
  until j≤i; (* Zeiger kreuzen *)
  a[j] := a[i]; a[i] := a[r]; a[r] := a[t];
  return i;
end;
```

Beispiel: Partitions-Funktion. 44 sei das Pivot-Element.

0 55 12 42 94 6 18 67 $\boxed{44}$

0 18 12 42 94 6 55 67 $\boxed{44}$

0 18 12 42 6 94 55 67 $\boxed{44}$

zusätzliche Vertauschung:

0 18 12 42 94 6 55 67 $\boxed{44}$

Rücktausch:

0 18 12 42 6 94 55 67 $\boxed{44}$

0 $\boxed{18 \quad 12 \quad 42 \quad 6}$ $\boxed{44}$ $\boxed{55 \quad 67 \quad 94}$

Sentinel-Element $a[0] = 0$

Allgemeine Anmerkungen zu QuickSort

Es gibt sehr viele unterschiedliche Varianten des QuickSort-Programms. Sie unterscheiden sich in der Wahl des Pivot-Elementes wie in der Implementierung der Partitions-Funktion und anderer Details.

Warnung: Bei allen Varianten ist der *Schleifenkontrolle* und den Indizes besondere Aufmerksamkeit zu schenken.

Betrachtet man in unserer Implementierung der Partitions-Funktion die Schleifenkontrolle für `i:=i+1` und `j:=j-1` genauer, so sollte einem folgendes auffallen:

1. Da `i:=`l`-1` im Falle l=1 auch den Wert 0 annehmen kann, benötigt man ein Element $a[0]$, daß sinnigerweise ein Sentinel-Element $a[0] = -\infty$ sein sollte, mittels dessen man sich dann Indexüberwachungen sparen kann.
2. Es gibt Elemente $a[l-1]$ und $a[r+1]$, für die in dem Vergleich
$$a[l-1] \leq a[k] \leq a[r+1]$$
für alle $k = l, \ldots, r$ Gleichheit möglich ist. Daher muß in den beiden inneren **repeat** ... **until** -Schleifen auf $\geq$ und $\leq$ getestet werden, wenn man die Tests zur Indexüberwachung sparen will.
3. Aus demselben Grund wird in der äußeren **repeat** ... **until** -Schleife auf $j \leq i$ getestet.
4. Insgesamt nimmt man dann aber eine Vertauschung zuviel vor, nachdem sich die Pointer gekreuzt haben. Deswegen muß man nach Beendigung der Schleife noch die letzte Vertauschung rückgängig machen.
5. Wenn alle Schleifen beendet sind muß nur noch das Pivot-Element positioniert werden. `a[i]:=a[r]` .

Fogende Bilder zeigen das Sortierverhalten von QuickSort nach 2, 3, 4, 5, 6, 7 Partitionierungen, wobei eine Partition minimal 12 Elemente enthält.

Varianten und Verbesserungen:

Eine andere Variante erhält man, wenn man die Abfrage **if** $(r > l)$ in geeigneter Modifikation direkt vor den rekursiven Aufrufen von QuickSort durchführt:

```
procedure QuickSort (l, r: integer);
  var i: integer;
begin
  i := Partition (l, r);
  if (l < i-1) then QuickSort (l, i-1);
  if (i+1 < r) then QuickSort (i+1, r);
end;
```

Abbildung 12: QuickSort einer zufälligen Permutation

Allgemein gibt es sehr viele Varianten von QuickSort. Die Variationen bestehen oft in der Auswahl des Pivot-Elementes, die das Verhalten von QuickSort bei unglücklicher Wahl enorm beeinträchtigen kann. Die Chance für eine solch unglückliche Wahl kann man u.a. mit folgenden Methoden verringern:

- v := (a[l] + a[r]) / 2
- Median-of-Three nimmt als Pivot-Element den Mittelwert von drei zufällig gewählten Elementen und bietet dadurch den Vorteil keine Sentinel-Elemente zu benötigen, und eine recht effiziente Aufteilung des Arrays vorzunehmen, was eine Verbesserung der Laufzeit um 5% bewirkt.

Da QuickSort ineffizient bei kleinen Arrays ist, kann es sinnvoll sein, ab einer bestimmten Arraygröße (in der Literatur finden sich Werte M=12 oder 22) ein einfacheres Sortierverfahren zu verwenden.

Durch den Rekursions-Overhead kann es zu großem Speicherplatzaufwand kommen, dies kann man mittels zweier Methoden verhindern:

- Eine Beschränkung des Stacks auf $2 \cdot \operatorname{ld} n$ ist möglich, indem man zunächst immer nur das kürzere der beiden Teilarrays bearbeitet, oder

- eine iterative Variante verwendet.

Zeitkomplexität von QuickSort

Wir wollen nun eine Analyse der oben angegebenen Implementation von QuickSort vornehmen.

Allgemein beruht die Effizienz von QuickSort darauf, daß in der innersten und somit am häufigsten wiederholten Schleife nur ein Vergleich durchgeführt wird. Weitere Vergleiche an dieser Stelle oder gar Bewegungen würden QuickSort enorm verlangsamen. Daher muß man sich bei der Analyse auch auf eine konkrete Implementation konzentrieren.

Sei $T(n)$ die Zahl der Vergleiche, um eine n-elementige Folge mittels QuickSort zu sortieren.

1. **Best Case** - In diesem Fall werden die Arrays in jedem Schritt genau halbiert, und man kommt zu dem o.a. Ergebnis

$$\begin{aligned} T(n) &= (n+1) + 2 \cdot T(\frac{n+1}{2}) \\ &= (n+1)\,\mathrm{ld}\,(n+1) \end{aligned}$$

2. **Worst Case** - Mathematisch läßt sich der Worst Case, also eine möglichst schlechte Aufteilung des Arrays folgendermaßen ausdrücken:

$$T(n) = (n+1) + \max_{1 \le k \le n} \{T(k-1) + T(n-k)\}$$

Der Partitions-Funktion kann man entnehmen, daß sie r-l+2 Vergleiche benötigt um das Teilarray von l bis r weiter aufzuteilen, daher ergibt sich mit $T(0) = T(1) = 0$ und einer induktiven Berechnung von $T(n)$ über n:

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

Die ungünstigste Wahl des Pivot-Elementes wäre jene, in der das Pivot-Element gerade ein Extremum (Minimum oder Maximum) ist, weil dann die Aufteilung nur darin bestünde, das Pivot-Element aus dem Array zu nehmen und dann die restliche Folge zu betrachten. Dies ist genau bei einer komplett vorsortierten Folge der Fall. An diesem Fall kann man dann auch zeigen, daß diese obere Schranke scharf ist.

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

3. **Average Case** - Um die mittlere Zahl von Vergleichen zu erhalten, mitteln wir über alle n möglichen Pivotelemente a[i] mit $i = 1, \ldots, n$. Für $n \ge 2$ gilt dann:

$$T(n) = \frac{1}{n} \sum_{k=1}^{n} (n + 1 + [T(k-1) + T(n-k)])$$

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

Eliminiere die Summe:

$$
\left.\begin{aligned}
n\cdot T(n) &= n(n+1)+2\cdot\textstyle\sum_{k=1}^{n} T(k-1) \\
(n-1)\cdot T(n-1) &= (n-1)n+2\cdot\textstyle\sum_{k=1}^{n-1} T(k-1)
\end{aligned}\right\}\text{subtrahiere}
$$

$$
\begin{aligned}
n\cdot T(n)-(n-1)\cdot T(n-1) &= 2n+2\cdot T(n-1) \\
nT(n) &= 2n+(n+1)T(n-1) \\
\frac{T(n)}{n+1} &= \frac{2}{n+1}+\frac{T(n-1)}{n}
\end{aligned}
$$

Substituiere sukzessive:

$$
\begin{aligned}
\frac{T(n)}{n+1} &= \frac{2}{n+1}+\frac{2}{n}+\frac{T(n-2)}{n-1} \\
&= \frac{2}{n+1}+\frac{2}{n}+\frac{2}{n-1}+\underbrace{\frac{T(n-3)}{n-2}}_{\text{Grenze:n=4}} \\
&\vdots \\
&= \sum_{k=2}^{n}\frac{2}{k+1}+\frac{T(1)}{2} \\
&= 2\cdot\sum_{k=3}^{n+1}\frac{1}{k}+\frac{T(1)}{2}
\end{aligned}
$$

Wir werden zeigen, daß gilt:

$$
\ln\frac{(n+1)}{m}\le\sum_{k=m}^{n}\frac{1}{k}\le\ln\frac{n}{(m-1)}
$$

Beweis siehe folgenden Abschnitt.

Damit erhält man die obere Schranke

$$
\frac{T(n)}{n+1}\le 2\ln\frac{n+1}{2}+\frac{T(1)}{2}
$$

und die untere Schranke

$$
\frac{T(n)}{n+1}\ge 2\ln\frac{n+2}{3}+\frac{T(1)}{2}
$$

Insgesamt folgt also für die Zahl der Vergleiche im Average Case:

$$\begin{aligned} \Rightarrow T(n) &= 2(n+1)\ln(n+1) + \ldots \\ &= 1.386(n+1)\,\mathrm{ld}\,(n+1) + \ldots \end{aligned}$$

Zusammenfassung : Analyse für die konkrete Implementation

$$\begin{aligned} \text{Best Case :} \quad T(n) &= (n+1)\cdot\ \mathrm{ld}\,(n+1) \\ &\in O(n\,\mathrm{ld}\,n) \\ \text{Average Case :} \quad T(n) &= 1,386(n+1)\cdot\ \mathrm{ld}\,(n+1) \\ &\in O(n\,\mathrm{ld}\,n) \\ \text{Worst Case :} \quad T(n) &= \frac{(n+1)(n+2)}{2} - 3 \\ &\in O(n^2) \end{aligned}$$

Beweis der Schranken von $\sum_{k=m}^{n} \frac{1}{k}$ mittels Integral-Methode

Sei $f(x)$ monoton fallend, dann bildet man die Unter- und Obersumme für diese Funktion.

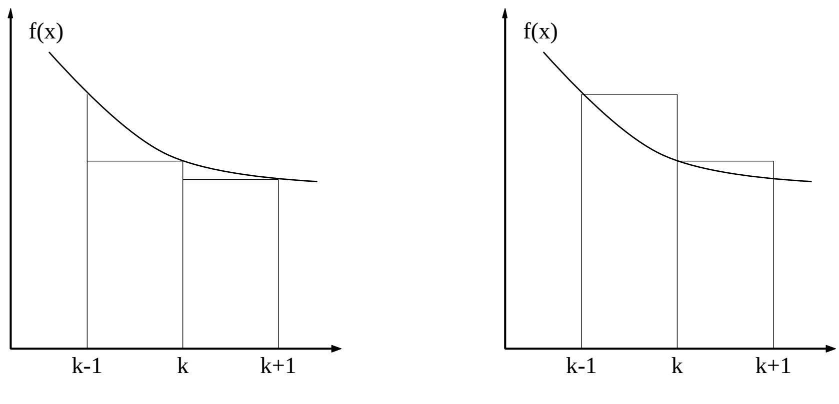


Abbildung 13: Unter- und Obersummen

$$\begin{aligned} \int_k^{k+1} f(x)dx &\leq 1\cdot f(k) \leq \int_{k-1}^{k} f(x)dx \\ \int_m^{n+1} f(x)dx &\leq \sum_{k=m}^{n} f(k) \leq \int_{m-1}^{n} f(x)dx \end{aligned}$$

Speziell gilt dann für $f(k) = \frac{1}{k}$ und $m \geq 2$

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

Für $m = 1$ erhält man die Partialsumme $\sum_{k=1}^{n} \frac{1}{k}$, die in der Mathematik eine wichtige Rolle spielt und daher mit einem eigenen Namen belegt ist.

Definition *Harmonische Zahlen*: Als Harmonische Zahlen bezeichnet man für $n \in \mathbb{N}$ die Reihe

$$H_n := \sum_{k=1}^{n} \frac{1}{k}$$

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

Die Idee zu HeapSort hatten J.W.J. Williams und R.W. Floyd unabhängig voneinander im Jahre 1964. Sie basiert auf der Überlegung den Auswahlschritt in *SelectionSort* zu verbessern. In SelectionSort muß man n mal das größte Element aus einer Menge auswählen. Diese Auswahl benötigt $O(n)$ Schritte, und so resultiert die für elementare Sortierverfahren übliche Komplexität von $O(n^2)$. Bei HeapSort stellt man nun vor jedem Auswahlschritt die unsortierte Teilfolge in Form eines sogenannten *Heaps* (engl. = Halde) dar. Dieser Vorbereitungsschritt benötigt $O(\operatorname{ld} n)$ Operationen. Die Auswahl des Maximums gelingt nun mit $O(1)$ Operationen. Mit den notwendigen n Auswahlschritten ergibt sich die verbesserte Komplexität als $O(n \operatorname{ld} n)$.

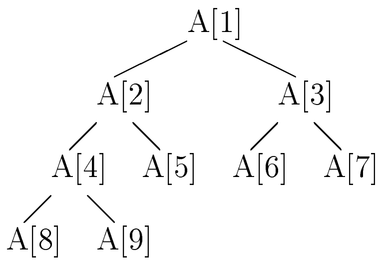
Definition *Heap-Eigenschaft*: Ein Array $A[1..n]$ erfüllt die Heap-Eigenschaft, falls für $2 \leq i \leq n$ gilt

$$A\left[\left\lfloor \frac{i}{2} \right\rfloor\right] \geq A[i]$$

Also ist ein Array $A[1..n]$ ein Heap beginnend in Position l mit $1 \leq l \leq n$, falls gilt

$$A\left[\left\lfloor \frac{i}{2} \right\rfloor\right] \geq A[i] \quad \text{für } l \leq i \leq n.$$

Man stelle sich das Array als Darstellung eines Binärbaums wie folgt vor:



Dann bedeutet die Heap-Eigenschaft nur, daß für jeden Knoten gilt, daß sein Wert größer ist als der seiner Söhne.

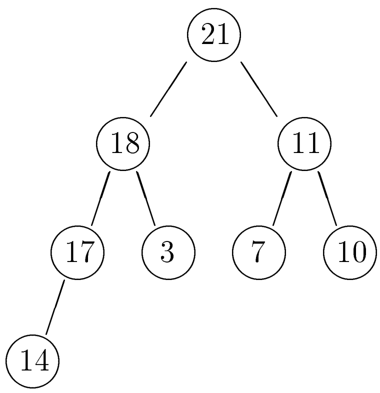
Und daraus folgt augenscheinlich, daß für einen Heap im Array $A[1..n]$ gilt:

$$A[1] = \max\{A[i] \mid i = 1, \ldots, n\}.$$

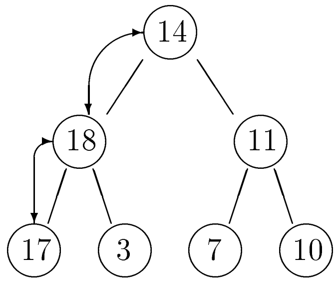
Beachte: Da alle Knoten ab $l = \left\lceil \frac{n}{2} \right\rceil$ keine Söhne mehr haben, ist auch klar, daß jedes Array $A[1..n]$ ein Heap beginnend in l ist, denn es ist ja keine Bedingung mehr zu erfüllen.

Um einen ersten Eindruck von der Arbeitsweise HeapSorts zu bekommen, und zu erkennen, wie man relativ einfach die Heap-Eigenschaft wiederherstellen kann, nachdem das Maximum entfernt wurde, betrachten wir folgendes Beispiel.

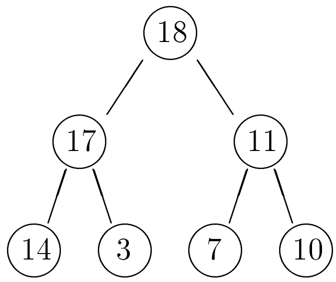
Beispiel: Gegeben sei ein Array, das die Heap-Eigenschaft erfüllt.



Wir entfernen die Wurzel (**21**), setzen das letzte Element des Arrays (**14**) an die erste Position (die Wurzel) und vertauschen es, um die Heap-Eigenschaft wieder herzustellen, solange mit dem größeren der beiden Söhne, solange es kleiner ist als dieser. Dieser Schritt wird auch *versickern* genannt.



Abschließend stellt sich das Array folgendermaßen dar. Man kann leicht überprüfen, daß es die Heap-Bedingung erfüllt.



Da wir weiterhin in situ sortieren wollen, entfernt man die Wurzel nicht und setzt das letzte Element in die Wurzel ein, sondern tauscht im i-ten Schritt die Wurzel mit dem $(n-i+1)$-ten Element.

Algorithmus: HeapSort
Nach diesen Vorbetrachtungen können wir einen Algorithmus für Heapsort formulieren:

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.

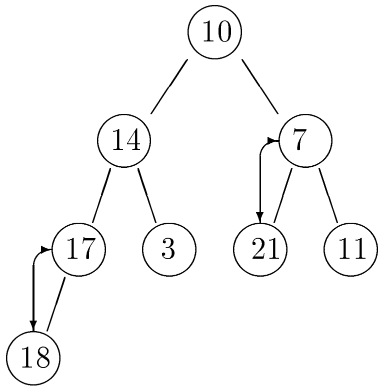
Der erste Schritt, der *Heap-Aufbau* bleibt zu klären. Anhand eines Beispiels (die gleiche Folge wie oben) wollen wir diesen entwickeln.

Beispiel: Heap-Aufbau. Wie wir oben schon gesehen haben, erfüllt jedes Array mit $l = \left\lceil \frac{n}{2} \right\rceil$ beginnend die Heap-Eigenschaft. Man muß also nur noch die vorderen Elemente des Arrays im Heap versickern.

Ausgangssituation:

10	14	7	17	3	21	11	18

$\rightarrow$ Heap



Vertausche 17 $\leftrightarrow$ 18

10	14	7	18	3	21	11	17

$\rightarrow$ Heap

Vertausche 7 $\leftrightarrow$ 21

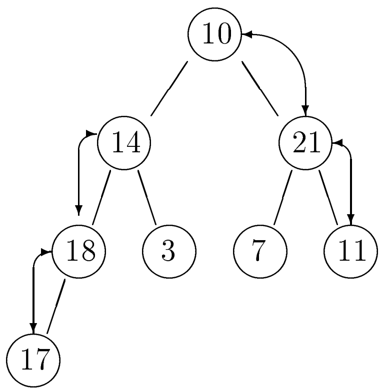
10	14	21	18	3	7	11	17

$\rightarrow$ Heap

Versickere 14 $\leftrightarrow$ 18 $\leftrightarrow$ 17

10	18	21	17	3	7	11	14

$\rightarrow$ Heap



Versickere zuletzt 10 $\leftrightarrow$ 21 $\leftrightarrow$ 11

21	18	11	17	3	7	10	14

Programm: HeapSort [Sedgewick 88]
Gegeben sei ein Array a[1..M]. Die Prozedur DownHeap(k) dient dem Versickern des Knotens k, und bearbeitet dabei das Teil-Array a[k..N] (a[1..N] ist der noch nicht sortierte Teil des Arrays).

```
procedure HeapSort;
  var k,t: integer;
begin
  N := M;
  for k := (M div 2) downto 1 do DownHeap(k);
  repeat
    t := a[1]; a[1] := a[N]; a[N] := t;
    N := N − 1;
```

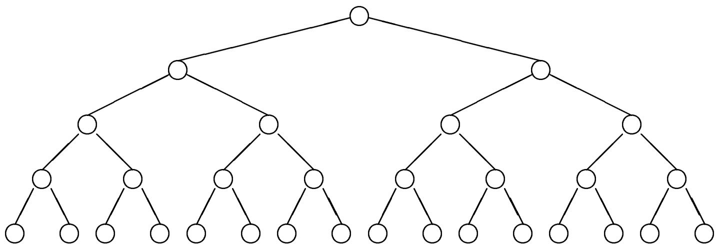
```
    DownHeap(1)
  until N ≤ 1;
end;


procedure DownHeap(k : integer);
  label 0;
  var i, j, v: integer;
begin
  v := a[k];
  while k ≤ (N div 2) do
    begin
      j := k + k; (* Berechne linken Sohn *)
      if j < N then (* Existiert rechter Sohn? *)
        if a[j] < a[j+1] then j := j + 1; (* Wähle größeren Sohn *)
      if v ≥ a[j] then goto 0;
      a[k] := a[j]; k := j;
    end;
  0: a[k] := v;
end;
```

In den folgenden Bildern ist die Sortierung einer zufälligen Permutation der natürlichen Zahlen von 1 bis 50 dargestellt. Zunächst wird der Heap-Aufbau dargestellt, dann die Sortierphase. [Sedgewick 88]

Komplexitätsanalyse von HeapSort

[Mehlhorn, Seite 47]
Wir betrachten die benötigte Anzahl von Vergleichen, um ein Array der Größe $2^k - 1$ zu sortieren. Die Motivation zu HeapSort lag ja darin, auch im Worst Case noch ein effizientes Sortierverhalten ($\in O(n \operatorname{ld} n)$)zu garantieren, daher wollen wir unser Augenmerk auch auf die Worst Case-Analyse richten.

Zum besseren Verständnis wollen wir diese Analyse am konkreten Beispiel eines Arrays für $k = 5$ und somit $n = 2^5 - 1 = 31$ erläutern:



$i = 0$: $2^0 = 1$ 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

Aufbau eines Heaps mit $n = 2^k - 1$ Knoten

- Auf der Ebene i für $0 \leq i \leq k - 1$ gibt es 2^i Knoten.

Abbildung 14: HeapSort einer zufälligen Permutation: Aufbau des Heaps

- Wir betrachten das Hinzufügen von Knoten der Tiefe $i = (k-2), (k-3), \ldots, 1$.
- Ein Knoten, der auf Niveau i hinzugefügt wird, kann maximal auf Niveau $(k-1)$ sinken.

  Pro Niveau werden dazu höchstens zwei Vergleiche benötigt.

  1. Vergleich **if** $v \geq a[j]$ **then** : Wird bei jedem Durchlauf der **while** -Schleife ausgeführt, die ihrerseits bei jedem Prozeduraufruf DownHeap(k) mindestens einmal durchlaufen wird.
  2. Vergleich **if** $a[j] < a[j+1]$ **then** : wird ausgeführt, falls 2. Sohn existiert.

  Deswegen ist die obere Schranke der Anzahl der Vergleiche die Summe dieser beiden, und ergibt sich zu

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

DownHeap

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 und kann mit

Abbildung 15: HeapSort einer zufälligen Permutation: Sortierphase

DownHeap um maximal i Niveaus versickern. Je Niveau werden dazu wiederum jeweils maximal zwei Vergleiche benötigt. So ergibt sich für die Summe der Vergleiche

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

Die beiden Summenformeln (1) und (2) lassen sich mit vollständiger Induktion über k beweisen.

Für die Anzahl $T(n)$ der Vergleiche gilt dann folgende *obere Schranke*:

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

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

Ergebnis: HeapSort sortiert ein n-elementiges Feld *garantiert* mit weniger als

$$2n \operatorname{ld}(n+1) - 2n \quad \text{Vergleichen.}$$

Wegener [Güting, Seite 196] hat als **Verbesserung** zu HeapSort ein Bottom-Up-HeapSort konzipiert, welches die Zahl der Vergleiche in die Nähe von $1 \cdot n \text{ ld } (n+1)$ rückt, und so dieses Verfahren konkurrenzfähig zu QuickSort macht.

2.5 Untere und obere Schranken

Nun haben wir einige effiziente Verfahren zum Sortieren von Folgen kennengelernt, und es bleibt die Frage offen: Wie effizient kann ein optimales Sortierverfahren überhaupt sein?
Als Modell betrachten wir einen vergleichsorientierten Algorithmus, d.h der Algorithmus kennt nur die Operation „Vergleich zweier Schlüssel“

Satz: Jeder vergleichsorientierte Algorithmus benötigt $(n-1)$ Vergleiche, um das Extremum einer n-elementigen Menge zu bestimmen.

Beweis (durch Widerspruch): Sei das Extremum mit weniger als $(n-1)$ Vergleichen eindeutig bestimmt worden.
Dann gibt es ein Element, mit dem das Extremum nicht verglichen worden ist, und über dessen Position zu dem Extremum nichts ausgesagt werden kann.
Dieses Element könnte kleiner als das Minimum bzw. größer als das Maximum sein $\Longrightarrow$ Widerspruch zur eindeutigen Bestimmung des Extremums.

Um uns der Lösung für die minimale Zahl der Vergleiche zu nähern, betrachten wir das Sortierproblem ganz elementar als Auswahl genau einer der $n!$ möglichen Permutationen des Arrays $A[1..n]$.
Ein solches Auswahlproblem läßt sich als *binärer Entscheidungsbaum* darstellen. Dazu wird jede mögliche Wahl als ein Blatt des Baumes interpretiert, und jeder innere Knoten als Position, an der ein (binärer) Vergleich durchgeführt wird.
Die Zahl der Vergleiche $T(n)$ um eine Folge zu sortieren, ist dann die Länge eines Pfades von der Wurzel zu dem Blatt, daß diese Sortierung repräsentiert.

- **Untere Schranke:** Die kürzesten Wege in einem Baum mit vorgegebener Knotenzahl gibt es in einem (Binär-)Baum mit minimaler Höhe (Abschnitt 1.3.4), so daß gilt $T(n) \geq$ minimale Höhe eines Binärbaumes mit $n!$ Blättern, also:
$$T(n) \geq \lceil \text{ ld } n! \rceil.$$
- **Obere Schranke:** Wir wählen einen „effizienten“ Algorithmus, nämlich Merge-Sort. Denn mit $T(n) = n\lceil \text{ ld } n\rceil - 2^{\lceil \text{ ld } n\rceil} + 1$ besitzt er die beste obere Schranke für die Anzahl der Vergleiche (Abschnitt 1.4.1).

Um sinnvolle Schranken für $T(n)$ angeben zu können, benötigen wir enge Schranken für $\text{ld } n!$, denn eine direkte Berechnung der Fakultät ist zu langwierig bei großen n.

Wir benutzen folgende enge untere Schranke für $\operatorname{ld} n!$:

$$n \operatorname{ld} n - n \operatorname{ld} e \leq \operatorname{ld} n!$$

Beweis: siehe Abschnitt 2.6

Setzt man nun die untere Schranke für $n!$ noch in die erste Ungleichung ein, so erkennt man, daß untere und obere Schranke für die optimale Lösung des Sortierproblems dicht beieinander liegen:

$$n \operatorname{ld} n - n \cdot \operatorname{ld} e \leq T(n) \leq n\lceil \operatorname{ld} n \rceil - n + 1$$

Somit haben wir gezeigt, daß das Sortierproblem nicht besser als in $O(n \operatorname{ld} n)$ lösbar ist, und also QuickSort und HeapSort in der optimalen Komplexitätsklasse liegen und nicht mehr „um Klassen verbessert" werden können.

2.6 Schranken für n!

Die Fakultät $n!$ wird oft benötigt; die direkte Berechnung ist aber gleichzeitig umständlich und analytisch schwierig zu behandeln. Aus diesem Grund benötigt man enge Schranken für $n!$.

(a) *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 also:

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

(b) *Engere Schranken*: Wir wollen engere Schranken für $n!$ mittels der *Integral-Methode* berechnen, und betrachten dazu die Abbildung $n \mapsto \ln n!$, denn es gilt

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

so daß sich über das Integral der reellen Funktion $x \mapsto f(x) = \ln x$ und die Summation folgendermaßen enge Schranken für $n!$ berechnen lassen.

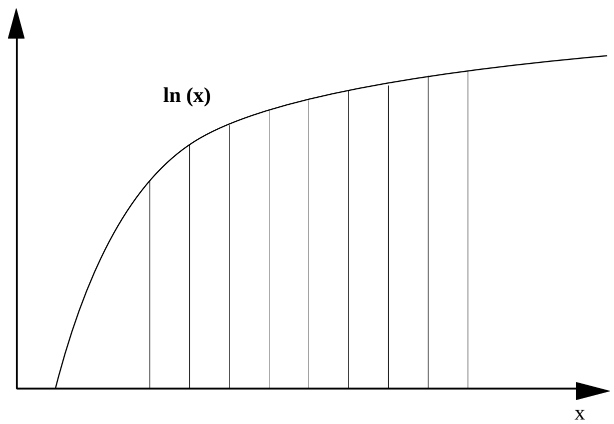


Abbildung 16: Verlauf von $\ln n$ und Partialsummenzerlegung

Wir approximieren das Flächenintegral von oben und unten durch *Trapez-Summen.*

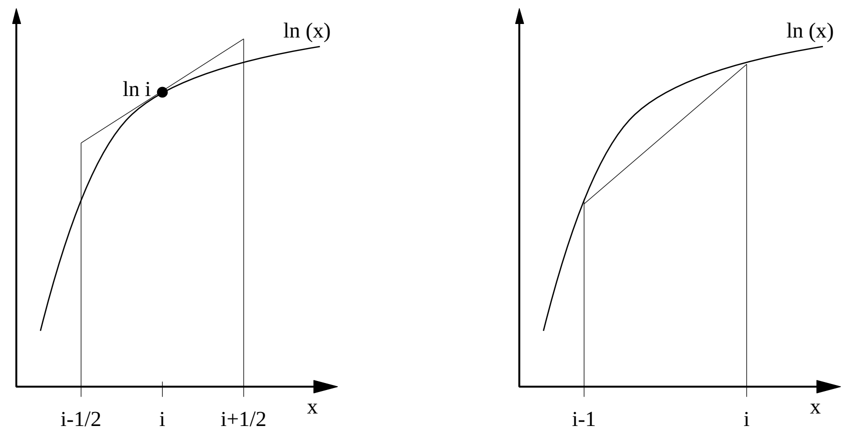


Abbildung 17: Trapezober- und Trapezuntersummen

Untere Schranke:
Der Trapezobersumme kann man entnehmen, daß die Fläche des Trapezes größer ist, als die des Integrals, also bildet das Integral eine untere Schranke für die Trapezfläche.

$$\int_{i-1/2}^{i+1/2} f(x)dx \leq \ln i$$

Um von $\ln i$ auf $\ln n!$ zu kommen müssen wir von 1 bis n über i summieren:

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

Wegen

$$\int_a^b \ln x dx = x \ln x - x|_a^b = b \ln b - b - a \ln a + a$$

folgt die enge untere Schranke von $\ln n!$

$$n \ln \frac{n}{e} + \frac{1}{2} \ln n + \frac{1}{2} \ln 2 \leq \ln n!$$

Obere Schranke:
Dazu betrachten wir die Trapez-Untersummen, denn das Integral bildet eine obere Schranke der Trapezuntersummen.

$$\frac{1}{2}[f(i-1) + f(i)] \leq \int_{i-1}^{i} f(x)dx$$

Summation über alle i von 2 bis n:

$$\ln n! - \frac{1}{2} \ln n \leq \int_1^n \ln x dx = n \ln \frac{n}{e} + 1$$

$$\Longrightarrow \ln n! \leq n \ln \frac{n}{e} + \frac{1}{2} \ln n + 1$$

Also gelten für $\ln n!$ folgende enge Schranken

$$n \ln \frac{n}{e} + \frac{1}{2} \ln n + \frac{1}{2} \ln \frac{1}{2} \leq \ln n! \leq n \ln \frac{n}{e} + \frac{1}{2} \ln n + 1$$

Für $n!$ erhält man entsprechend folgende Schranken:

$$\sqrt{2} \leq \frac{n!}{(n/e)^n \cdot \sqrt{n}} \leq e$$

(c) *Engste Schranken und Stirlingsche Formel* (ohne Beweis)
Setzt man

$$n! = \sqrt{2\pi n}(n/e)^n \cdot \alpha_n$$

dann gilt:

i. für $2 \leq n$:

$$e^{1/(12n+1/4)} \leq \alpha_n \leq e^{1/12n}$$

ii. für $n \to \infty$:

$$\lim_{n\to\infty} \alpha_n = 1$$

Zusammenfassung

Zusammenfassend wollen wir noch einmal die gesamte Rechenzeit $T(n)$ der Sortierverfahren im Average Case betrachten. Dazu wurden alle Verfahren umgesetzt in RAM-Programme und die Kosten $T(n)$ im Einheitskostenmaß ermittelt: [Mehlhorn]

SelectionSort	(2.2.1)	$= 2.5n^2 + 3(n+1)\,\mathrm{ld}\,n + 4.5n - 4$
QuickSort	(2.3)	$= 9(n+1)\,\mathrm{ld}\,(n+1) + 29n - 33$
HeapSort	(2.4)	$= 20n\,\mathrm{ld}\,n - n - 7$
MergeSort	(1.4.1)	$= 12n\,\mathrm{ld}\,n + 40n + 97\,\mathrm{ld}\,n + 29$

Man sieht, daß MergeSort sehr gut abschneidet, besonders wenn man sich erinnert, daß für MergeSort die Kosten im Worst Case gleich denen im Average Case waren. Nachteilig ist bei MergeSort jedoch, daß es in der einfachen Version nicht in situ arbeitet, und sich eine in situ-Version relativ kompliziert gestaltet.

3 Suchen in Mengen

3.1 Einführung

Gegeben sei eine Menge von Elementen (Records), von denen jedes aus einer Schlüsselkomponente und beliebigen weiteren besteht.

In der Regel werden Duplikate ausgeschlossen, wobei der Begriff Duplikat sich sowohl nur auf den Schlüssel beziehen kann, als auch auf den vollen Record.

Im letzteren Fall wäre der Begriff *Menge* jedoch nicht mehr streng interpretiert.

Die Darstellung von Daten als Menge ist sehr weit verbreitet (Datenbanken sind fast immer Mengen) und die darauf basierende Verarbeitung umfaßt typischerweise Aufgaben wie *Einfügen, Entfernen, Lesen* etc., die alle das *Suchen* in Mengen beinhalten. Diese Operationen faßt man unter dem Begriff *Dictionary-Operationen* (*dictionary* engl.=Wörterbuch) zusammen, und benutzt im allgemeinen folgende Schreibweisen.

Notation [Mehlhorn]:

- **Universum** U**:** Menge aller möglichen Schlüssel, aus denen eine Teilmenge $S \subseteq U$ aller tatsächlich vorhandenen Schlüssel betrachtet wird.
- **Search(**x, S**):** Liefert den zu x gehörigen Record (oder die Adresse an der x im Speicher steht), falls $x \in S$, und sonst eine Meldung: „$x \notin S$“.
- **Insert(**x, S**):** Fügt das Element x zur Menge S hinzu:
  $S := S \cup \{x\}$
  Falls x bereits Element der Menge S ist, wird eine Fehlermeldung ausgegeben.
- **Delete(**x, S**):** Entfernt das Element x aus der Menge S:
  $S := S \backslash \{x\}$
  Falls $x \notin S$ gibt es eine Fehlermeldung.

Weitere Operationen:

- **Order(**k, S**):** Liefert das k-te Element der geordneten Menge S zurück.
- **ListOrder(**S**)** Erstellt eine *geordnete Liste* der Elemente der Menge S.
- **Enumerate(**S**):** Zählt alle Elemente der Menge S auf.
- **FindMin(**S**):** Entspricht Order$(1, S)$.
- **Initialize(**S**):** Initialisiert die Darstellung als (leere) Menge, d.h. $S := \emptyset$
- **Union(**S, A**):** $S := S \cup A$
- **Intersection(**S, A**):** $S := S \cap A$

- **Difference(S, A):** $S := S \backslash A$
- **Find(x, A_k):** Liefert die Teilmenge $A_k \subset U$ zurück, zu der das Element x gehört.

Hinweis: Diese Terminologie ist nicht standardisiert, und so finden sich zahlreiche Synonyme für obige Begriffe, z.B. wird oft *Member, Contains* oder *Access* für *Search* verwendet.

Je nachdem was für Schlüssel man verwendet, hat das Universum eine stark verschiedene Größe. Die Anzahl der tatsächlich verwendeten Schlüssel $|S|$ hängt sehr von der Anwendung ab. Hier einige Beispiele [Mehlhorn, Seite 97]:

- Die Symboltabelle eines Compilers: 6 Character (Buchstaben) und eine Ziffer: $|U| = (26 + 10)^6 = 2.2 \cdot 10^9$ und $|S| < 10^3$.
- Ein Autorenverzeichnis einer Bibliothek in lexikographischer Ordnung hat eine ähnliche Größenordnung.
- Die Konten einer Bank: 6-stellige Konto-Nummern, von denen etwa 50% tatsächlich genutzt werden: $|U| = 10^6$ $|S| = 5 \cdot 10^5$.

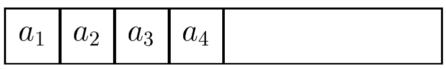
  Hier sind die Größe des Universums und die Menge der benutzten Schlüssel gleich groß.

3.2 Einfache Implementierungen

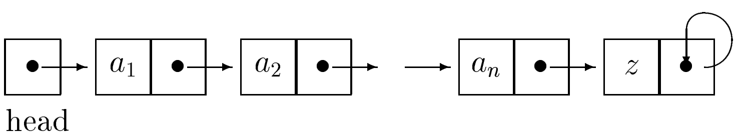
3.2.1 Ungeordnete Arrays und Listen

Darstellung: vgl. Abschnitt 1.3.2

- Liste als Array



- Verkettete Liste



3.2.2 Vergleichsbasierte Methoden

Um vergleichsbasierte Methoden anzuwenden, muß eine Ordnung auf der Menge angenommen werden, dies ist aber keine wirkliche Einschränkung, denn auf der internen Darstellung der Elemente von U läßt sich immer eine Ordnung definieren [Mehlhorn, Seite 139].

Wir wollen speziell Suchverfahren für Daten im *geordneten Array S* betrachten. D.h. es gilt $S[i] < S[i+1]$ für $0 < i \leq n-1$.
Dann können wir grob drei Suchverfahren unterscheiden:

1. Sequentielle oder lineare Suche. Sie ist auch ohne Ordnung möglich.
2. Binärsuche,
3. Interpolationssuche.

Diese drei Verfahren unterscheiden sich nur in der Wahl, welches Element als nächstes untersucht wird. Daher kann man ein **Allgemeines Programmschema für Suchalgorithmen** formulieren:

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

procedure Search(a,S);
  var low, high: integer;
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 "a wurde an Position " next " gefunden.";
  else "a wurde nicht gefunden!";
  return
end;
```

Varianten der Anweisung: next := *an integer* $\in$ *[low..high]*

- **Lineare Suche**(sequentiell): $next := low$
- **Binärsuche**(sukzessives Halbieren):

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

- **(Lineare) Interpolationssuche:** Sinnvoll unter der Annahme, daß die Schlüssel etwa linear wachsen.

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

Komplexität: Wir betrachten die Zahl der Schlüsselvergleiche $T(n)$, die durchschnittlich bei erfolgreicher, und bei erfolgloser Suche nötig sind, um bei einer n-elementigen Menge zu einem Ergebnis zu kommen. Bei einer geordneten Liste schneiden, wegen des vorzeitigen Suchabbruchs erfolgreiche und erfolglose Suche gleich ab.

1. **Lineare Suche**: $C(n) \in O(n)$.
   - Erfolgreiche Suche: $n/2$
   - Erfolglose Suche: $n/2$ bzw.
     n bei einer ungeordneten Liste.
2. **Binärsuche**: $C(n) \in O(\,\mathrm{ld}\, n)$ Rekursionsgleichung:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + 1 \\ \Rightarrow T(n) &< \mathrm{ld}\, n + 1 \end{aligned}$$

   Es werden also garantiert weniger als $\mathrm{ld}\, n + 1$ Vergleiche benötigt.
3. **Interpolationssuche** (ohne Beweis und Erläuterung):
   - average-case: $C(n) \in O(\,\mathrm{ld}\,(\,\mathrm{ld}\, n))$
   - worst-case: $C(n) \in O(n)$ (Entartung in lineare Suche)

Korrektheit des Programms: Um die Korrektheit des Programms zu beweisen, läßt sich die Bedingung der **while** -Schleife als folgendes logisches Prädikat P formulieren:

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

Damit wird der Beweis folgendermaßen geführt:

- Vor der Schleife ist P offensichtlich erfüllt.
- In der Schleife gilt $a \neq S[next]$ und also
  1. Entweder $a < S[next]$:
     Woraus folgt, daß wegen der Ordnung auf S $a \notin S[next..high]$ und somit gilt, falls $a \in S \Rightarrow a \in S[low..next - 1]$.
  2. Oder $a > S[next]$:
     Analog zu 1. gilt dann, falls $a \in S \Rightarrow a \in S[next + 1..high]$.

  Falls $low \leq high$ gilt, folgt wegen der ± 1 Änderung:
  $low \leq next \leq high$

- Die Schleife terminiert, da in jedem Durchlauf $high - low$ mindestens um 1 verringert wird.
- Wenn die Schleife terminiert gilt also P und
  1. entweder $a = S[next]$: D.h. die Suche endet erfolgreich,
  2. oder $low \geq high$. Angenommen $a \neq S[next]$. Die Suche endet erfolglos, da, aus P folgt daß $a \in S[low..high]$ mit $low \geq high$:
     - Falls $high < low$ ist $S[low..high] = \{\}$ und somit $a \notin S[1..n]$
     - Falls $high = low$: dann ist wegen P $next = high$ und wegen $a \neq S[next]$) auch $a \notin S[1..n]$.

3.2.3 Bitvektordarstellung (Kleines Universum)

Ideal für die drei Operationen Search, Insert und Delete wäre eine Darstellung, bei der diese eine Zeitkomplexität $T(n) \in O(1)$ haben und die gesamte Darstellung eine Platzkomplexität $\in O(n)$.
Die Zeitkomplexitäts-Bedingung wird erfüllt von der *Bitvektordarstellung*. Der einzige Nachteil ist, daß die maximale Anzahl der Elemente $N = |U|$ vorgegeben sein muß, und die Darstellung auch eine Platzkomplexität $\in O(N)$ hat, und sich daher nur für kleine Universen eignet.

Die Schlüssel seien aus $S \subseteq U = \{0, 1, ..., N-1\}$.

Dann faßt man die Schlüssel als Index im Bitvektor (Array of Bit[0..N-1]) auf, und es gilt:

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

Die Platzkomplexität ist $\in O(N)$, die Zeitkomplexität für *Search, Delete* und *Insert* ist $\in O(1)$ und die für *Initialize* $\in O(N)$.

3.2.4 Kleines Universum

Um auch die Zeitkomplexität von *Initialize* $\in O(1)$ zu erhalten, kann man eine erweiterte Bitvektordarstellung wählen, bei der man zwei Hilfsarrays verwendet, und so die aufwendige Initialisierung entfällt. [Mehlhorn, Seite 270]
Andere Namen für diese erweiterte Bitvektordarstellung sind:

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

Man benutzt also folgende drei Arrays:

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

Der Bitvektor ist derselbe wie zuvor, da er aber nicht initialisiert wird, muß überprüft werden, ob der Wert an der Stelle i Gültigkeit besitzt. Dazu dienen die beiden Hilfsfelder *Pointer-Array* und *Stack-Array*.

Das Pointer-Array ist parallel zum Bitvektor angelegt und enthält einen Pointer auf ein Element des Stack-Arrays (also einen Integer-Index). Falls dieses Element zum Stack gehört (sein Index also kleiner ist als der Indexzähler top) und einen korrekten Back-Pointer auf das Pointer-Array enthält, dann ist das Element i in der Menge enthalten.

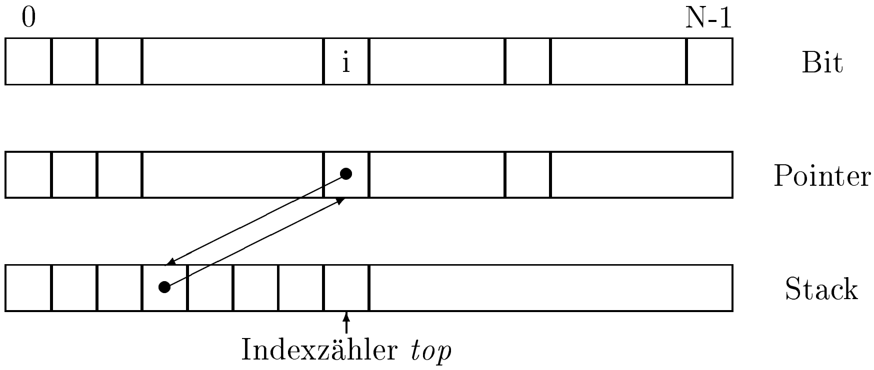
Formal beruht das Konzept also auf folgenden drei Invarianten:

Es gilt $i \in S$ genau dann, wenn

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

wobei die leere Menge mit $top = -1$ initialisiert wird.

Durch den Zähler top und den Back-Pointer in $Stk[.]$ entfällt also die Initialisierung.



Implementation 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“ s.Seite 55).

Insert(i, S) und Delete(i, S) sind fast identisch, der einzige Unterschied besteht darin, daß für *Delete* das angesprochene Bit auf 0, und für *Insert* auf 1 gesetzt wird, und die **if** -Bedingung für *Delete* immer erfüllt ist, und daher der **else** -Zweig nur für *Insert* durchlaufen wird.

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

Anmerkungen:

- Die Variable *top* gibt die maximale Anzahl der jemals mit *Insert* oder *Delete* gesetzten Elemente von $S \subseteq U$ an.
- Statt im Bitvektor $Bit[0..N-1]$ könnte die Information auch direkt als zusätzliche Information von jeweils einem Bit im Pointer-Array $Ptr[0..N-1]$ gespeichert werden.
- Die Methode zum Optimieren der Initialisierungskosten funktioniert auch für das Initialisieren großer Arrays in numerischen Rechnungen.
- Der Index des Arrays *Stack* kann statt eines Bits auch einen Pointer auf einen Speicherbereich enthalten, in dem der gesamte Record gespeichert ist.

Komplexität

Wir werfen einen Blick auf die Komplexität der Suche mittels der Bitvektordarstellung und vergleichen diese dann mit den herkömmlichen Verfahren.

- 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): $O(1)$

Folgende Tabelle stellt die unterschiedlichen Suchverfahren in ihrer Komplexität gegenüber, dabei ist

$N = |U|$ 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	ld n^*	n	n	N
Bitvektor	1	1	1	N

Tabelle: O-Komplexität der Suche.

Anmerkungen:
1^*: Falls keine Duplikate vorkommen sollen: n statt 1.
ld n^*: Search ist mit Binärsuche implementiert.

3.3 Hashing

3.3.1 Grundbegriffe

Sowohl BucketSort als auch die Bitvektor-Darstellung basieren auf dem Prinzip, aus dem Schlüssel eines Elementes direkt die Adresse abzulesen. Der Nachteil dieser Methoden ist, daß die Platzkomplexität proportional zur Größe des Universums ist. Bei vorgegebener kleinerer Platzkomplexität bietet sich das *Hashing* an. Beim Hashing, im Deutschen auch *Schlüsseltransformation* oder *Streuspeicherung* genannt, wird ebenfalls aus dem Schlüssel die Adresse berechnet, jedoch ist die Adresse nicht mehr unmittelbar umkehrbar. Somit kann man die Berechnung so auslegen, daß genau m Speicherplätze belegt werden. Allerdings kann es dann zu Mehrfach-Belegungen (*Kollisionen*) kommen.

Hashverfahren können nach einigen Kriterien unterschieden werden:

- Die *Hashfunktion* $h(x)$ gibt vor, wo ein Schlüssel x in die *Hashtabelle* T eingetragen wird.

- Falls die berechnete Adresse schon belegt ist benötigt man eine *Kollisionsstrategie*. Dabei kann nach *offenen* und *geschlossenen* Verfahren unterschieden werden. Erstere arbeiten dabei dynamisch mit verketteten Listen, letztere müssen den verfügbaren Speicher sondieren, um einen Speicherplatz für das Element zu finden. Dazu eignet sich
  - lineares Sondieren,
  - quadratisches Sondieren und
  - Doppelhashing.
- Erweiterbares Hashing erlaubt die Verbindung mit Hintergrundspeicher. [Mehlhorn, S.135]

Prinzip

Gegeben sei eine zur Verfügung stehende Anzahl von Speicherplätzen m in einer

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

Dann wird ein Schlüssel $x \in$ Universum $U = \{0, \ldots, N-1\}$ abgebildet auf einen Speicherplatz in der Hashtabelle. Dazu dient die

Hashfunktion $h(x)$:

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

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. $h(x) = h(y)$, für $x \neq y$.

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. Denn dort hat man ein sehr großes Universum U, nämlich die Menge aller Zeichenketten mit der maximalen Länge 20 (z.B. Namen). 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, und für $|S| \approx 1000$ auch unsinnig.

3.3.2 Hashfunktionen

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

- Die ganze Hashtabelle soll 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]

Divisions-Rest-Methode

Seien die Schlüssel $x \in \mathbb{N}_0$ (fast alle in der Praxis vorkommenden Schlüssel lassen sich entsprechend umwandeln). Dann definiert man $h(x)$ wie folgt:

$$h(x) = x \bmod m$$

Dabei ist m die Größe der Hashtabelle, auf die es bei der Qualität der Verteilung stark ankommt. Bestens 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 besticht durch die Einfachheit der Berechnung, hat aber einen Nachteil:

Aufeinanderfolgende Schlüssel werden auf aufeinanderfolgende Speicherplätze abgebildet. Das führt zu einem unerwünschten Clustering, welches die Effizienz des Sondierens reduziert (s. Abschnitt 3.3.3).

Dennoch ist die Divisions-Rest-Methode eine der effizientesten und auch weitest verbreiteten.

Beispiel: Verteilen von Namen über m Behälter mittels der Divisions-Rest-Methode [Güting, Seite 97].

Gegeben seien Zeichenketten $c = c_1 \ldots c_k$. Diese müssen zunächst mittels einer geeigneten Methode auf $\mathbb{N}_0$ abgebildet werden:

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

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

Zur Vereinfachung betrachten wir nur die ersten drei Buchstaben, dann läßt sich für c folgendermaßen ein Platz in der Hashtabelle errechnen:

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

Wir wählen $m = 17$; $S =$ Abkürzungen der (deutschen) Monatsnamen

0	NOV	
1	APR	DEZ
2	MAE	
3		
4		
5		
6	MAI	SEP
7		
8	JAN	
9	JUL	
10		
11	JUN	
12	AUG	OKT
13	FEB	
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 (s.Abschnitt 3.3.3) aufeinanderfolgender Zahlen zu verhindern.

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

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

Allerdings eignen sich als Größe der Hashtabelle nur Werte, die Potenzen der Basis des Zahlensystems entsprechen.

Sei $m = 100$ (in der Tat eine schlechte Wahl für die Divisions-Rest-Methode), dann ergibt eine Gegenüberstellung der Methoden

x	$x \bmod 100$	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 Kollisionsstrategien (Sondieren)

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

Wahrscheinlichkeit von Kollisionen

In der diskreten Mathematik besteht ein analoges, unter dem Namen *Geburtstags-Paradoxon* bekanntes Problem. Es lautet:

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

Die Analogie zum Hashing ist naheliegend:
Größe der Hashtabelle $m = 365$ Tage
Zahl der Schlüssel $n =$ Anzahl der Personen
Eine Kollision ist dann gleichbedeutend mit dem gleichen Geburtstag.

Annahme: Die Hash-Funktion sei ideal, d.h. die Verteilung über die Hash-Tabelle sei absolut gleichmäßig.
Für die Wahrscheinlichkeit mindestens einer Kollision bei n Schlüsseln und einer m-elementigen Hashtabelle $Pr(Kol|n, m)$ gilt folgendes:

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

Die Wahrscheinlichkeit $Pr(NoKol|n, m)$ läßt sich recht einfach berechnen.

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. Dann 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 $p(n; m)$

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

Für die Tabelle zum Geburtstagsproblem ($m = 365$) ergeben sich damit folgende Werte in Abhängigkeit von der Zahl der Personen:

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

Interessant ist nun, wie m mit n wachsen muß, damit $Pr(Kol|n,m)$ konstant bleibt. Die obige Formel gibt auf diese Frage nur eine indirekte Antwort. Es läßt sich aber mit folgender Vereinfachung ein direktes Ergebnis herleiten:

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

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

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

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 der Hashtabelle m 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.

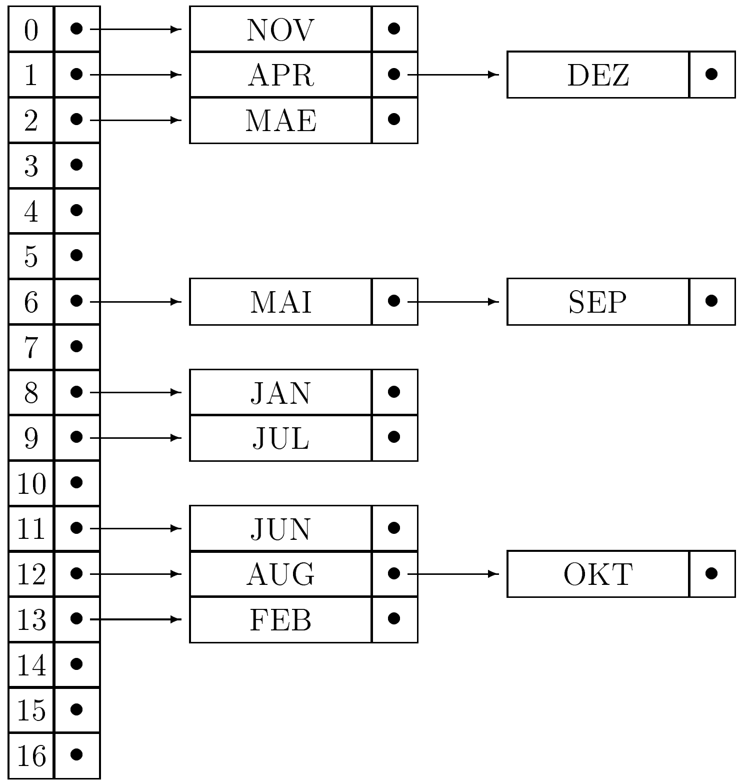
Geschlossene Hashverfahren hingegen können nur eine begrenzte Zahl von Schlüsseln (meistens nur einen) unter einem Eintrag unterbringen. Sie arbeiten mit einem Array und werden wegen des abgeschlossenen Speicherangebots als geschlossen bezeichnet. Sie müssen mittels Sondierungsfunktionen neue Adressen für Überläufer suchen und werden daher auch *Hashverfahren mit offener Adressierung* genannt (nicht zu verwechseln mit offenen Hashverfahren).

Offenes Hashing (Hashing mit Verkettung)

Die offenen Hashverfahren arbeiten mit verketteten Listen und sind daher dynamisch im Speicherplatzbedarf. Dazu wird jeder *Behälter* (Platz in der Hashtabelle) durch eine beliebig erweiterbare Liste von Schlüsseln dargestellt. Ein Array von Zeigern verwaltet die Behälter.

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

Erinnern wir uns an obiges Beispiel mit den Monatsnamen, dann stellt sich das Array (die Hashtabelle) folgendermaßen dar:



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: Abhängig vom Belegungsfaktor α ist also das Verhalten dieses Hashings, das über die Wahl des Hashingverfahrens entscheidet.

- $\alpha \ll 1$: Die Listen sind sehr kurz, daher ist das Verhalten ähnlich einer Array-Addressierung.
- $\alpha \approx 1$: Die Listen sind mäßig lang. Man befindet sich im Übergangsbereich.
- $\alpha \gg 1$: Die Listen sind sehr lang, daher ähnelt das Gesamtverhalten auch dem einer verketteten Liste.

Geschlossene Hashverfahren

[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.

Die Adresse ergibt sich dann aus der Summe der Ergebnisse der eigentlichen Hashfunktion und der Sondierungsfunktion. Dieses Ergebnis muß dann noch mittels mod auf den gültigen Adressbereich zurückgeführt werden.

So, daß dann für jedes $x \in U$ eine Folge $h(x, j)$ $(j = 0, 1, \ldots, m - 1)$ von Positionen in der Hashtabelle definiert ist.

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

Lineares Sondieren ist die einfachste Art zu Sondieren, dabei werden einfach der Reihe nach alle Plätze ausgehend von der ursprünglichen Hashadresse überprüft. Also gilt dann für die (Re-)Hashfuntkion

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

Lineares Sondieren hat einen entscheidenden Nachteil, es 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 denkbar ineffizient.

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

0	NOV	
1	APR	DEZ
2	MAE	
3		
4		
5		
6	MAI	SEP
7		
8	JAN	
9	JUL	
10		
11	JUN	
12	AUG	OKT
13	FEB	
14		
15		
16		

Quadratisches Sondieren

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

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

Es läßt sich zeigen, daß i^2 eine gute Wahl ist, denn wenn m eine Primzahl ist, dann sind die Zahlen $i^2 \bmod m$ alle verschieden für $i = 0, 1, \ldots, \left\lfloor \frac{m}{2} \right\rfloor$.

Weiter gilt folgende Verfeinerung:

$$\begin{aligned} h(x, 2j-1) &= (h(x){+}j^2) \bmod m \qquad 0 \leq j \leq \frac{m-1}{2} \\ h(x, 2j) &= (h(x){-}j^2) \bmod m \end{aligned}$$

Für das quadratische Sondieren ist m prim mit $m \bmod 4 = 3$ eine besonders gute Wahl, wie sich über die Zahlentheorie [Radke (1970)] zeigen läßt.

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 $h'(x)$ idealerweise so, daß gilt

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

und die beiden Funktionen h und h' unabhängig sind. Denn dann gilt

$$Pr(h(x) = h(y) \wedge h'(x) = h'(y)) = \frac{1}{m^2}$$

Also kann man wie folgt eine sehr gute, kaum vom idealen Hashing unterscheidbare (Re-)Hashfunktion definieren

$$h(x,i) = (h(x) + h'(x) \cdot i^2) \bmod m$$

3.3.4 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 tritt keine Häufung auf.

Sei $q(i; n, m)$ die Wahrscheinlichkeit, daß für eine m große Hashtabelle mindestens i Kollisionen bei der Eintragung von n Schlüsseln auftreten.

Diese Wahrscheinlichkeit ist gleich der, daß die ersten $i-1$ Positionen $h(x,0), h(x,1), \ldots, h(x, i-1)$ der Sondierungs-Folge besetzt sind.

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 $n-1$ voll 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 $n-(i-1)$ voll sind. Mittels Rekursion ergibt sich so

$$q(i; n, m) = q(i-1; n, m) \cdot \frac{n-i+1}{m-i+1}$$

$$\begin{aligned}
&\vdots \\
&= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \ldots \cdot \frac{n-i+1}{m-i+1} \\
&= \prod_{j=0}^{i-1} \frac{n-j}{m-j}
\end{aligned}$$

- $C_{Ins}(n,m)$ 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:

$$C_{Del}(n,m) := C^{+}_{Sea}(n,m)$$

Als **Näherung** für $n, m \gg 1$ benutzt man:

$$q(i;n,m) = \left(\frac{n}{m}\right)^i$$

und kann die Formeln der geometrischen Reihe nutzen, um die Kosten einfacher zu berechnen:

$$\begin{aligned} C^{-}_{Sea}(n,m) = C_{Ins}(n,m) &\approx \sum_{i=0}^{\infty} \left(\frac{n}{m}\right)^i \\ &= \frac{1}{1-\alpha} \end{aligned}$$

$$\begin{aligned} C^{+}_{Sea}(n,m) = C_{Del}(n,m) &\approx \frac{1}{n}\int_0^{n-1} \frac{m}{m-x}\,dx \\ &= \frac{m}{n}\ln\frac{m}{m+1-n} \\ &\approx \frac{1}{\alpha}\ln\frac{1}{1-\alpha} \end{aligned}$$

Nichtideales Hashing

Hashen wir mit einer nicht-idealen Hashfunktion, gehen also von Clustering aus, z.B. beim linearen Sondieren, dann erhält man andere Ergebnisse [Knuth (1973)]. Charakteristisch sind die folgenden beiden Größen.

$$\begin{aligned} C^{+}_{Sea}(n,m) &= \frac{1}{2}\left[1 + \frac{1}{1-\alpha}\right] \\ C^{-}_{Sea}(n,m) &= \frac{1}{2}\left[1 + \frac{1}{1-\alpha^2}\right] \end{aligned}$$

3.3.5 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.

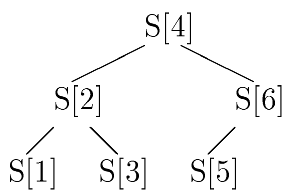
3.4 Binäre Suchbäume

3.4.1 Allgemeine binäre Suchbäume

[Mehlhorn, Seite 140] Ausgehend von der binären Suche (s. Abschnitt 3.2.2) sind die binären Suchbäume zunächst nur eine anschauliche Darstellung, die sich aber schnell verselbständigt hat. Für die binäre Suche gilt

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

Die Veranschaulichung eines Arrays S[1..6] durch einen binären Baum sieht im allgemeinen so aus



Wieder muß man bei der Betrachtung der **Zeitkomplexitäten** die Array- und die Zeiger-Darstellung unterscheiden.

- Die *Search-Operation* sind beide noch gleich gut. Da der Wert von $[high - low + 1]$ bei jedem Durchgang der **while** -Schleife (s. Seite 85) halbiert wird ergibt sich eine maximale Zeitkomplexität von

$$T(n) \in O(\,\mathrm{ld}\, n)$$

- Für die Insert- und Delete-Operationen zeigt sich die Array-Darstellung problematisch, denn z.B. erfordert Insert das Verschieben eines Teils des Arrays, und daher gilt dann für das **Array**

$$T(n) \in O(n)$$

  Ausweg: Wird der obige Suchbaum durch **Zeiger** statt durch ein Array realisiert, laufen auch die Operationen *Delete* und *Insert* effizient ab mit

$$T(n) \in O(\,\mathrm{ld}\, n)$$

Definition *Binärer Suchbaum*: Ein binärer Suchbaum für die n-elementige Menge $S = \{x_1 < x_2 < \ldots < x_n\}$ ist ein Binärbaum mit n Knoten $\{v_1, \ldots, v_n\}$.

Die Knoten sind mit den Elementen von S beschriftet, d.h. es gibt eine injektive Abbildung

$$Inhalt : \{v_1, \ldots, v_n\} \to S$$

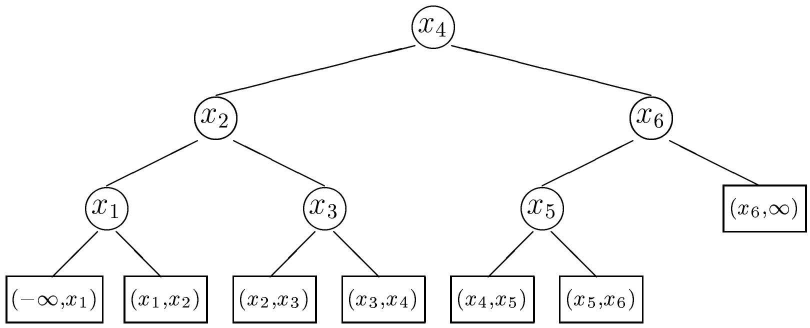
Dabei bewahrt die Beschriftung die Ordnung, d.h. wenn v_i im linken Unterbaum der Wurzel des Baumes v_k ist, und v_j im rechten Unterbaum, dann gilt

$$\mathit{Inhalt}[v_i] < \mathit{Inhalt}[v_k] < \mathit{Inhalt}[v_j]$$

Eine äquivalente Definition ist: Ein binärer Suchbaum ist ein Binärbaum, dessen *Inorder-Durchlauf* die Ordnung auf S ergibt.

In der Regel identifizieren wir Knoten mit ihrer Beschriftung: Knoten v mit Beschriftung x bezeichnen wir also auch mit „Knoten x“.

Beispiel: Binärer Suchbaum



Die *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.

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 $a = \mathit{Inhalt}[v]$
- in einem Blatt (x_i, x_{i+1}) mit $x_i < a < x_{i+1}$, falls $a \notin S$

Beispiel: Deutsche Monatsnamen.
Dargestellt ist ein binärer Suchbaum in lexikographischer Ordnung entstanden durch Einfügen der Monatsnamen in kalendarischer Reihenfolge.

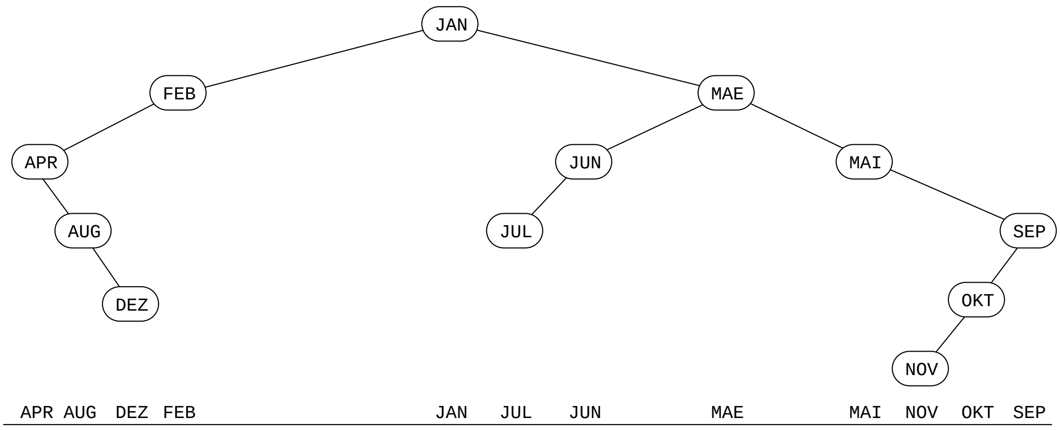


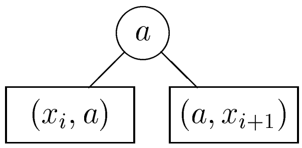
Abbildung 18: Binärer Suchbaum für deutsche Monatsnamen

Die Inorder-Traversierung bzw. die Projektion auf die x-Achse erzeugt die alphabetische Reihenfolge.

Übung: Erstellen Sie den Baum für die umgekehrte Reihenfolge.

Die Standard-Operationen stellen sich folgendermaßen dar:

- 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. Man unterscheidet nun:

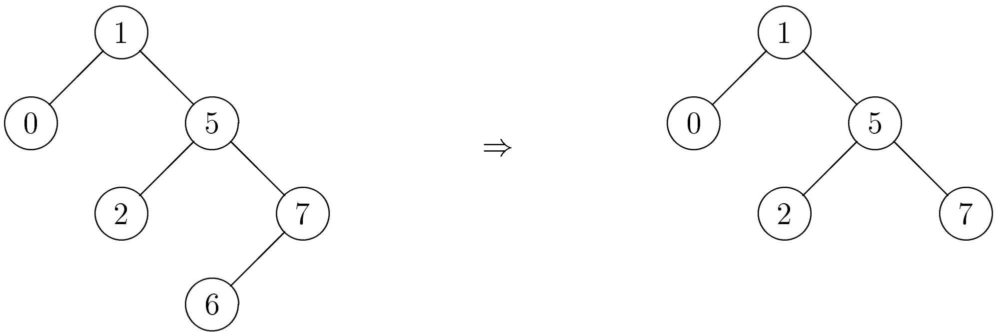
  (a) Der Knoten v hat mindestens ein Blatt als Sohn:
  - ersetze v durch den anderen Sohn und
  - streiche v und das Blatt aus dem Baum.

(b) Der Knoten v hat zwei Söhne die innere Knoten sind:
   – Ersetze den Knoten v durch w, den rechtesten Unterknoten im linken Unterbaum von v.
     Diesen findet man durch folgendes Vorgehen:
     * Wähle einmal $LeftSon[v]$
     * Wähle rekursiv immer wieder $RightSon[v']$ des aktuellen Knotens, bis man auf einen Knoten ohne rechten Sohn trifft.
   – Um w von seinem ursprünglichen Platz zu entfernen, geht man vor wie im Fall (a).

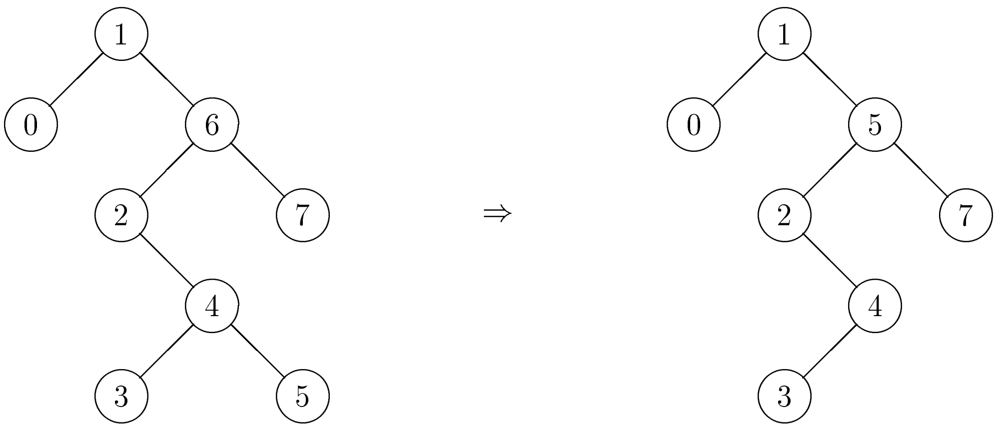
Dabei bleibt die Suchbaum-Eigenschaft erhalten!

Beispiel: Delete(6,S) [Mehlhorn, Seite 143]

Fall (a): Der Knoten 6 hat mindestens ein Blatt als Sohn.



Fall (b): Der Knoten 6 hat zwei Söhne.



Zeitkomplexitäten

Wir untersuchen wieder die drei Standard-Operationen *Search*, *Insert* und *Delete*. Dazu muß man sich darüber im Klaren sein, daß man dazu im wesentlichen einen Weg des Suchbaums von der Wurzel bis zu einem Blatt verfolgen muß. Aus diesem Grunde liegt ihre Komplexität dann auch in

$$O(h(T))$$

$h(T)$ ist dabei die Höhe des Suchbaumes T, für die im Best Case gilt $O(\text{ ld } n)$ und im Worst Case $O(n)$.

Die Operation *ListOrder(S)* entspricht laut Definition dem Traversieren in symmetrischer Ordnung (inorder) und besitzt daher eine Komplexität

$$O(n)$$

Die Operation *Order(k, S)*, also die Ausgabe der k kleinsten oder größten Elemente, benötigt eine geringfügige Erweiterung des Suchbaums:

- In jedem Knoten wird zusätzlich die Anzahl der Knoten seines linken Unterbaumes gespeichert. Mit $O(1)$ kann dieser Wert bei Delete- und Insert- Operationen aktualisiert werden.
- Damit kann man die Funktion *Order* dann 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.

Wie auch schon bei allgemeinen Bäumen (s. Abschnitt 1.3.4) muß man sich über die Eigenheiten der Implementierung eines Baumes verständigen. Die augenfälligsten Merkmale sind

- das *Head-Element* (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 grafischen Darstellung kann man diese Unterschiede schnell entnehmen:

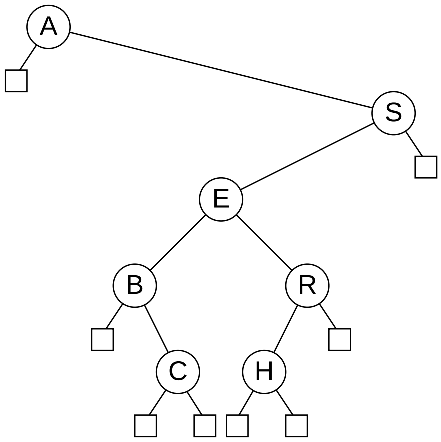


Abbildung 19: Binärer Suchbaum

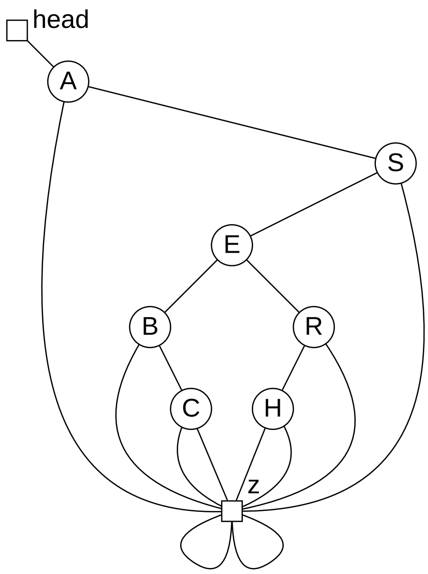


Abbildung 20: Implementation mit Dummy-Knoten

Implementierung

Bis auf *Delete(v, S)* sind die o.a. Operationen recht einfach zu implementieren.

```
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;
```

Nun zur Implementation der Funktion *Delete*. Wie oben schon besprochen muß ein passendes Element an die Position des zu löschenden Elementes treten, damit die Suchbaum-Eigenschaften erhalten bleiben.

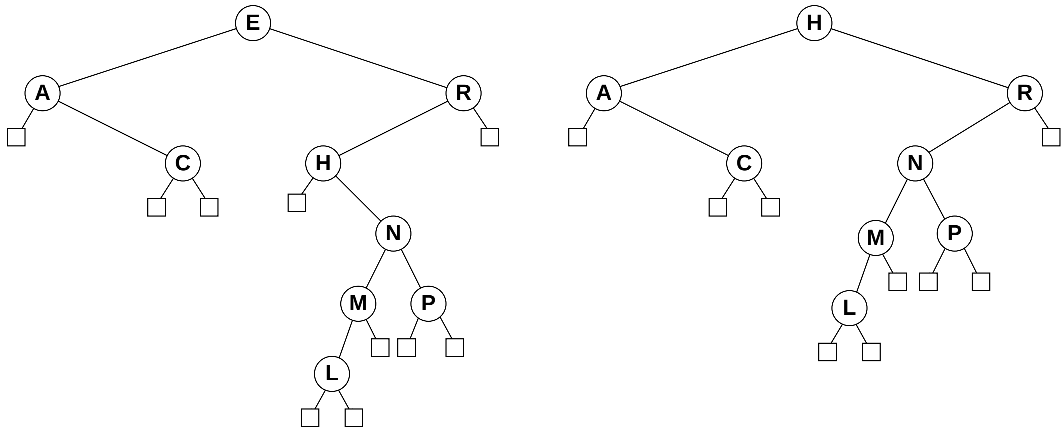


Abbildung 21: Beispiel eines Baumes T vor und nach **Delete**(T, E)

Um das im Bild 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.

Im Code präsentiert sich dann eine Prozedur die den Knoten t aus dem von x induzierten Teilbaum löscht folgendermaßen. Dabei wird in x der Knoten gespeichert, der an die Stelle des gelöschten treten soll.

```
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;
```

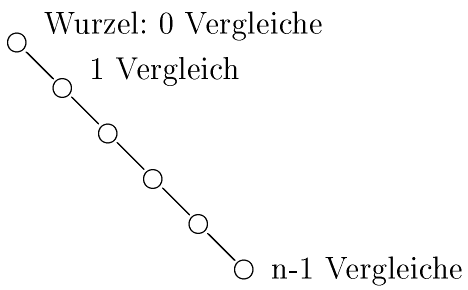
Mit diesen Prozeduren und Funktionen lassen sich die fünf Operationen folgendermaßen realisieren:

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

Wir betrachten die Konstruktion eines binären Suchbaums durch n Insert-Operationen. Dabei sei $C(n)$ die Zahl der dazu benötigten Vergleiche. Dann gilt für die üblichen Fälle:

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

   $$C(n) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

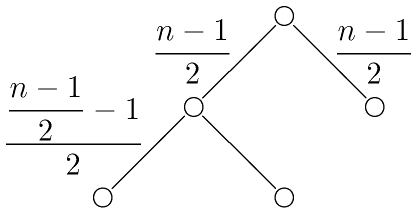
   und für den *Aufwand*

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

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.Abschnitt 1.3.4).

   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.

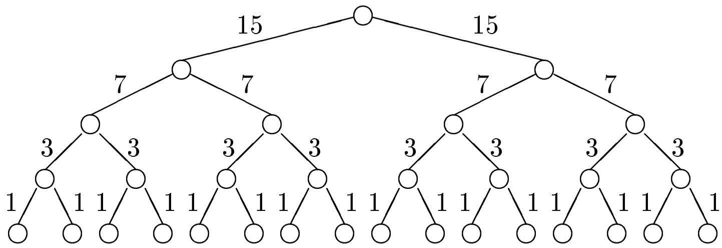
Folgende Darstellung veranschaulicht die Verteilung der n Datensätze auf die Zweige des Binärbaumes.



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 $n = 2^h - 1 = 31$.



$$\begin{aligned} 2 \cdot 15 &= 30 \\ 4 \cdot 7 &= 28 \\ 8 \cdot 3 &= 24 \\ 16 \cdot 1 &= 16 \end{aligned}$$

Summe: 98 Vergleiche

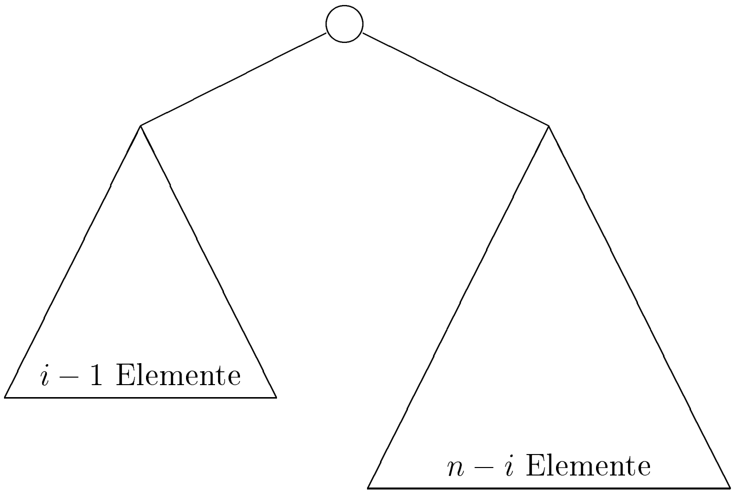
Summiert man nun über die Anzahl *aller* Vergleiche, so erhält man

$$\begin{aligned} C(n) &= \sum_{i=1}^{h-1} \underbrace{2^i}_{\text{Äste}} \cdot \underbrace{(2^{h-i} - 1)}_{\text{Vergleiche pro Ast}} \\ &= \sum_{i=1}^{h-1} (2^h - 2^i) \\ &= (h-1) \cdot 2^h - \left[\sum_{i=0}^{h-1} 2^i - 1\right] \\ &= (h-1) \cdot 2^h - \frac{2^h - 1}{2 - 1} - 1 \\ &= 2^h \cdot (h-2) + 2 \\ &= (n+1) \cdot \operatorname{ld}(n+1) - 2n \end{aligned}$$

und für den Aufwand

$$\frac{C(n)}{n} \approx \operatorname{ld} n$$

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, die denen von QuickSort (s.Abschnitt 2.3) entsprechen:

$$\begin{aligned} C(n) &= \frac{1}{n}\sum_{i=1}^{n}[n-1+C(i-1)+C(n-i)] \\ &= n-1+\frac{2}{n}\sum_{i=0}^{n-1}C(i) \\ &= 2n\cdot\ln n+\ldots \\ &= 2n\cdot(\ln 2)\,\mathrm{ld}\,n+\ldots \\ &= 1.386n\,\mathrm{ld}\,n+\ldots \end{aligned}$$

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] Ein optimaler binärer Suchbaum S ist ein gewichteter Baum, dessen Gewichtung der Zugriffsverteilung entspricht, die die Häufigkeit oder Wichtigkeit der Elemente von S widerspiegelt.

In der nachfolgenden Betrachtung wollen wir uns nur auf die Funktion *Search*(a, S) beschränken.

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 dann 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]

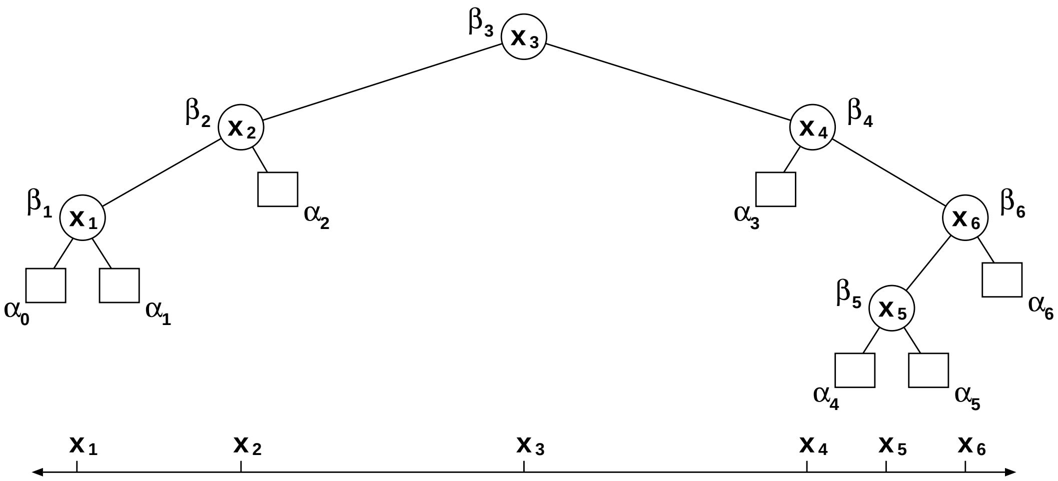


Abbildung 22: Baumdarstellung der Zugriffsverteilung

Komplexität von Search(a, S)

Wir wollen hier nur den *Average Case*, also mittlere Zahl von Vergleichen betrachten. Falls erfolgreich, endet die Suche in einem inneren Knoten; falls nicht, endet sie in einem Blatt. Die Tiefe des Knotens bzw. Blattes ergibt direkt die Zahl der benötigten Vergleiche.

Gegeben sei ein Suchbaum, für den

- b_i die Tiefe des Knotens x_i ist, und
- a_j die Tiefe des Blattes (x_j, x_{j+1}).

Die mittlere Anzahl von Vergleichen wird dann wiedergegeben durch die (gewichtete) Pfadlänge P, für die gilt

$$P = \sum_{i=1}^{n} (\beta_i \cdot (1 + b_i)) + \sum_{j=0}^{n} (\alpha_j \cdot a_j)$$

Beachte: Ein Blatt erfordert dabei einen Vergleich weniger als ein Knoten.

Nun wollen wir einen optimalen Suchbaum konstruieren, der bei gegebener Zugriffsverteilung eine minimale mittlere Anzahl von Vergleichen benötigt.
Dazu bedienen wir uns wieder der *Dynamischen Programmierung*.

Bellman/Knuth-Algorithmus

[Ottmann, Seite 397] 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.

An diesem Punkt setzt die dynamische Programmierung an, für die wir eine Hilfsgröße $c(i,j)$ mit $i < j$ definieren:

$$c(i,j) := \text{ optimale Pfadlänge für den Teilbaum } \{x_i, \ldots, x_j\}$$

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.

Daher gilt mit $\quad w(i,j) = \sum_{l=i}^{j} \alpha_l + \sum_{l=i+1}^{j} \beta_l$

$$\begin{aligned} c(i,i) &= 0 \qquad \text{für alle } i \\ c(i,j) &= w(i,j) + \min_{i<k\leq j} [c(i,k-1) + c(k,j)] \quad \text{für } i < j \end{aligned}$$

Diese DP-Gleichung (Gleichung der dynamischen Programmierung, Bellmansche Optimalitätsgleichung) kann man iterativ lösen und erhält das Ergebnis für die Pfadlänge

$$P = c(0,n)$$

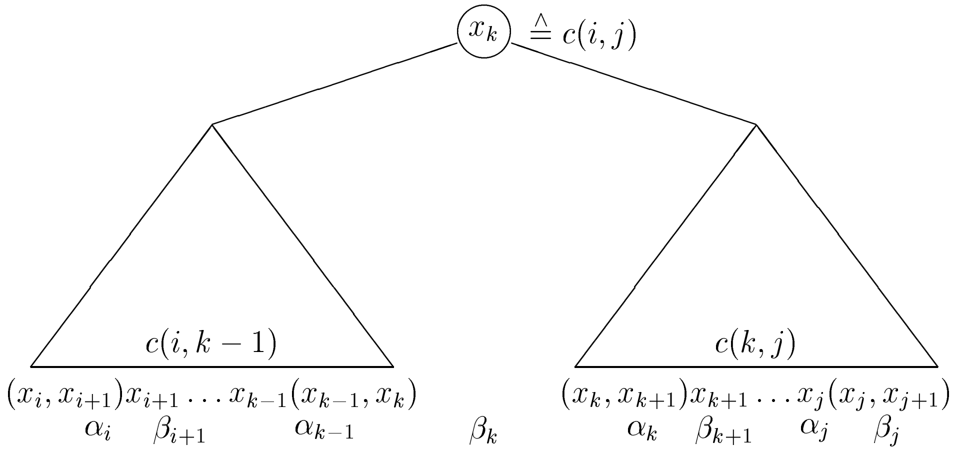


Abbildung 23: Schematische Darstellung der Teilbäume.

Programm-Entwurf zur iterativen Lösung der DP-Gleichung

Die Lösung ist ähnlich der für das Matrix-Produkt (s.Abschnitt 1.4.3). 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] die Indizes der Vorgänger und dient der Rekonstruktion des optimalen Suchbaums.

Dann geht man wie folgt vor:

1. Initialisiere zunächst $c(i,i) = 0$ für alle $i = 0, \ldots, n$,
   und berechne vorab $w(i,j)$ für alle $i, j = 0, \ldots, n$.

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**
           $j := i + l$
           $c(i,j) = w(i,j) + \min\limits_{i<l\leq j}[c(i, l-1) + c(l, j)]$
           $r(i,j) = \arg\min\limits_{i<l\leq j}[c(i, l-1) + c(l, j)]$

3. Rekonstruktion des optimalen Suchbaums mittels des Indexarrays $r(i,j)$

Obigem Entwurf kann man direkt die ungefähren Komplexitäten der Konstruktion eines optimalen Suchbaumes entnehmen und erhält

- die *Zeitkomplexität* $O(n^3)$ und
- die *Platzkomplexität* $O(n^2)$.

An folgendem Beispiel wollen wir den Algorithmus veranschaulichen.

Beispiel [Aigner, Seite 187]

Gegeben sei folgende Zugriffsverteilung

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

Warnung:
Die naive Implementierung mittels Rekursion hat eine *exponentielle* Komplexität.

Man muß also bei der Dynamischen Programmierung darauf achten, daß man bei iterativem Vorgehen die Tabelle geschickt aufbaut oder mittels Memoization die Zwischenwerte in einer Tabelle speichert.

Eine *Verbesserung der Komplexität* von $O(n^3)$ auf $O(n^2)$ kann man mittels der „Vierecksungleichung“ für $r(i, j)$ erreichen, wenn man sich die *Monotonie-Bedingung* zunutze macht. Diese sagt für die Matrix $r(i, j)$ aus, daß

$$r(i, j-1) \leq r(i, j) \leq r(i+1, j) \;\; \forall i, j \;\in 0, \ldots, n$$

[Mehlhorn, Seite 151]

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.

Hier schaffen gerade die **Balancierten Bäume** Abhilfe. Sie garantieren Komplexitäten $O(\log n)$, indem sie durch bestimmte *Balance-Funktionen* eine Entartung der Baumstruktur verhindern.

Grundsätzlich kann man zunächst (mindestens) zwei Balance-Kriterien unterscheiden:

1. *Gewichtsbalancierte Bäume*: Diese, auch BB(a)-Bäume (**B**ounded **B**alance) genannt, benutzen als Balance-Kriterium die Anzahl der Blätter in den Unterbäumen, die möglichst gleich gehalten wird. Dabei gibt a die Beschränkung (Boundary) der Balance an, also den maximalen relativen Unterschied in der Zahl der Blätter der zwei Teilbäumen eines 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 aufgrund besonders guter Merkmale nochmal als *B-Baum* bezeichnet. Ein besonders bekannter Vertreter ist der (2,3)-Baum. [Mehlhorn, Seite 186]

3.5.1 AVL-Bäume

[Güting, Seite 120] Im Jahre 1962 entwarfen Adelson-Velskij und Landis diese erste Variante der balancierten Suchbäume. Der damalige Entwurf war noch nicht komplett ausgereift, so daß er nicht die effizienteste Struktur darstellt, aber sich aufgrund seiner Einfachheit das Prinzip der balancierten Bäume daran sehr gut darstellen läßt.

Definition *AVL-Baum*: 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.

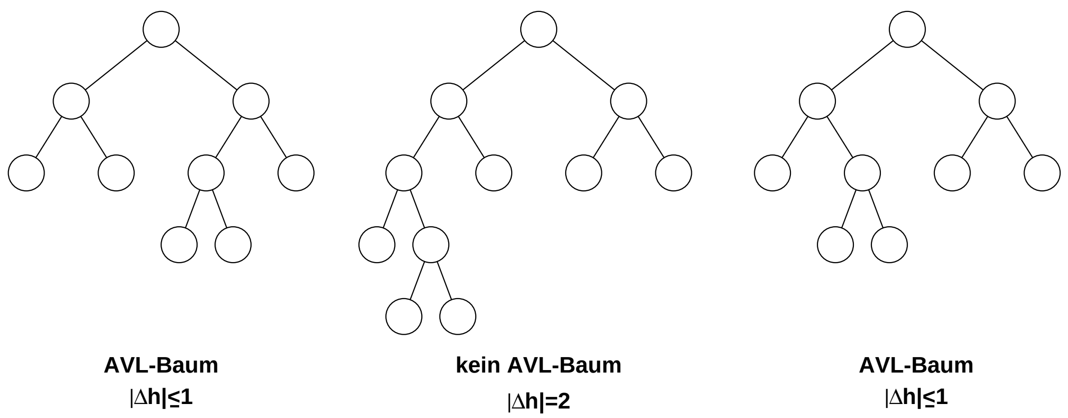


Abbildung 24: Beispiele für AVL-Bäume

Höhe eines AVL-Baumes

Bevor wir uns den Rebalancierungs-Operationen zuwenden, wollen wir untersuchen, wie hoch ein AVL-Baum bei vorgegebener Knotenzahl maximal, also im Worst Case sein kann.
Sei $N(h)$ die minimale Anzahl der Knoten in einem AVL-Baum der Höhe h.

Wir betrachten also einen minimal gefüllten Baum, also auch einen Baum maximaler Höhe.

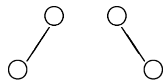
Ein AVL-Baum läßt sich nun folgendermaßen rekursiv definieren, und wir werden sehen, wie sehr diese Definition der der Fibonacci-Zahlen (s. Abschnitt 1.2.4) ähnelt.

- $h = 0$: Der Baum besteht nur aus der Wurzel und es gilt

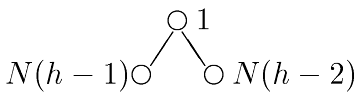
$$N(0) = 1$$

- $h = 1$: Da wir minimal gefüllte Bäume betrachten enthält die nächste Ebene nur einen Knoten, und es gilt

$$N(1) = 2$$



- $h \geq 2$: Da auch jeder Teilbaum eines AVL-Baumes das AVL-Kriterium erfüllen muß, können wir *rekursiv* zwei Bäume der Höhe $h-1$ und $h-2$ zu einem neuen, minimal gefüllten Baum der Höhe h kombinieren.



Daraus folgt für die Anzahl der Knoten eines minimal gefüllten AVL-Baumes diese Rekursionsgleichung

$$N(h) = 1 + N(h-1) + N(h-2)$$

Die der Rekursion der Fibonacci-Zahlen $Fib(n)$ sehr ähnlich ist. Vereinfacht läßt sich dann schreiben

$$N(h) = Fib(h+3) - 1 \leq n$$

wobei n die Anzahl der Knoten eines beliebigen AVL-Baumes ist.

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

$$\begin{aligned} Fib(h+3) &= \frac{1}{\sqrt{5}}\left[\Phi^{h+3} - \phi^{h+3}\right] \\ &\geq \frac{1}{\sqrt{5}}\Phi^{h+3} - \frac{1}{2} \end{aligned}$$

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

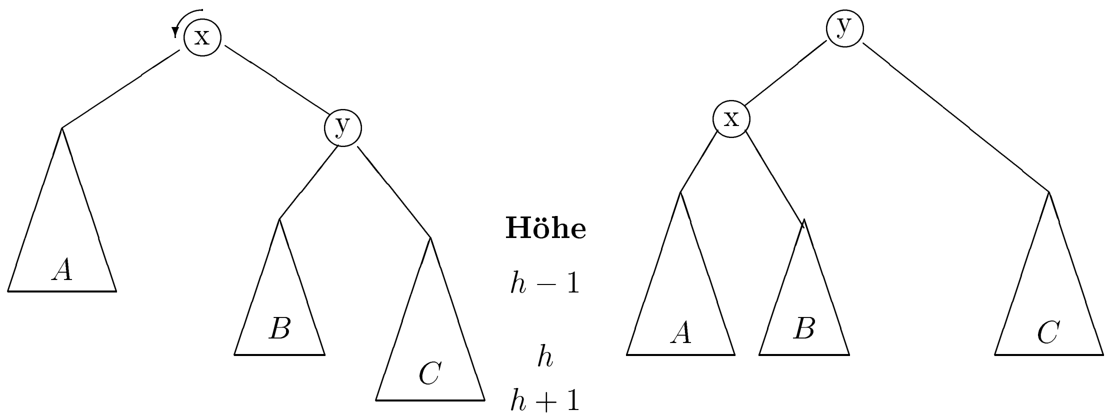
Nun wollen wir uns den Operationen auf AVL-Bäumen zuwenden. Dabei können zunächst die normalen Operationen wie gehabt ausgeführt werden, jedoch muß im Anschluß die Balance überprüft, und gegebenenfalls wiederhergestellt werden.

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 $O(\mathrm{ld} n)$.

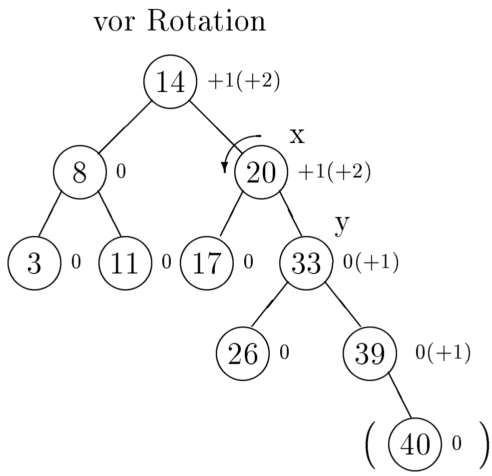
Ist die Balance an einer Stelle gestört, so kann sie in konstanter Zeit ($O(1)$) mittels einer *Rotation* oder *Doppel-Rotation* wieder hergestellt werden.

Ein AVL-Baum ist in einem Knoten aus der Balance, falls die beiden Teilbäume dieses Knotens einen Höhenunterschied größer eins aufweisen.

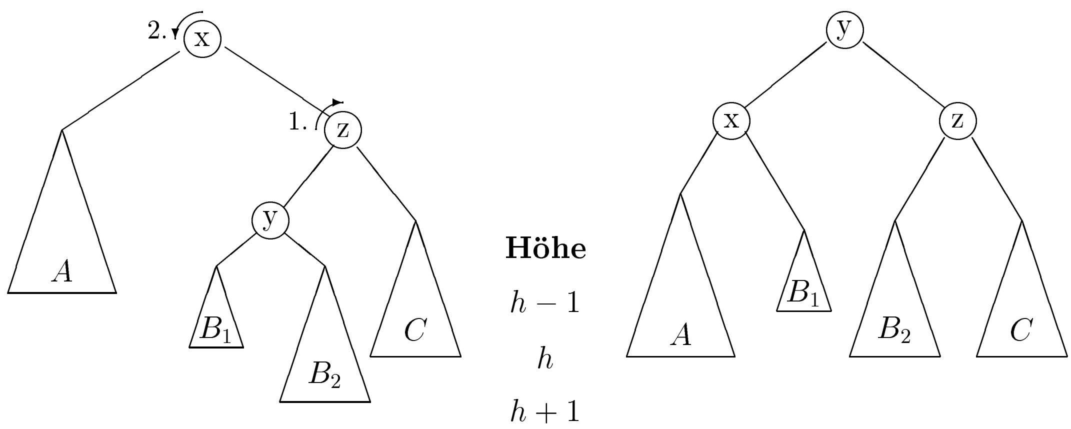
- **Rotation**: Ist ein Knoten betroffen (x), so betrachtet man nur den von ihm induzierten Teilbaum. Eine einfache Rotation muß durchgeführt werden, wenn der betroffene (also höhere) Teilbaum (C) *außen* liegt. Dann rotiert der betroffene Knoten zum kürzeren Teilbaum runter und übernimmt den inneren Sohn (B) des heraufrotierenden Knotens (y) als inneren Sohn.



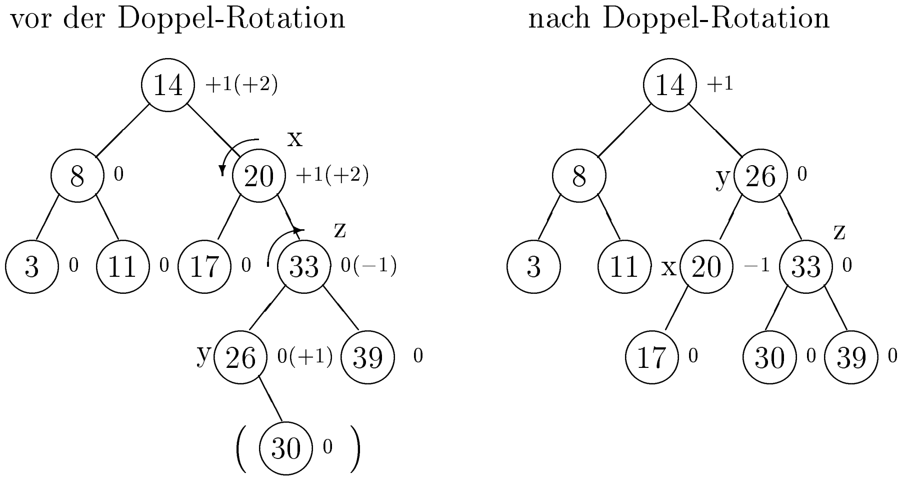
Zur Veranschaulichung ein Beispiel: In den folgenden AVL-Baum sei das Element **40** eingefügt worden, wodurch die AVL-Bedingung im Element **20** verletzt wurde.



- **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. Dazu 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: Wie man leicht nachvollziehen kann müssen insgesamt nur vier Pointer umdirigiert werden, um eine Rotation bzw. Doppel-Rotation durchzuführen.

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

Die (a,b)-Bäume stellen eine Erweiterung der AVL-Bäume dar. Dabei geben a und b die Grenzen für die Anzahl der Söhne an, die ein Knoten besitzen kann.

Definition *(a,b)-Baum* [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

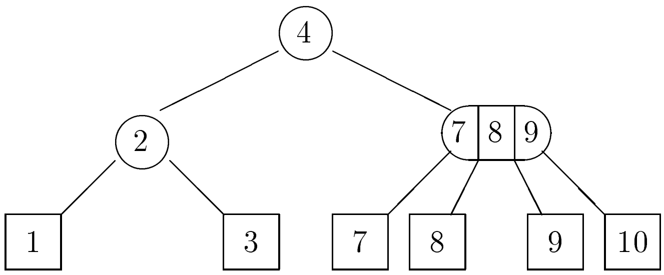
1. alle Blätter die gleiche Tiefe haben,
2. für jeden Knoten v gilt $\rho(v) \leq b$,
3. für alle Knoten v *außer der Wurzel* gilt $a \leq \rho(v)$, und
4. die Wurzel mindestens zwei Söhne hat.

b heißt auch die *Ordnung* des Baumes. 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

Ohne im Detail auf die blattorientierte Speicherung einzugehen, wollen wir kurz das Prinzip schildern, nach dem ein solcher Baum aufgebaut ist. [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:

$$Inhalt(w) = x$$

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 Knoten:

   - 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:

$$\begin{array}{lcl} 2a^{h-1} & \leq n \leq & b^h \\ \log_b n & \leq h \leq & 1 + \log_a \frac{n}{2} \end{array}$$

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).

Schnell herleiten kann man auch, daß die Speicherausnutzung bei einem (a,b)-Baum entsprechend mindestens $(2a-1)/(2b-1)$ beträgt, z.B. für einen (2,4)-Baum 3/7.

Standard-Operationen: Die drei Standard-Operationen setzen sich genau wie bei den AVL-Bäumen aus den einfachen Operationen und falls nötig bestimmten Rebalancierungs-Funktionen zusammen. 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, als auch für den Average case bei $O(\log n)$ mit $n = |S|$, denn wie üblich hängt sie von der Höhe des Baumes ab.

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* verwendet man eher kleine Bäume, da die Zugriffe hier schnell erfolgen können und es dann sinnvoll ist, viele kleine Zugriffe zu haben. Untersuchungen haben ergeben, daß sich dann am besten *(2,4)-Bäume* und *(2,3)-Bäume* (B-Baum) eignen.

- 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 → hohe Ordnung). Denn es gilt, daß die Anzahl der Zugriffe proportional zur Höhe des Baumes wächst.

  Ein kleines 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 werden und am Schwanz eingefügt. 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.

Eine Priority Queue besteht dann aus einer nach Priorität geordneten Menge S, und den folgenden

- *primären Operationen*
  - Insert(x, S),
  - DeleteMin(x, S): Entfernen des Elements mit der höchsten Priorität,
- und den erweiterten Operationen
  - Initialize(S)
  - ReplaceMin(x, S)
  - und u.U. Delete(x, S), Search(x, S)

Zur Umsetzung verwendet man die zwei Datenstrukturen

- AVL-Baum und
- Heap,

also einen partiell geordneten, links-vollständigen Binärbaum, bei dem in jedem Knoten das Minimum (Maximum) des jeweiligen Teilbaumes steht.

Damit lassen sich dann gut folgende zwei Standardoperationen umsetzen:

- **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. Dazu dient eine 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.

Dieses gute Abschneiden betrachtend eröffnet sich natürlich die Frage, warum die Heap-Datenstruktur nicht zur Mengendarstelllung mit den drei Standard-Operationen *Search*, *Insert* und *Delete* verwendet wird?

Bei dem Versuch die *Search-Operation* in der Heap-Struktur umzusetzen zeigt sich schnell, daß man dabei auf größere Probleme stößt.

4 Graphen

4.1 Grundlagen

Viele reale Probleme liegen in Form von Graphen, oder der speziellen Form eines Netzwerkes vor, z.B.

- Jegliche Art von *Verbindungen* zwischen verschiedenen Punkten in Netzwerken, wie z.B.
  - Straßennetz,
  - Eisenbahnnetz,
  - Kanalisation oder
  - Telefonnetz.
- Auch die Verdrahtung elektronischer Schaltungen läßt sich naheliegenderweise als Graph darstellen.
- Genauso Beziehungen zwischen Personen, wie
  - Person A kennt Person B
  - Person A spielt gegen Person B

Generell eignen sich Graphen also besonders gut, wenn es darum geht Beziehungen zwischen verschiedenen Elementen einer Menge darzustellen.
Hat man anhand eines Problems einen Graphen erstellt, dann ergeben sich *spezielle Aufgaben und Fragen.*

- Existiert eine Verbindung zwischen A und B?
- Wie lautet die kürzeste Verbindung von A nach B?
- Wo liegt ein minimaler Spannbaum im Netz?
- Wie plane ich eine optimale Rundreise? (Traveling Salesman Problem)

4.1.1 Begriffe

- Ein *Knoten* ist ein Objekt aus der darzustellenden Menge, dessen Beziehungen zu anderen Knoten durch die *Kanten* dargestellt werden.
- Sind die Kanten nur in eine vorgegebene Richtung nutzbar, so spricht man von einem *gerichteten* Graphen, andernfalls von einem *ungerichteten.*
- Haben die Kanten unterschiedliche Gewichte, so handelt es sich um einen *gewichteten* Graphen. Die Entfernungen in einem Straßennetz könnten z.B. eine Gewichtung sein, oder auch der Durchfluß eines Kanals in einer Kanalisation. Haben die Kanten alle einen Einheitswert, so heißt der Graph *ungewichtet.*

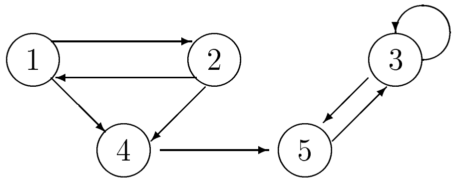


Abbildung 25: Beispiel Graph G_1

4.2 Definitionen und Darstellungen

Bevor wir uns den Graphen-Anwendungen widmen, bedarf es einiger Definitionen.

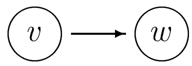
Definition *Gerichteter Graph*: Ein *gerichteter Graph* (**Di**rected **Graph**) ist ein Paar $G = (V, E)$ mit einer endlichen, nichtleeren Menge V, deren Elemente Knoten (Nodes, **V**ertices) heißen, und einer Menge $E \subseteq V \times V$, deren Elemente Kanten (**E**dges, Arcs) heißen, und die nicht symmetrisch sind.

Dann ist $|V|$ die Knotenanzahl und $|E| \leq |V|^2$ die Kantenanzahl.

Läßt man nicht alle Kanten im Graphen zu, z.B. weder Kanten mit gleichem Anfangs- und Endknoten, noch Kanten, die zwei Knoten verbinden, die in entgegengesetzter Richtung schon verbunden sind, dann erhält man entsprechend engere Grenzen ($|E| = \dfrac{|V|(|V|-1)}{2}$).

Meist werden die Knoten einfach durchnumeriert: $i = 1, 2, \ldots, |V|$.

- Die Kante $e : v \to w$ stellen wir graphisch als Pfeil vom *Anfangs-* zum *Endknoten* dar

  v und w heißen dann *adjazent*.

- Weiter verwendet man auch folgende Synonyme. Sind v und w adjazent, dann nennt man sie *Nachbarn*. Der Anfangsknoten v heißt auch *Vorgänger* seines *Nachfolgers* w.

- Als *Grad* bezeichnet man die Anzahl der ein- bzw. ausgehenden Kanten eines Knotens.

- 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 in einem Graphen.

  Die *Länge eines Pfades* ist die Anzahl der Kanten auf diesem Pfad. Sind alle Knoten auf dem Pfad mit Ausnahme von $v_1 = v_n$ paarweise verschieden sind, so heißt der Pfad *einfach*.

- Ein *Zyklus* ist ein einfacher Pfad mit $v_1 = v_n$ und Länge > 0.

- Ein *Teilgraph* eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E$.

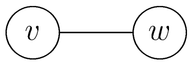
- Führt man Markierungen oder Beschriftungen für Kanten und Knoten ein, dann spricht man von einem gewichteten Graphen. Der Wert einer Kante wird *Kosten* $c[v, w]$ oder $cost(v, w)$ genannt.

  Sie können z.B. die Reisezeit oder -kosten zwischen zwei Punkten ausdrücken, die Entfernung von A nach B oder die Kapazität eines Kanals etc.

Definition *Ungerichteter Graph*: Ein ungerichteter Graph definiert sich ähnlich einem gerichteten, jedoch ist die Relation E symmetrisch, d.h.

$$(v, w) \in E \Rightarrow (w, v) \in E$$

Dementsprechend benutzt man eine graphische Darstellung ohne Pfeil:



Die Begriffe sind analog zu denen für gerichtete Graphen, jedoch sind bisweilen Modifikationen erforderlich, z.B. muß ein Zyklus mindestens drei Knoten haben.

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.

- Bei der *knoten-orientierten Darstellung* identifiziert man zur Vereinfachung der Adressierung die Knoten mit den natürlichen Zahlen.

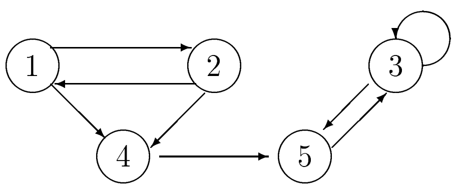
  Sei daher $V = \{1, 2, \ldots, n\}$ mit $n := |V|$.

  Dann bieten sich die folgenden Darstellungsformen an:

  – Die **Adjazenzmatrix** (markierte Adjazenzmatrix) ist ein zweidimensionales Feld A, in dem mittels $(i, j) \in V^2$ jede Beziehung zwischen zwei Knoten adressiert werden kann. Im Falle eines ungewichteten Graphen ist A eine boolesche Matrix mit:

  $$A_{ij} = \begin{cases} true & \text{falls } (i,j) \in E \\ false & \text{sonst} \end{cases}$$

  Eine solche Matrix A_{ij} läßt sich als Array $A[i, j]$ darstellen. Dann stellt sich obiger Graph G_1 wie folgt dar:



von \ nach	1	2	3	4	5
1		1		1	
2	1			1	
3			1		1
4					1
5			1		

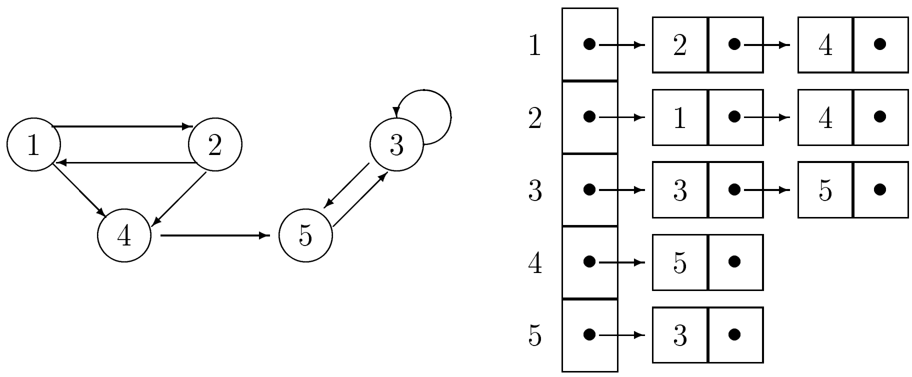
Der Vorteil dieser Darstellung liegt darin, daß die Entscheidung ob $(i, j) \in E$ ist in Zeit $O(1)$ getroffen werden kann. Der Nachteil liegt jedoch bei „dünnen" Graphen, also solchen für die $|E| \ll |V|^2$ gilt, denn der Platzbedarf ist mit $O(|V|^2)$ konstant.

Außerdem ist die Initialisierung mit einer Zeitkomplexität $O(|V|^2)$ ebenfalls ineffizient. Jedoch läßt sich hier ähnlich der erweiterten Bitvektor-Darstellung (vgl. Abschnitt 3.2.3) Abhilfe schaffen.

Trägt man statt der booleschen Werte Kanten-Markierungen entsprechend einer Kostenfunktion ein, so läßt sich diese Darstellung einfach für gewichtete Graphen modifizieren.

– Für die **Adjazenzliste** werden die Knoten selbst wahlweise in einer Liste oder einem Array gespeichert und für jeden Knoten wird eine Liste seiner Nachfolger- oder Nachbarknoten gespeichert.

  Für den Graphen G_1 stellt sich die Adjazenzliste dann wie folgt dar



  Modifiziert man die Abspeicherung dahingehend, daß man die Vorgänger statt der Nachfolger abspeichert, so erhält man eine „invertierte" Adjazenzliste.

  Der Vorteil der Adjazenzlisten-Darstellung ist der geringe Platzbedarf $O(|V| + |E|)$, der mit einer Zeitkomplexität $O(|E|/|V|)$ im Average Case für die Entscheidung, ob zwei Knoten adjazent sind erkauft wird.

- Für die *kanten-orientierte Darstellung* ist ein eigener Index i erforderlich, über den die Kanten adressiert werden können.

  Für jede Kante wird dann der Anfangs- und Endknoten gespeichert und gegebenenfalls die Kosten (Markierung).

Die Art der Speicherung kann zwischen einer statischen Array- und einer dynamsichen Listendarstellung variieren.

Je nach der Anzahl der Kanten in Relation zur Anzahl der Knoten unterscheidet man verschieden dichte Graphen:

- Für einen *vollständigen Graphen* (complete graph) gilt

$$|E| = |V|^2$$

oder, falls bei einem gerichteten Graphen keine Rückverbindungen zugelassen sind

$$|E| = \frac{|V|(|V|-1)}{2}$$

- Als *dichten Graphen* (dense graph) bezeichnet man einen Graphen, falls für das Kanten-zu-Knoten-Verhältnis gilt

$$|E| \simeq |V|^2$$

- Ein *dünner Graph* (sparse graph) ist ein Graph mit

$$|E| \ll |V|^2$$

Die Dichte eines Graphen ist Hauptkriterium bei der Auswahl einer Darstellungsform. Ziel ist ein Optimum von Speicherplatz- und Zugriffszeit-Effizienz.

Programme zu den Darstellungen

Zunächst die Implementation der Adjazenzmatrix-Darstellung für einen ungewichteten Digraphen. Das Programm liest einen solchen von der Tastatur ein, und speichert ihn in besagter Form.

Dabei ist 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;
```

Sind die Knoten nicht einfach durchnumeriert, so muß man ihnen mittels einer Funktion `index(name)` , die sich mittels einer Hashfunktion realisieren ließe, einen Index zuweisen.

Nun die Implementation der Adjazenzliste. In dieser werden die Knoten selbst in einem Array verwaltet und bestehen aus Pointern (`link`), auf ihre Nachfolger.

```
program adjlist (input, output);
  const maxV = 1000;
  type  link = ↑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);
  new(z); 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);
      new(t); t↑.v := x; t↑.next := adj[y]; adj[y] := t;
      new(t); t↑.v := y; t↑.next := adj[x]; adj[x] := t;
    end
end;
```

4.3 Graph-Durchlauf

Definition *Expansion*: 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 $X_G(v_1), ..., X_G(v_k)$.

Anmerkungen:

- Die Knoten des Graphen G kommen in der Regel mehrfach im Baum $X_G(v)$ vor.
- Hat der Graph Zyklen, so ist $X_G(v)$ unendlich.
- Der Baum $X_G(v)$ stellt die Menge aller Pfade dar, die in G von v ausgehen.

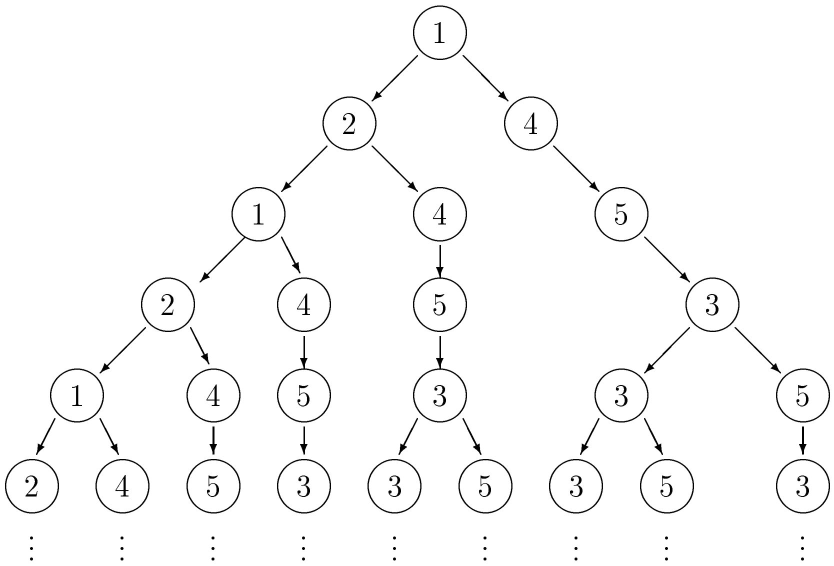
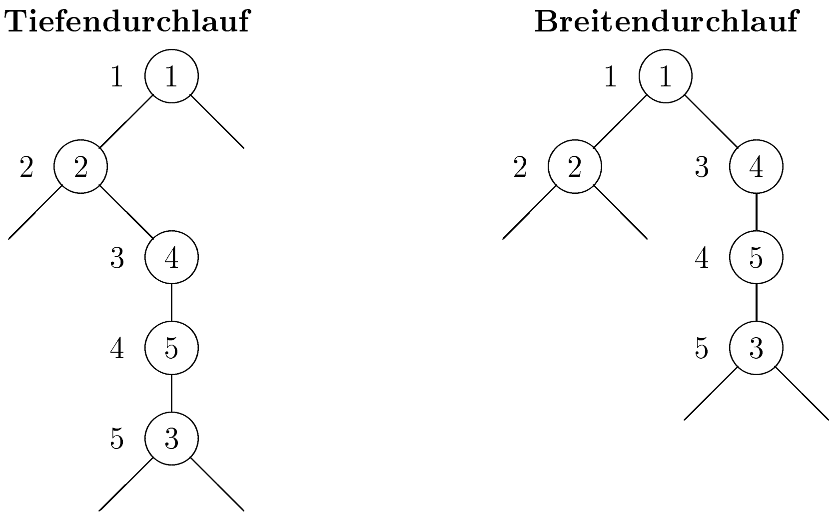


Abbildung 26: Beispiel Graph G_1 (s.Seite 130) in seiner Expansion $X_G(v)$ mit $v = 1$.

Hat man nun eine Expansion eines Graphen, so stellt sich der **Tiefendurchlauf** (depth first) durch den Graphen gerade als *Preorder-Traversierung* der Expansion von G dar. Zur Abarbeitung legt man die Randknoten auf einem Stack ab. Trifft man bei der Abarbeitung auf einen bereits besuchten Knoten, so bricht man die weitere Verfolgung dieses Pfades ab, und wendet sich dem nächsten Knoten auf dem Stack zu.

Der **Breitendurchlauf** (breadth first) durch G hat kein direktes Traversierungs-Korrelat, wird aber als Level Order-Traversierung bezeichnet. Es müssen nämlich alle Knoten ebenenweise durchlaufen werden. Dazu besucht man einfach der Reihe nach alle in einer Queue verwalteten Randknoten.

Der Unterschied zum Baumdurchlauf besteht also im wesentlichen darin, die schon besuchten Knoten als „visited“ zu markieren, da die Knoten in der Expansion ja mehrfach vorkommen können.



Programm für den Graph-Durchlauf

Wir gehen davon aus, daß die Knoten des Graphen alle mit den Zahlen $i = 1, \ldots, n$ mit $n := |V|$ indiziert sind. In einem Array *val* wird ein Wert für jeden Knoten gespeichert, der seinen Status kennzeichnet.

Zunächst werden alle Knoten mit einem Wert $val[v] = 0$ initialisiert. Dies sind die *ungesehenen Knoten* (unseen vertices).

Knoten die schon besucht und abgearbeitet worden sind werden in den „Suchbaum“ aufgenommen und erhalten einen Wert entsprechend der Reihenfolge ihrer Aufnahme: $val[v] := id > 0$. Der Wert *id* wird im Programm hochgezählt. Die Knoten im Suchbaum heißen *Baumknoten*.

Die Knoten, die über eine Kante mit einem Baumknoten verbunden sind heißen *Randknoten* und müssen zusätzlich mit $val[v] = -1$ gekennzeichnet werden, wenn das Programm nicht rekursiv, sondern iterativ mittels eines Stacks oder einer Queue arbeitet.

Die Werte im Hilfsarray *val* haben also folgende Bedeutung

$$val[v] = \begin{cases} 0 & \text{Knoten } v \text{ ist ungesehen} \\ -1 & \text{Knoten } v \text{ Randknoten auf dem Stack/Queue} \\ id > 0 & \text{Knoten } v \text{ abgearbeitet} \end{cases}$$

Unabhängig von der Art des Durchlaufs (depth first oder breadth first) läßt sich dann folgender allgemeiner Algorithmus formulieren.

1. Initialisiere zu Beginn alle Knoten als ungesehen: $val(v) := 0$.
2. **REPEAT**
   - Wähle entsprechend dem Auswahlkriterium (depth first, breadth first, priority first) einen Randknoten x,
   - füge x dem Suchbaum hinzu ($val[x] := id$),
   - bearbeite x entsprechend der Durchlauf-Art, d.h. füge alle ungesehenen Nachbarknoten von x den Randknoten hinzu.

   UNTIL alle Knoten in den Suchbaum aufgenommen.

Setzt man dieses Vorgehen in ein Programm um, so erhält man für den Graph-Durchlauf folgendes Programm-Gerüst.

```
procedure GraphDurchlauf;
  var id, k : integer;
      val : array[1..maxV] of integer;
begin
  id := 0;
  (* s.Kommentar *)
  for k := 1 to V do val[k] := 0;
```

```
  for k := 1 to V do
    if val[k] = 0 then visit(k)
end;
```

Kommentar: Benutzt man einen Stack oder eine Queue, so muß man diese an der kommentierten Stelle mittels `StackInit` bzw. `QueueInit` initialisieren.

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);
    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 Adjazenzliste*: Hier benutzt man einen **Stack**, um die Randknoten zu verwalten. Damit ein Randknoten nicht mehrmals auf den Stack geschrieben wird, muß er mit $val(i) := -1$ markiert werden. In der rufenden Prozedur (GraphDurchlauf) muß der Stack an besagter Stelle initialisiert werden (`StackInit`). Die Prozedur *visit(k)* präsentiert sich dann wie folgt:

```
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
          then begin
            push(t↑.v); val[t↑.v] := -1
          end;
        t := t↑.next
      end
  until stackempty
end;
```

Beachte: Die Reihenfolge der Abarbeitung der Randknoten ist nicht exakt dieselbe wie im rekursiven Programm.

- *Iterativer Breitendurchlauf mit Adjazenzliste*: Um den Breitendurchlauf zu realisieren, muß man nur die Reihenfolge der Abarbeitung der Randknoten ändern. Eine dahingehende Änderung liegt vor, wenn man die Randknoten nicht auf einem Stack sondern mit einer **Queue** verwaltet.

  Also muß man in der Prozedur *visit(k)* nur die Stack-Routinen *pop* durch *get*, *push* durch *put* und *stackempty* durch *queueempty* ersetzen, und an kommentierter Stelle eine Queue initialisieren (`QueueInit`).

  Somit ergibt sich die Prozedur wie folgt:

```
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;
```

Komplexität

Die Zeitkomplexität ist nicht abhängig von der Art des Durchlaufes, sondern nur von der der Darstellung:

- Bei Verwendung einer Adjazenzliste wird jeder Knoten und jede Kante bearbeitet: $O(|E| + |V|)$.
- Bei Verwendung einer Adjazenzmatrix wird jedes Element der Matrix bearbeitet: $O(|V|^2)$.

4.4 Kürzeste Wege

Zu den wichtigen Verfahren auf Graphen gehören diejenigen zur Auffindung kürzester Wege von A nach B, von einem Knoten zu allen anderen oder von allen zu allen anderen.

4.4.1 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus berechnet alle kürzesten Wege von einem Startknoten v_0 (source) zu allen anderen, daher auch Single Source Best Path, und wurde von Dijkstra bereits im Jahre 1959 vorgestellt.

Es ist ein sehr elegantes und effizientes Verfahren, und nachdem die Originalarbeit nur drei Seiten umfaßte, wurde schon ein Vielfaches darüber geschrieben.

Gegeben sei ein gerichteter Graph G mit der Knotenmenge V und folgender Bewertungsfunktion $c : E \to \mathbb{R}_0^+$

$$c[v,w] \quad = \quad \begin{cases} \geq 0 & \text{Kante von } v \text{ nach } w \text{ existiert mit Kosten } c[v,w], \\ \infty & \text{Kante von } v \text{ nach } w \text{ existiert nicht.} \end{cases}$$

Gesucht ist nun ein Pfad von gegebenem Startknoten v_0 zu einem Knoten w mit minimalen Gesamtkosten. Die Gesamtkosten setzen sich als Summe der Einzelkosten der Kanten auf dem Pfad zusammen.

Es wird sich aber zeigen, daß die Verallgemeinerung, vom Startknoten v_0 die kürzesten Wege zu allen Knoten zu finden, auch nicht mehr Aufwand erfordert.

Dem Dijkstra-Algorithmus liegt das **Optimalitätsprinzip** zugrunde [Ottmann, Seite 620]. Es besagt, aufgrund der Additivität der Gesamtkosten, daß für jeden besten Pfad $p = (v_0, v_1, \ldots, v_k)$ von v_0 nach v_k auch jeder Teilpfad $p' = (v_i, \ldots, v_j)$ von p mit $0 \leq i < j \leq k$, ein bester Pfad von v_i nach v_j ist.

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 Eindeutigkeit muß nicht gegeben sein und wird auch nicht behauptet. Die Additivität der Kostenfunktion ist wesentlich für das Optimalitätsprinzip.

Arbeitsweise des Algorithmus

Der Dijkstra-Algorithmus geht nun in umgekehrter Reihenfolge vor und baut immer länger werdende kürzeste Wege aus den schon bekannten kürzesten Wegen auf. Das entspricht etwa einer äquidistanten Welle um den Startpunkt, während derer Ausbreitung jeder Weg zu einem gerade erreichten Knoten vermerkt wird. (Vgl. Abb. 28).

Dabei unterscheidet der Algorithmus, ähnlich den Graph-Durchlauf-Algorithmen zwischen drei Arten von Knoten, die in den folgenden Mengen zusammengefaßt werden:

- S_k, d.h. den k Knoten, zu denen bereits ein kürzester Weg bekannt ist.
- Den Randknoten, also denjenigen, die von S_k aus erreicht werden können. $D(S_k, v)$ beschreibt dann die Gesamtkosten des besten Pfades aus S_k nach v. Also gilt für die Randknoten
$$\{v \in V : D(S_k, v) < \infty\} \setminus S_k$$
- und den unerreichten Knoten, die nicht direkt von S_k aus errreicht werden können:
$$\{v \in V : D(S_k, v) = \infty\}$$

Zu S_{k-1} wird im k-ten Schritt derjenige Randknoten w_k hinzugefügt, der die geringsten Gesamtkosten $D(S_{k-1}, w_k)$ für den Pfad vom Startknoten hat.

Daraufhin wird dann die Menge der Randknoten und ihre jeweiligen Gesamtkosten neu berechnet.
$$D(S_k, v) := \min\{D(S_{k-1}, v), D(S_{k-1}, w_k) + c[w_k, v]\}$$

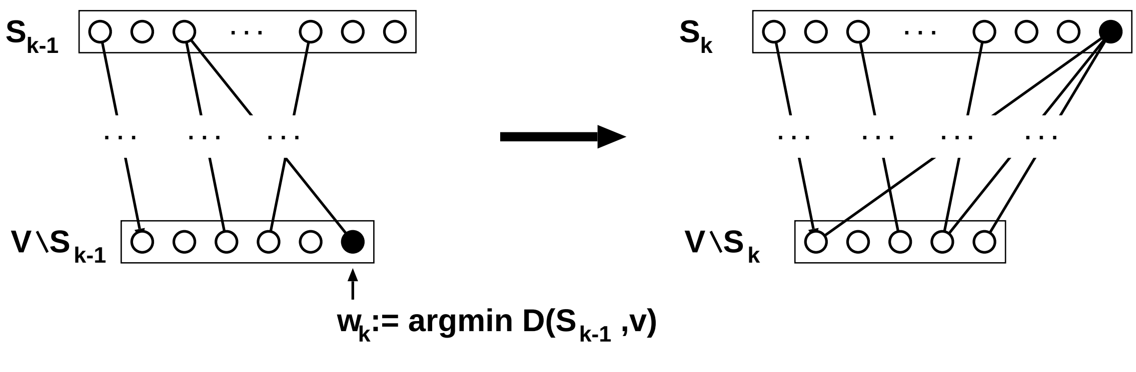


Abbildung 27: Ein Schritt des Dijkstra Algorithmus

Die Randknoten müssen nicht explizit gespeichert werden, da sie sich ja unmittelbar aus der Kostenfuntkion D ergeben.

Eine Verallgemeinerung dieses Konzeptes führt zu dem sogennnaten A^*-Algorithmus und verwandten heuristischen Suchverfahren, wie sie in der Künstlichen Intelligenz verwendet werden. Als mathematischer Algorithmus läßt sich unser Schema folgendermaßen formulieren

(1) **Initialization**
(2) $\quad S_0 := \{v_0\}$;
(3) $\quad D(S_0, v) := c[v_0, v] \;\; \forall\, v \in V \setminus S_0$
(4) **For each path length** $k = 1$ **to** $|V| - 1$ **do begin**
(5) $\quad w_k := \arg\min\{D(S_{k-1}, w\}$ `mit` $w \in V \setminus S_{k-1}$;
(6) $\quad S_k := S_{k-1} \cup \{w_k\}$;
(7) $\quad$ **For each vertex** $v \in V \setminus S_k$ **do**
(8) $\qquad D(S_k, v) := min\{D(S_{k-1}, v), D(S_{k-1}, w_k) + c[w_k, v]\}$
(9) **end**;

Einige wichtige Eigenschaften, die man am Algorithmus ablesen kann sind die folgenden:

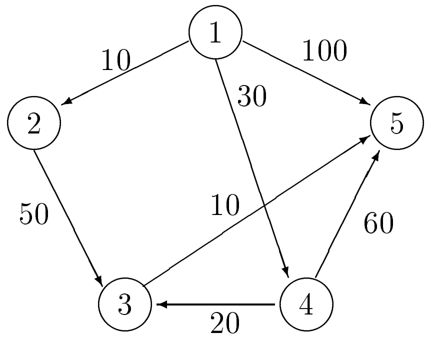
- Der Dijkstra-Algorithmus liefert keine Näherung, sondern garantiert Optimalität.
- Durch Hinzunahme einer Kante mit positiven Kosten können die Gesamtkosten nur wachsen.
- Der längste beste Pfad hat maximal $(|V| - 1)$ Kanten.
- Bei negativen Bewertungen ist der Dijkstra-Algorithmus nicht ohne weiteres anwendbar, denn falls Zyklen mit insgesamt negativer Bewertung existieren, gibt es kein Minimum mehr.

Da die Einzelmengen S_k am Ende uninteressant sind, kann man sie einfach in *einer* Menge S (ohne Index) verwalten.
$V := \{1, \ldots, n\}$
$S :=$ Menge der Knoten, zu denen der kürzeste Weg schon bekannt ist.

```
    procedure Dijkstra;
    begin
(2)   S := {1};
(3)   for i := 2 to n do
        D[i] := c[1,i];
(4)   for i := 1 to n-1 do begin
(5)     choose 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;
```

Beispiel: An folgendem Beispiel wollen wir die Arbeitsweise des Algorithmus demonstrieren [Aho et al. 83].

Der Startknoten v_0 sei der Knoten 1.



k	w	S_k	$D(S_k,2)$	$D(S_k,3)$	$D(S_k,4)$	$DS_k,5)$
0	-	$\{1\}$	10	∞	30	100
1	2	$\{1,2\}$	-	60	30	100
2	4	$\{1,2,4\}$	-	50	-	90
3	3	$\{1,2,3,4\}$	-	-	-	60
4	5	$\{1,2,3,4,5\}$	-	-	-	-

Der beste Pfad zu Knoten 3 ist also z.B. $1 \to 4 \to 3$ mit Länge 50.

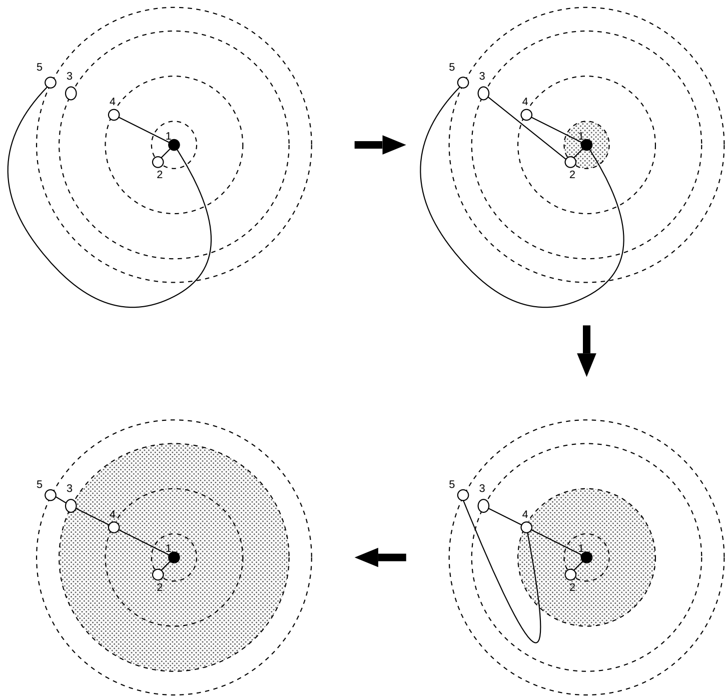


Abbildung 28: Bild der äquidistanten Welle um den Startpunkt für obiges Beispiel

Komplexitätsanalyse

Je nach Implementierung ergeben sich unterschiedliche Komplexitäten für die Laufzeit des Programms. Wir wollen folgende Implementierungen untersuchen:

1. **Adjazenz-Matrix ohne explizite Speicherung der Randknoten**. Dies entspricht obigem Konzept, das Dijkstra folgt.

   - Die Initialisierung in den Zeilen (2) und (3) hat Komplexität $O(|V|)$.
   - Die Schleife in Zeile (4) wird $O(|V|)$-mal durchlaufen und hat in den Zeilen (5) und (7+8) jeweils eine interne Komplexität von $O(|V|)$, also insgesamt $O(|V|^2)$.

   Insgesamt folgt also für diese Implementierung eine Komplexität

   $$O(|V|^2)$$

   Für *dichte Graphen* mit $|E| \cong |V|^2$ ist diese Implementierung also sehr effizient.

2. **Adjazenz-Liste mit Speicherung der Randknoten in einem Heap (Priority Queue)**. Für *dünnere Graphen* ($|E| \ll |V|^2$) bietet sich generell die Speicherung des Graphen in Adjazenzlisten mit den jeweiligen Kostenangaben an. Verwaltet man dann die Randknoten explizit in einem Heap (Priority Queue), dann kann man für den eigentlichen Berechnungsschritt (ab Zeile (4))folgende Komplexität ermitteln:

   Die Reduktion des Heaps um das jeweils kleinste Element bis alle Elemente gewählt wurden entspricht HeapSort und hat eine entsprechende Komplexität $O(|V| \log |V|)$. Die Aktualisierung des Heaps von außen durch hinzufügen der neu hinzugekommenen Randknoten muß prinzipiell für jede Kante durchgeführt werden und bedarf jeweils einer Komplexität $O(\log |V|)$, also insgesamt $O(|E| \log |V|)$. Es ergibt sich also eine Komplexität von

   $$O((|E| + |V|) \cdot \log |V|)$$

   Da im allgemeinen $|E| \geq |V|$ gilt also $O(|E| \log |V|)$. Das bedeutet für dünne Graphen eine bessere Effizienz als $O(|V|^2)$, kann aber für $|E| \cong |V|^2$ auch $O(|V|^2 \cdot \log |V|)$ bedeuten.

3. **Speicherung der Randknoten in einem „Fibonacci-Heap“**. Verwaltet man die Randknoten in einem Fibonacci-Heap mit seinen besonderen Eigenschaften, auf die wir hier nicht näher eingehen wollen, so ergibt sich eine Zeitkomplexität

   $$O(|E| + |V| \cdot \log |V|),$$

   die garantiert genausogut oder besser ist als das bessere der beiden obigen Verfahren.

4.4.2 All Pairs Best Path

Im Jahre 1962 stellte Floyd einen Algorithmus vor, der sich dem Problem widmet, die kürzesten paarweisen Pfade zwischen **allen** Knoten eines Graphen zu finden.

Prinzipiell denkbar ist es, den Dijkstra-Algorithmus für alle Startknoten anzuwenden. Daraus ergibt sich eine Komplexität von $O(|V| \cdot |V|^2) = O(|V|^3)$.

Ein anderer Ansatz, der direkt von der *Dynamischen Programmierung* herrührt, ist der von Floyd:

Sei $G = (V, E)$ ein gerichteter Graph mit $V = \{1, 2, ..., n\}$ und nicht-negativer Bewertung

$$c[v, w] = \begin{cases} \geq 0 & \text{, falls Kante } v \rightarrow w \text{ existiert} \\ \infty & \text{, sonst} \end{cases}$$

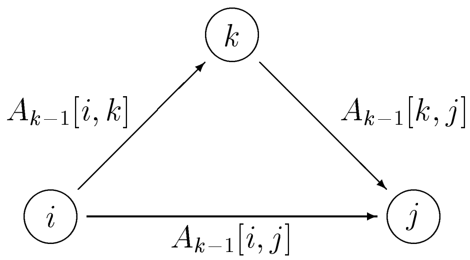
Wir definieren zunächst eine $|V| \times |V|$-Matrix A_k mit den Elementen:

$$A_k[i, j] := \quad \begin{array}{l} \text{minimale Kosten, um von Knoten } i \text{ zu Knoten } j \text{ zu kommen} \\ \text{und der Pfad nur aus Knoten} \in \{1, ..., k\} \text{ besteht} \end{array}$$

Aufgrund der Definition gilt folgende Rekursionsgleichung (DP-Gleichung):

$$A_k[i, j] := \min\{A_{k-1}[i, j], A_{k-1}[i, k] + A_{k-1}[k, j]\}$$

Folgende Abbildung veranschaulicht die Gleichung:



Vereinfachung: Da sich kein Element $A_k[i, j]$ im Vergleich zur vorhergehenden Matrix $A_{k-1}[i, j]$ ändert, wenn i oder j gleich k sind ($A_k[i, k] = A_{k-1}[i, k]$ und $A_k[k, j] = A_{k-1}[k, j]$), benötigt man keine n Matrizen $A_k[i, j]$ mit $k = 1, ..., n$, sondern es genügt eine einzige Matrix $A[i, j]$.

Die Matrix $A[i, j]$ wird mit $c[i, j] \quad \forall \quad i, j \in \{1, \ldots, |V|\}$ initialisiert.

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, und 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;
```

Aus der Programm-Struktur (drei verschachtelte Schleifen über i, j und k) ergibt sich eine Zeitkomplexität $O(|V|^3)$.

Die Matrizen erfordern eine Platzkomplexität $O(|V|^2)$.

Warshall-Algorithmus

Ist der gegebene Graph ein unbewerteter Graph, so handelt es sich um den von Warshall 1962 veröffentlichten Spezialfall. Dann berechnet der Floyd-Algorithmus nämlich, ob überhaupt ein Pfad für jedes Knotenpaar existiert.

Und somit beschreibt die Matrix A nach dieser Berechnung die *transitive Hülle* des von C beschriebenen Graphen.

Definition *Transitive Hülle*: 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:

$$(v, w) \in \bar{E} \Longleftrightarrow \text{es gibt in } G \text{ einen Pfad von } v \text{ nach } w.$$

Gilt also für die Kostenfunktion $c : E \to \{true, false\}$

$$c[v, w] = \begin{cases} \textbf{true}, \text{ falls Kante } v \to w \text{ existiert} \\ \textbf{false}, \text{ sonst.} \end{cases}$$

dann ergibt die folgende Modifikation des Floyd-Algorithmus den Algorithmus von

Warshall. Entsprechend wird statt der Real/Integer-Kosten-Matrix eine Boolesche Matrix verwendet.
Das Element $A[i, j]$ der booleschen Matrix gibt 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 Minimum Spanning Tree (MST, Prim 1957)

Wir wollen minimale Spannbäume in ungerichteten Graphen betrachten, und müssen daher noch einige Definitionen von Begriffen in ungerichteten Graphen vornehmen.
Definitionen *für ungerichtete Graphen*: Gegeben sei ein ungerichteter Graph $G = (V, E)$.

- *Verbunden*: 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. Dann gilt auch $G =$ transitive Hülle $\bar{G}$.
- *Baum*: Ist der Graph G verbunden (zusammenhängend) und azyklisch, so heißt er freier Baum.
  Wählt man einen Knoten daraus als Wurzel, so erhält man einen (normalen) Baum.
- *Spannbaum*: Ist G ein freier Baum, dann ist ein Spannbaum ein Teilbaum von G, falls er alle Knoten von G enthält, und seine Kanten eine Teilmenge von E bilden.
  Ist G bewertet, dann ergeben sich die Kosten eines Spannbaums von G als die Summe der Kosten aller seiner Kanten.

Definition *Minimaler Spannbaum (MST)*: Sei G ein ungerichteter bewerteter Graph. Dann heißt ein Teilbaum von G, der alle Knoten verbindet und für den die Summe der Kosten aller seiner Kanten minimal ist, Minimaler Spannbaum (Minimum Spanning Tree).

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), ähnlich dem Dijkstra-Algorithmus und dem Kruskal-Algorithmus (1957), eine Eigenschaft zunutze, die im wesentlichen auf der additven Zerlegbarkeit der Kostenfunktion beruht.

MST property

Sei $G = (V, E)$ ein verbundener, ungerichteter Graph mit der Bewertungsfunktion $c[v, w] \geq 0$ für eine Kante (v, w). Teilt man V in zwei disjunkte Teilmengen $U \subset V$ und $V \setminus U$ und gilt 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.

Denn wäre dies nicht der Fall, dann enthielte der minimale Spannbaum eine Kante (u', v'), die die beiden disjunkten Mengen verbindet, 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.

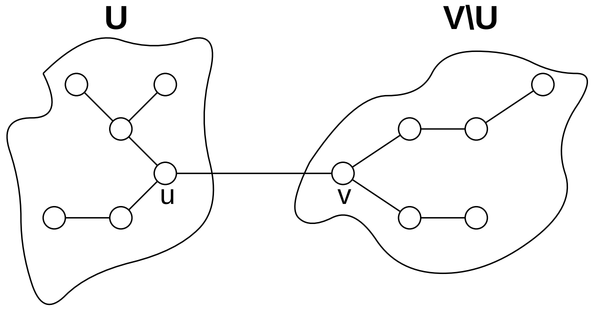


Abbildung 29: Eigenschaft eines Minimalen Spannbaums

Die *MST property* besagt also, daß alle paarweise disjunkten Teilmengen eines minimalen Spannbaumes über eine „minimale" Kante verbunden sind.

Prim-Algorithmus

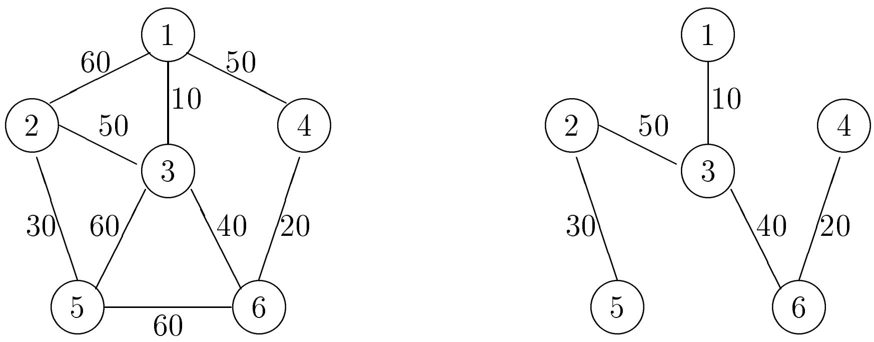
Gegeben sei ein ungerichteter gewichteter Graph $G = (V, E)$. $T = (U, D)$ sei der zu konstruierende Minimalbaum mit der Knotenmenge U, und der Kantenmenge D.

Dann kann man den Prim-Algorithmus folgendermaßen formulieren:

`Initialisierung von` $D := \{\}$ `und` $U := \{v_0\}$`;`
while `(`$U \neq V$`)` **do begin**

`Wähle beste Kante` (u, v) `(mit` $u \in U$ `und` $v \in V$`), die` U `und` $V \setminus U$ `verbindet;`
$U := U \cup \{v\}$; $D := D \cup \{(u, v)\}$
end;

Beispiel: Prim-Algorithmus



Um die beste Kante zwischen U und $V \setminus U$ zu finden, werden zwei parallele Arrays definiert, die bei jedem Durchgang aktualisiert werden, und den Rand verwalten. Sei $i \in V \setminus U$:

- CLOSEST[i] := $\arg\min\{c[u, i] : u \in U\}$ verwaltet die Randknoten selbst, und
- LOWCOST[i] := $c[i, CLOSEST[i]]$ die Kosten um diese von U aus zu erreichen.

Der vorgestellte Code basiert auf dem Prim-Algorithmus aus [Aho et al. 83], wurde jedoch folgendermaßen zur Lauffähigkeit abgeändert:

- Um zwischen erreichten und unerreichten Knoten zu unterscheiden wird ein zusätzliches Array REACHED verwendet. [Aho et al. 83] setzt dafür die LOWCOST-Werte auf ∞. Dies erlaubt jedoch nicht die Unterscheidung zwischen in den MST aufgenommenen Knoten und gar nicht gesichteten Knoten.
- Damit werden dann die **if**-Abfragen ergänzt, um zu garantieren, daß die erreichten Knoten nicht noch einmal untersucht werden.
- Außerdem wurde eine innere Schleife (mit {*} markiert) geändert und läuft anders als bei [Aho et al. 83] auch immer von 2 bis n.

In Programm-Code umgesetzt präsentiert sich der Prim-Algorithmus dann folgendermaßen. Diese Implementation gibt das Ergebnis mittels **writeln** auf der Standard-Ausgabe aus.

```
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 (LOWCOST[j] < min) AND (REACHED[j]=false) 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 (C[k,j] < LOWCOST[j]) AND (REACHED[j]=false) then begin
        LOWCOST[j] := C[k,j];
        CLOSEST[j] := k;
      end
  end
end; { Prim }
```

Komplexität

Am Code erkennt man direkt die zwei verschachtelten Schleifen mit einer Komplexität von jeweils $O(|V|)$, also besitzt der Prim-Algorithmus eine gesamte Zeitkomplexität von

$$O(|V|^2).$$

4.6 Zusammenfassung

Wir haben gesehen wie und in welcher Komplexität die Algorithmen zum Auffinden praxis-relevanter Strukturen in Graphen 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 soll nur kurz erwähnt sein, 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]

Aber auch in der Wissenschaft ist die Frage noch offen, ob man das Traveling Salesman Problem auch in polynomieller Zeit ($O(n^k)$) lösen kann.