# Algorithmen und Datenstrukturen

Herbert Kopp,
Fachhochschule Regensburg
Studiengang Informatik

# Inhalt

**1 Einleitung**

**2 Begriffliche Grundlagen**
- 2.1 Algorithmen
  - 2.1.1 Merkmale von Algorithmen
  - 2.1.2 Komplexitätsmaße
- 2.2 Datentypen
  - 2.2.1 Basisdatentypen
  - 2.2.2 Abstrakte Datentypen
  - 2.2.3 Konstruktive Definition von Datentypen
  - 2.2.4 Datenstrukturen

**3 Einfache strukturierte Datentypen**
- 3.1 Lineare Listen
  - 3.1.1 Allgemeine Eigenschaften linearer Listen
  - 3.1.2 Implementierung linearer Listen als Array
  - 3.1.3 Implementierung linearer Listen durch verkettete Elemente
  - 3.1.4 Spezielle lineare Listen
- 3.2Matrizen

**4 Sortierverfahren**
- 4.1 Grundsätzliche Gesichtspunkte des Sortierens
- 4.2 Elementare Sortiermethoden
  - 4.2.1 Selection Sort
  - 4.2.2 Insertion Sort
  - 4.2.3 Shell Sort
  - 4.2.4 Bubble Sort
- 4.3 Spezielle Sortiermethoden
  - 4.3.1 Quicksort
  - 4.3.2 Heap Sort
  - 4.3.3 Merge Sort
    - 4.3.3.1 Verschmelzen sortierter Teillisten
    - 4.3.3.2 Rekursives 2-Wege-Merge-Sort-Verfahren
    - 4.3.3.3 Bottom up-Merge-Sort-Verfahren
    - 4.3.3.4 Natürliches 2-Wege-Merge-Sort-Verfahren
  - 4.3.4 Radix Sort
    - 4.3.4.1 Radix Exchange Sort
    - 4.3.4.2 Sortieren durch Fachverteilung
- 4.4 Externe Sortierverfahren
  - 4.4.1 Voraussetzungen
  - 4.4.2 Ausgeglichenes 2-Wege Mergesort-Verfahren
  - 4.4.3 Weitere externe Sortierverfahren
- 4.5 Vorsortierung und optimales Sortieren

**5. Elementare Suchverfahren**
- 5.1 Bestimmung des Medianwertes
- 5.2 Sequentielles Suchen
- 5.3 Binäres Suchen
- 5.4 Suchen in Texten
  - 5.5.1 Direkte Muster-Suche
  - 5.5.2 Das Verfahren von Knuth, Morris und Pratt

**6 Bäume**

6.1 Begriffe im Zusammenhang mit Bäumen
6.2 Typen binärer Bäume
6.3 Implementierungstechniken für Bäume
6.4 Operationen mit binären Bäumen
6.5 Durchlaufprinzipien für Bäume
6.6 Fädelung
6.7 Analytische Betrachtungen für Bäume
6.8 Balancierte Binärbäume
6.9 Analyse von AVL-Bäumen
6.10 B-Bäume
6.11 B-Bäume und VSAM-Dateiorganisation

**7 Hash-Verfahren**

7.1 Grundlagen der gestreuten Speicherung
7.2 Hash-Funktionen
7.3 Geschlossene Hash-Verfahren
7.4 Offene Hash-Verfahren
8.4.1 Lineares Sondieren
8.4.2 Quadratisches Sondieren
8.4.3 Coalesced Hashing

**8 Datenkompression**

8.1 Lauflängencodes
8.2 Codes mit variabler Länge
8.2.1 Methodische Grundlagen
8.2.2 Die Huffman-Codierung
8.3 Arithmetische Kompression
8.4 Kompressionsverfahren
9.4.1 Die Squeeze-Kompression
9.4.2 LZW-Kompressionstechniken
8.4.2.1 LZW-Kompression mit fester Codewortlänge
8.4.2.2 LZW-Kompression mit variabler Codewortlänge und Code-Recycling
8.4.2.3 LZSS- und LZH-Kompression
8.5 Kompression in Bilddatei-Formaten
8.5.1 Das PCX-Format
8.5.2 Das BMP-Format
8.5.3 Das TIF-Format
8.5.4 Das JPEG-Format

## Literatur zur Datenorganisation


[1] L. Ammeraal: Programmdesign und Algorithmen in C
Hanser, 1987

Das Buch erhebt nicht den Anspruch auf Vollständigkeit. Es erläutert jedoch alle Algorithmen ausführlich und gibt eine Implementierung in C dazu an. Es werden Verfahren aus den folgenden Gebieten behandelt:
- Sortierverfahren
- Kombinatorische Algorithmen
- Listen, Bäume, Graphen
- Grundlagen der Compilierung

[3] Th.Ottmann,
P.Widmayer: Algorithmen und Datenstrukturen, Spektrum-Verlag

Umfassende, verständliche Darstellung aller grundlegenden AD-Themenbereiche:
- Sortieren, Suchen, Hashverfahren, Bäume, Mengen, geometrische Algorithmen,
- Graphenalgorithmen, Suchen in Texten, Parallele Algorithmen.
- Umfangreiches Aufgabenmaterial.

Eignet sich auch später als Nachschlagewerk.

[4] R.Sedgewick: Algorithmen, Adison & Wesley

Umfangereiche Sammlung von Algorithmen:
- Grundlagen (elementare Datenstrukturen, Bäume, Rekursion, Algorithmen)
- Sortieralgorithmen, Suchalgorithmen
- Verarbeitung von Zeichenfolgen
- Geometrische Algorithmen
- Graphenalgorithmen
- Mathematische Algorithmen (Zufallszahlen, Arithmetik, Gauß, Kurvenapproximation, Integration, Parallele Algorithmen, FFT, dynamische Programmierung, Lineare Programmierung)
- Suchen in Graphen

Eignet sich gut als Nachschlagewerk für verschiedenste Algorithmen. Übungsaufgaben und Fragen nach den Kapiteln

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

Kompakte und gut verständliche Darstellung der wichtigsten Themenbereiche:
- Sortieren
- Rekursive Algorithmen
- Dynamische Datenstrukturen (Zeiger, Listen, Bäume)
- Key-Transformationen

Algorithmen sind in Modula-2 formulier, Aufgaben nach jedem Kapitel.

# 1 Einleitung

Ziel eines jeden Datenverarbeitungsvorgangs ist die Verarbeitung von Daten. Wir betrachten dazu einige DV-Anwendungen:

- eine Recherche in einer Adreßkartei,
- die Prognose des Wetters für den nächsten Tag,
- die Steuerung eines Schweißroboters oder
- die Multiplikation zweier Zahlen.

Zwischen diesen Vorgängen fallen uns zunächst extreme Unterschiede auf:

- Die Anzahl der zu verarbeitenden Datenwerte unterscheidet sich um viele Zehnerpotenzen.
- Der Zeitaufwand für die Vorgänge variiert zwischen Mikrosekunden und Stunden.
- Die Kompliziertheit der Verarbeitungsverfahren ist höchst verschieden.
- Der Entwicklungsaufwand für die Programme kann im Minutenbereich liegen oder mehrere Jahre betragen.

Mindestens ebenso wichtig sind aber die Gemeinsamkeiten zwischen ihnen:

- Die verarbeiteten Daten stellen stets ein Modell realer Gegebenheiten dar: So arbeitet das Wetterprognoseprogramm auf einem Datensatz, der den Atmosphärenzustand über der gesamten Nordhalbkugel repräsentiert.
- Es besteht eine sehr intensive Abhängigkeit zwischen dem algorithmischen Verfahren und der Struktur seiner Daten: Wenn z.B. eine Adreßdatei sequentiell organisiert ist, dann ergeben sich für die Suchmethode und die dafür benötigte Zeit ganz andere Konsequenzen, als bei einer indexsequentiellen Organisation.

Konzepte und Begriffsbildungen im Zusammenhang mit den im Rechner gespeicherten Daten und ihrer Verarbeitung haben sich seit dem Beginn der elektronischen Datenverarbeitung in den fünfziger Jahren ständig fortentwickelt. Dies hatte direkte Konsequenzen für die Hardware-Architektur der Systeme:

- Die ersten elektronischen Rechner besaßen nur einfachste Datentypen, wie z.B. ganze Zahlen oder Zeichenfolgen und ebenso einfache Operationen dafür.
- Die Weiterentwicklung der CISC-Architekturen führte zu hochentwickelten, in der Firmware verankerten Datenstrukturen, z.B. indizierten Arrays, Warteschlangen, Prozeßumgebungen.
- Die RISC-Technologie ging wieder genau den umgekehrten Weg hin zu einfachen Datenstrukturen und elementaren, aber sehr schnellen Operationen dafür.

Eine wesentlich größere Vielfalt an Datenstrukturen entstand auf der Softwareseite:

- Es wurden komplexe statische und dynamische Strukturen entwickelt, z.B. Arrays, Records, Listen, Bäume und Verarbeitungsmethoden dafür.
- Dies hatte wiederum Konsequenzen für die in Programmiersprachen bereitgestellten Sprachmittel und führte auch zu ganz neuen Konzepten, etwa der Datenkapselung, Vererbung und der virtuellen Methoden, wie sie in der objektorientierten Programmierung eine Rolle spielen.

> *Diese Lehrveranstaltung soll einen Fundus an Konzepten und Methoden für die Organisation von Daten und ihre Verarbeitung vorstellen, auf den in konkreten Anwendungsfällen zurückgegriffen werden kann. Unser Augenmerk wollen wir dabei stets auch auf die folgenden Punkte richten.*

- Die Organisationsform der Daten bestimmt maßgeblich die Verfahren, mit denen wir sie verarbeiten können. Sehr häufig steht die Wahl einer geeigneten Datenstruktur am Anfang aller Überlegungen:

  *Versuchen Sie z.B. statt der üblichen schriftlichen Multiplikationsmethode für ganze Zahlen im Dezimalsystem ein Verfahren aufzustellen, das auf einer Zahldarstellung mit römischen Ziffern beruht!*

- Ein Verfahren ist nur dann einsetzbar, wenn gesichert ist, daß es korrekte Ergebnisse liefert:

  *Wenn wir mit einem CAD-System, das mit der üblichen einfachen Gleitpunktgenauigkeit arbeitet, die Geometrie einer 20 m langen Welle konstruieren, können wir nicht erwarten, daß die Genauigkeit der Achse im Mikrometerbereich liegt.*

- Die Speicherplatz- und die Laufzeitkomplexität einer Methode hat entscheidenden Einfluß auf ihre Brauchbarkeit:

  *Wenn die Laufzeit zweier Sortierverfahren mit* $\boldsymbol{O}\left(N^2\right)$ *bzw.* $\boldsymbol{O}\left(N\log(N)\right)$ *von der Länge des zu sortierenden Datenbereichs abhängt, dann müssen wir schon für 100.000 Datensätze statt weniger Sekunden einige Stunden sortieren.*

# 2 Begriffliche Grundlagen

In diesem Abschnitt präzisieren wir die grundlegenden Begriffe, die wir beim Umgang mit Daten im Rechner benötigen.

## 2.1 Algorithmen

Für unseren Zweck können wir einen intuitiven Algorithmusbegriff verwenden, denn wir werden keine prinzipiellen Überlegungen über Algorithmen und Berechenbarkeit anstellen, wie sie in den folgenden Fragestellungen deutlich werden:

- Gibt es zu einem Problem einen Algorithmus, der es löst ?
- Stellt eine Verfahrensbeschreibung einen Algorithmus dar ?
- Gibt es zu einem Problem einen Algorithmus, der es in einer bestimmten Zeit löst ?

### 2.1.1 Merkmale von Algorithmen

Wir betrachten Verfahrensvorschriften zur Lösung eines Problems als Algorithmen, wenn sie die folgenden Merkmale besitzen:

- Sie bestehen aus endlich vielen Verarbeitungsvorschriften.
- Die für den Verfahrenablauf notwendigen Informationen liegen vollständig zu Beginn der Verarbeitung vor.
- Sie spezifizieren eindeutig eine Folge elementarer, eindeutig definierter Operationen.
- Für jeden Satz von Eingabedaten kommt das Verfahren nach endlich vielen Schritten zu einem Ergebnis. Die Anzahl der Schritte muß dabei nicht von vornherein abschätzbar sein.

Dabei unterscheiden wir zwischen dem Algorithmus als Verfahrensvorschrift und der Implementierung dieser Vorschrift, etwa als Programm in einer Programmiersprache.

#### Korrektheit von Algorithmen

Daß ein Algorithmus die gestellte Aufgabe tatsächlich löst, muß stets begründet werden. Dies setzt voraus, daß auch die Problemstellung genügend exakt spezifiziert ist. Formale Korrektheits-beweise sind in der Regel sehr aufwendig und von beschränktem praktischen Wert, so daß wir uns oft auf informelle Begründungen beschränken müssen.

Die bei der Implementierung von Algorithmen auf dem Rechner notwendigen Tests können zwar die Anwe senheit von Fehlern nachweisen, nicht aber die Korrektheit des Verfahrens.

#### Komplexität von Algorithmen

Drei unterschiedliche Hauptkriterien sind zu berücksichtigen, wenn es um die Komplexität von Algorithmen geht:

- die Kompliziertheit der Methode,
- der Speicherplatzbedarf des Verfahrens,
- die Laufzeit des Verfahrens.

Die *Kompliziertheit* des Verfahrens hat vor allem Einfluß auf den Aufwand, der für seine Entwicklung und die Implementierung notwendig ist. Da dieser nur einmalig anfällt, spielt er kaum eine Rolle, wenn das Verfahren sehr oft eingesetzt werden kann oder wenn die gewünschten Ergebnisse anders nicht zu bekommen sind. Es kann aber billiger sein, ein nicht-optimales Verfahren zu benutzen, wenn dieses selten eingesetzt wird oder der mögliche Gewinn nur einige Sekunden Laufzeit ausmacht.

Beispiel: Die Entwicklung eines einfach zu bedienenden und trotzdem leistungsfähigen Textverarbeitungsprogramms rechtfertigt einen hohen Aufwand, während zum Sortieren einer
Namensliste mit 100 Adressen auch ein einfaches, schnell aufzuschreibendes Sortier-
programm völlig ausreicht.

Der *Speicherbedarf* für die Datenbereiche, auf denen ein Verfahren arbeitet, spielt auch im Zeitalter virtueller Adreßräume oft noch eine wichtige Rolle, hauptsächlich wegen der sehr unter-schiedlichen Zugriffszeiten zu Daten im RAM-Speicher einerseits und auf Externspeichern andererseits.

Beispiel: Die Suche eines Eintrags im Telefonbuch der Stadt Regensburg läßt sich wegen der umfangreichen Datenmengen sicher nicht im Arbeitsspeicher abwickeln.

Die *Laufzeit* eines Verfahrens kann ebenfalls der entscheidende Faktor für seine Brauchbarkeit sein, weil in der Regel die Ergebnisse nur dann von Interesse sind, wenn sie rechtzeitig vorliegen.

Beispiel: Eine Wetterprognose für morgen sollte möglichst schon heute vorliegen.

## 2.1.2 Komplexitätsmaße

Die Speicher- und Laufzeitkomplexität eines Algorithmus soll unabhängig von einer speziellen Implementierung Aussagen darüber machen, welchen Aufwand das Verfahren bezüglich Rechen-zeit oder Platzbedarf im Arbeitsspeicher erfordert. Welche Merkmale eines Verfahrens als Komplexitätsmaß benutzt werden, ist sehr unterschiedlich, z.B.

- die Anzahl zeitkritischer Elementaroperationen (Vertauschungen, Multiplikationen, ...)
- Die Anzahl der Durchläufe innerer Schleifen
- die Anzahl von Vergleichsoperationen
- die Anzahl der belegten Speicherplätze, Records usw.

Oft hängt dieser Aufwand ab vom Umfang der Eingabedaten: Das Sortieren einer Adreßliste wird in der Regel von der Anzahl der darin enthaltenen Adressen abhängig sein.

Die exakte, zahlenmäßige Kenntnis dieses Zusammenhangs ist oft unwichtig. Daher beschreibt man die Größenordnung oft durch die ***O***-Notation, die wir hier exemplarisch vorstellen:

- Ein Verfahren benötigt $\boldsymbol{O}(N)$ Schritte, wenn die exakte Schrittzahl $c_1 * N + c_0$ beträgt, wobei es uns auf die Konstanten $c_0$ und $c_1$ nicht ankommt und $N$ z.B. die Anzahl der Eingabedatensätze bezeichnet.
- Ein Verfahren benötigt $\boldsymbol{O}(N^2)$ Schritte, wenn die exakte Schrittzahl $c_2 \cdot * N^2 + c_1 * N + c_0$ beträgt, wobei es uns auf die Konstanten $c_2$, $c_1$ und $c_0$ nicht ankommt.

Es bleiben also konstante Faktoren und alle Polynomterme niedrigerer Ordnung außer Acht. Die Komplexität aller praktisch brauchbaren Verfahren ist höchstens polynomial, d.h. von einer Ordnung $\boldsymbol{O}(N^k)$, Methoden mit exponentieller Komplexität, d.h. mit einem Aufwand $\boldsymbol{O}(N^k)$ sind in der Regel zu aufwendig für einen praktischen Einsatz.

# 2.2 Datentypen

Die folgenden Begriffsbildungen präzisieren unsere bisher noch intuitive Vorstellung von Daten:

**Datentypen**

Ein Datentyp $T = <W,O>$ besteht aus einer Wertemenge $W$ und einer Menge $O$ von Operationen, die auf diese Werte angewandt werden dürfen. Wir unterscheiden dabei zwischen dem abstrakten Datentyp und seiner Implementierung.

Während in der Anfangszeit der Datenverarbeitung das Hauptaugenmerk auf der Verarbeitung lag, ist man inzwischen zu der Erkenntnis gelangt, daß den Daten selbst mindestens die gleiche Bedeutung zukommt und daß diese nur in engem Zusammenhang mit den Operationen verstanden werden können, die wir mit ihnen vornehmen. Mit dem Begriff des Datentyps wollen wir unabhängig von einer speziellen Implementierung festlegen, um welche Art von Daten es sich handelt und was wir mit ihnen tun dürfen:

## 2.2.1 Basisdatentypen

Unter Basisdatentypen verstehen wir diejenigen Datentypen, die oft direkt in der Hardware-Architektur der Rechner verankert sind, also insbesondere die in der folgenden Tabelle angegebenen Typen:

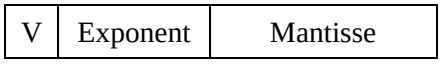
| Wertebereich | Operationen |
|---|---|
| ganze Zahlen | +,–, *, DIV, := |
| reelle Zahlen | +, –, *, / , := |
| logische Werte | AND, OR, XOR, NEG, := |
| Zeichen | := |

**Beispiel:**

Exemplarisch betrachten wir den Basis-Datentyp der reellen Zahlen:

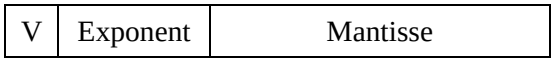
- Die Wertemenge ist die aus der Mathematik bekannte unendlich große Menge reeller Zahlen.
- die Operationsmenge besteht aus den bekannten arithmetischen Operatoren sowie Transportoperationen. Man kann die Vergleichsoperatoren dazuzählen, auch wenn deren Ergebnis in einer anderen Wertemenge liegt.
- Die Implementierung dieses Datentyps in der VAX-Architektur führt zu mehreren *konkreten* Datentypen. Diese besitzen verschiedene Darstellungen im Speicher und die dazu passenden Operationen:

F-Format:

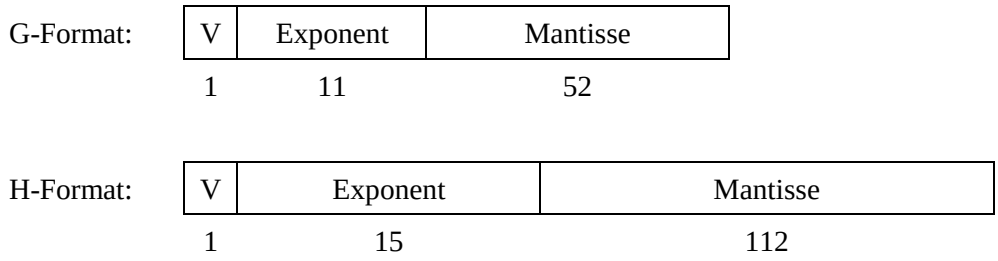



1 8 23

D-Format:




1 8 55




Es ist klar, daß diese Implementierungen weitere charakteristischen Eigenschaften der konkreten Datentypen implizieren:

- die Genauigkeit der Darstellung spielt eine Rolle
- die Wertebereiche sind endlich und verschieden groß
- der Platzbedarf ist unterschiedlich
- die Operationen sind unterschiedlich und benötigen unterschiedlich viel Zeit.

## 2.2.2 Abstrakte Datentypen

Aufbauend auf den Basisdatentypen spezifizieren wir alle "höheren" Datentypen. Auch dabei unterscheiden wir die abstrakten Typenspezifikationen von ihren konkreten Implementierungen. Einige Beispiele für abstrakte Datentypen sind:

- Records
- Mengen
- Listen, Bäume, Schlangen usw.

Zur Spezifikation solcher höheren Datentypen kennt man verschiedene Techniken

#### Axiomatische Definition von Datentypen

Dabei werden Axiome formuliert, die einen Datentyp vollständig und widerspruchsfrei charakterisieren. Das Vorgehen dabei ist ähnlich, wie wir es von der Mathematik her kennen:

Eine Menge heißt dort z.B. eine Gruppe, wenn die auf ihr definierten Operationen den Gruppenaxiomen genügen (Assoziativität, neutrales Element, Invertierbarkeit).

Während diese Methode für theoretische Überlegungen sicher vorteilhaft ist, hat sie für praktische Anwendungen Nachteile:

- Ob die Axiome eines Datentyps der intuitiven Vorstellung davon entsprechen ist nicht immer leicht zu sehen.
- die Vollständigkeit und Widerspruchsfreiheit der Axiome muß nachgewiesen werden.
- aus der axiomatischen Definition läßt sich in der Regel keine Implementierung für den Datentyp ableiten.

#### Beispiel:

Den abstrakten Datentyp ***STACK*** stellen wir uns als lineare Anordnung von Elementen vor, bei der nur am "oberen" Ende Elemente hinzugefügt oder weggenommen werden können. Die axiomatische Spezifikation dafür benutzt folgenden Ansatz:

① Die Wertemenge des Datentyps ***STACK*** besteht aus Wertepaaren, deren Komponenten den folgenden beiden Mengen entstammen:
   - der Menge der im Stack abgelegten Elemente: **ITEMS**
   - der Menge aller Stacks: **STACKS**

② Als Operationen des Datentyps ***STACK*** betrachten wir:
- push: **STACKS x ITEMS** ⇨ **STACKS**
- pop: **STACKS** ⇨ **STACKS x ITEMS**

③ Diese Operationen könnte man durch Axiome der folgenden Art charakterisieren:
- pop (push (S,I)) = (S,I)
- push(pop(S)) = S

Es ist unmittelbar klar, daß es weiterer Präzisierungen bedarf, um den intuitiven Stackbegriff axiomatisch ganz zu fassen. So berücksichtigen die obigen Axiome z.B. noch nicht den Sonderfall eines leeren Stack für die Pop-Operation.

### 2.2.3 Konstruktive Definition von Datentypen

Die konstruktive Methode zur Definition von Datentypen baut auf bereits bekannten Datentypen auf. Es wird angegeben, wie ihre Wertemenge sich aus den Werten elementarer Datentypen zusammensetzt. Die Operationen damit werden aus den Operationen mit elementaren Werten abgeleitet:

**Beispiel:**

Die Punkte in der euklidischen Ebene lassen sich als Datentyp folgendermaßen konstruktiv definieren:

① Die Wertemenge $\boldsymbol{P}=\boldsymbol{R}x\boldsymbol{R}$ besteht aus allen Paaren reeller Zahlen. Diese können wir als Punkte in einem kartesischen Koordinatensystem auffassen.

② Als Operation auf diesem Datentyp definieren wir wie in der Geometrie üblich den euklidischen Abstand zum Nullpunkt: Für einen Punkt $p = (x, y)$ ist seine Distanz $\boldsymbol{D}$ zum Nullpunkt gegeben durch den Ausdruck:

$$\boldsymbol{D}(p) = \sqrt{x^2 + y^2}$$

Mit diesen Angaben können wir direkt eine Implementierung des Datentyps herleiten. Diese könnte in C wie folgt aussehen:

```
struct point { int x;
               int y;
             };
struct point  p1, p2;

double dist(struct point p, q);
{return((double)sqrt((double)pt.x*pt.x + (double)pt.y*pt.y)}
```

### 2.2.4 Datenstrukturen

Den häufig verwendeten Begriff der Datenstruktur wollen wir verstehen als eine konkrete Implementierung eines abstrakten Datentyps.

Datenstrukturen können wir durch die konkrete Implementierung in einer Programmiersprache anschaulich darstellen. Deshalb verwenden wir häufig auch stellvertretend Datenstrukturen, wenn wir eigentlich abstrakte Datentypen untersuchen. Dabei müssen wir jedoch Merkmale des Datentyps strikt unterscheiden von den nur der Implementierung eigentümlichen Charakteristika.

# 3. Einfache strukturierte Datentypen

Wir betrachten nun exemplarisch einige "höhere" Datentypen und Aspekte Ihrer Implementierung, nämlich lineare Listen und Matrizen. Wir werden dabei feststellen, daß die Charakteristika einer konkreten Implementierung oft genauso wichtig sind, wie die Eigenschaften des abstrakten Datentyps.

## 3.1 Lineare Listen

### 3.1.1 Allgemeine Eigenschaften linearer Listen

Der Datentyp *Lineare Liste* baut auf dem abstrakten Konzept der endlichen Folgen von Elementen einer Menge auf. Wir nehmen für das folgende stets eine beliebige, aber fest vorgebene Basismenge ***M*** an. Die Elemente von ***M*** besitzen in der Regel eine innere Struktur, z.B. von der Form:

```
struct Node  {
               int   key;          // Schlüsselfeld
               char  info[35];     // Informationsfeld
               Node  *next;        // Kettungszeiger
             };
```

Zum Datentyp der linearen Listen müssen wir eine Wertemenge und eine Menge von Operationen auf dieser spezifizieren:

#### Wertemenge linearer Listen

Die Wertemenge des Datentyps lineare Liste besteht aus allen endlichen Folgen von Elementen des Basistyps **`Node`**. Wir verwenden für einzelne Listen die Schreibweise

$$\langle a_1, a_2, \cdots, a_n \rangle$$

Mit $\langle\ \rangle$ bezeichnen wir eine leere Liste.

#### Operationen mit linearen Listen

Die folgenden Operationen stellen eine Auswahl typischer Listen-Operationen dar:

- Füge Element *x* auf der Position *p* ein.
- Entferne das Element auf der Position *p*.
- Lies das Element auf Position *p*.
- Suche das Element mit dem Wert *x* (Ergebnis ist seine Position).
- Erzeuge eine neue Liste.
- Löse eine Liste auf.
- Stelle fest, ob eine Liste leer ist.
- Hänge zwei Listen $L_1$ und $L_2$ aneinander an.
- Entferne Element *x* aus der Liste.
- Füge Element *x* in die Liste ein.

Je nach der konkreten Implementierung des Datentyps ergibt sich für diese Operationen ein unterschiedlicher Aufwand. Es ist also für jeden Anwendungsfall speziell zu entscheiden, welche Operationen überhaupt oder vorwiegend gebraucht werden. Daraus kann man dann die geeignetste Implementierung als Datenstruktur ableiten.

## 3.1.2 Implementierung linearer Listen als Array

Eine erste Implementierungstechnik für lineare Listen geht von der sequentiellen Speicherung aller Listenelemente in einem Array aus. Eine mögliche Definition in C dafür ist:

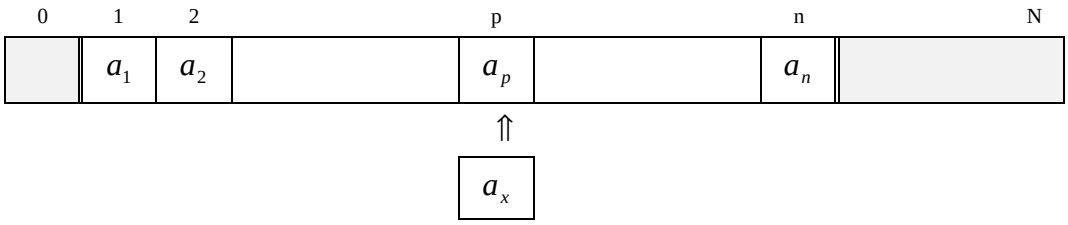
```
#define N 1000
struct  Aliste  {
                  int a[N+1];    // Listen-Feld
                  int lenght;    // Anzahl gültiger Elemente
                };
```

Die wesentlichen Merkmale dieser Implementierumg sind die folgenden:

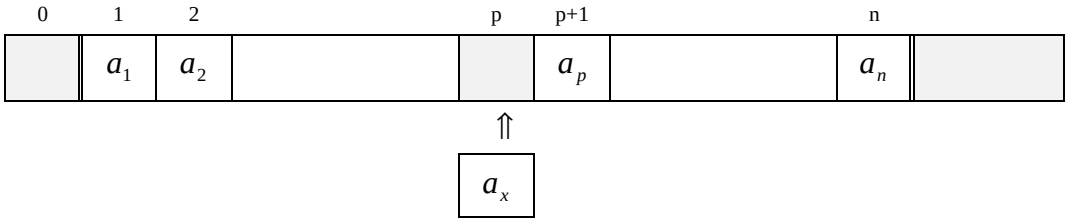
- Das Array-Element mit dem Index *0* dient als Hilfszelle, z.B. zur Aufnahme eines Stopper-Elements bei Suchoperationen.
- Die Reservierung des Speichers für die Liste erfolgt einmalig für die Maximallänge der Liste.
- Die Position eines Elements im Speicher ist direkt abhängig von seiner logischen Stellung in der Liste. Der Zugriff zu Elementen kann daher sehr effizient durch eine Adreßrechnung erfolgen.

**Beispiel:**

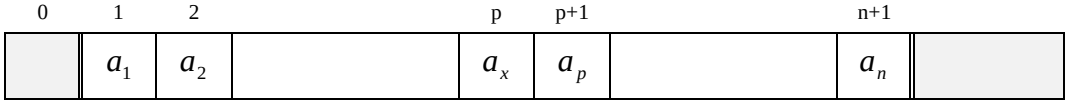
Als typische Operation betrachten wir das Einfügen eines Elementes $a_x$ an der Position p. Die Ausgangssituation ist dabei die folgende, wobei n die Anzahl der gültigen Listenelemente ist:




In einem ersten Schritt verschieben wir vom rechten Ende der Tabelle her die Elemente $a_1, a_2, \cdots, a_n$ um je eine Position nach rechts. An der Stelle *p* entsteht so eine Lücke:




Danach setzen wir das Element $a_x$ auf die freie Position p des Array:




Das Entfernen von Elementen erfolgt nach dem umgekehrten Verfahren.

Für den Aufwand der beiden Operationen

- Einfügen eines Elements $a_x$ an der Position p und
- Entfernen des Elements $a_x$ an der Position p

können wir feststellen:

- Im günstigsten Fall sind keine Verschiebeoperationen nötig.
- Im schlechtesten Fall sind n Verschiebeoperationen nötig.
- Im Mittel sind bei unsortierter Liste $n/2$ Verschiebeoperationen erforderlich. Wenn man jedoch die Liste so sortiert, daß die häufigsten Einfüge/Entferne-Operationen nahe beim Listenende liegen, dann sind im Mittel weniger Operationen erforderlich.

Sequentiell in einem Array gespeicherte Listen besitzen also die folgenden Charakteristika:

- Zugriffe auf eine Position sind effizient durch Adreßrechnung möglich.
- Einfügen/Entfernen/Suchen ist relativ teuer und von der Vorsortierung abhängig.

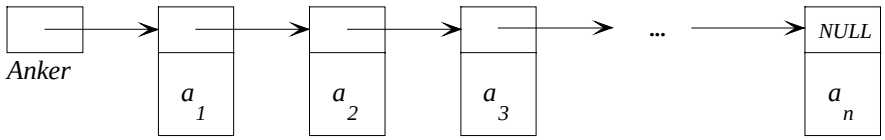
## 3.1.3 Implementierung linearer Listen durch verkettete Elemente

In diesem Abschnitt betrachten wir mehrere Varianten der Implementierung linearer Listen, die alle darauf aufbauen, daß ihre Elemente über Zeiger miteinander verkettet sind. Sie besitzen die folgenden gemeinsamen Merkmale:

- Zwischen der Position eines Elements in der Liste und seiner Position im Speicher besteht kein Zusammenhang. Daher kann der Zugriff nicht über eine Adreßrechnung erfolgen.
- Der von den Listenelementen belegte Speicherplatz kann dynamisch verwaltet werden.
- Die Position eines Elements in der Zeigerkette ist identisch mit seiner logischen Position in der Folge der Listenelemente.

### Listentyp 1

Dieser Listentyp hat die folgende einfache Struktur:




Die C-Definiton zu dieser Datenstruktur lautet z.B. folgendermaßen:

```
struct Node *PNode; // Pointer auf ein Listenelement

struct Node  {                  // Listenelement:
               char  info[35]; // Informationsfeld
               PNode next;     // Kettungszeiger
             }
```

Die wesentlichen Eigenschaften dieser Listen-Variante sind:

- Listenanfang: es gibt einen Anker, der auf das erste Element der Liste zeigt.
- Listenende: ist durch einen NULL-Zeiger charakterisiert.
- Verkettung: Jedes Element enthält einen Vorwärtszeiger
- Positionszeiger: Zeigt direkt auf das Listenelement.

Wir betrachten einige für diese Datenstruktur typischen Operationen:

① Einfügen eines Elements am Anfang:
- Übernimm den Zeiger im Anker in den next-Zeiger des neuen Elements.
- Stelle den Anker auf das neue Element.

② Einfügen eines Elementes auf der Position p:
- Übernimm den next-Zeiger des alten *p*-ten Elementes in den next-Zeiger des neuen.
- Stelle den next-Zeiger des alten *p*-ten Elements auf das neue Element.
- Vertausche die Element-Information des alten *p*-ten Elementes und des neuen.

③ Entfernen des Elements an der Position *p*:
Sofern ein Element auf Position *p* im Innern der Liste entfernt werden soll, kann man wie folgt verfahren:
- Übernimm Element $a_{p+1}$ auf den Platz von $a_p$ .
- Übernimm den next-Zeiger von Element $a_{p+1}$ in den Next- Zeiger von Element $a_p$ .
- Gib den Speicherplatz des alten Elements $a_{p+1}$ wieder frei.

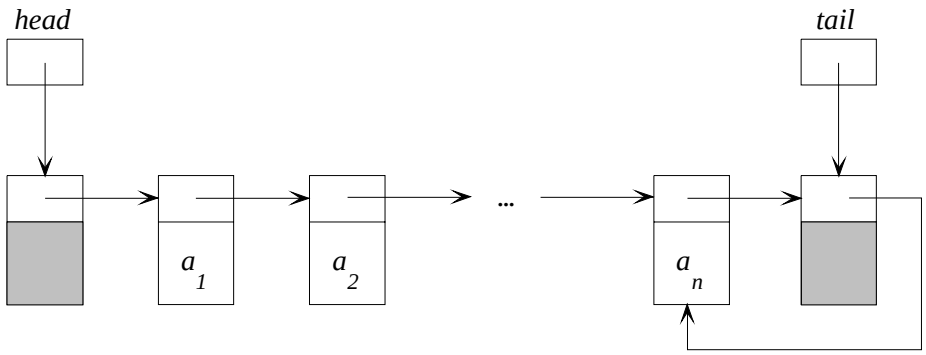
Falls das letzte Element der Liste entfernt werden soll, muß man jedoch den Vorgänger durch einen Suchprozeß ermitteln, der die ganze Liste durchläuft. Alternativ kann man in manchen Fällen auch zwei stets hintereinander herlaufende Zeiger verwenden.

## Listentyp 2

Dieser Listentyp verwendet einen zusätzlichen Zeiger auf das Ende der Liste und hängt am Anfang und am Ende je ein Dummy-Element an. Er vermeidet daurch einige der Nachteile des ersten Typs:

- Beim Durchsuchen der Liste muß man nicht ständig prüfen, ob das Listenende schon erreicht ist, wenn man am Listenende ein Stopper-Element ablegt.
- Es müssen weniger Sonderfälle behandelt werden.

Er hat die folgende Struktur:



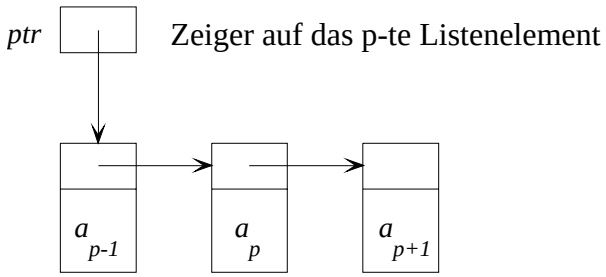
Die wesentlichen Eigenschaften dieser Variante sind:

- ♦ Dummy-Elemente: Vor und hinter der Liste wird je ein zusätzliches Element eingefügt.
- ♦ Listenanfang: Es gibt einen Anker "head", der auf das erste Element zeigt.
- ♦ Listenende: Es gibt einen Anker "tail", der auf das letzte Element zeigt. Das Dummy-Element am Ende der Liste verweist auf das letzte "richtige" Listenelement zurück.
- ♦ Verkettung: Jedes Element enthält einen Vorwärtszeiger.
- ♦ Positionszeiger: Zeigt direkt auf das Listenelement.

Problematisch bei dieser Variante ist immer noch das Entfernen von Elementen: Dazu muß man entweder die Liste noch einmal von Anfang an durchsuchen oder man muß zwei hintereinander herlaufende Positionszeiger führen. Die folgende, dritte Variante vermeidet diesen Nachteil:

## Listentyp 3:

Diese Implementierungstechnik ist fast identisch mit der 2. Variante. Sie handhabt aber Zeiger auf das Element $a_p$ der Liste etwas anders: Ein Zeiger auf das Element $a_p$ zeigt auf das Listenelement, dessen Vorwärtszeiger die Adresse des *p*-ten Elements enthält, wie die folgende Abbildung zeigt:

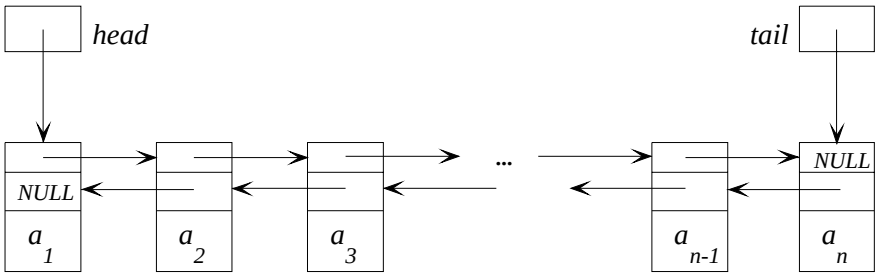



Damit wird das Aushängenen von Elementen ebenfalls sehr einfach: In PASCAL genügt dazu z.B. die Funktion

```
PNode deleteItem (PNode *item)
{ (*item)->next = (*item)->next->next; }
```

## Listentyp 4:

Die vierte Implementierungsvariante für lineare Listen besitzt Verkettungszeiger in zwei Richtungen. In Bezug auf head- und tail-Zeiger sowie auf Dummy-Elemente kommen alle zuvor besprochenen Alternativen in Frage. Sie hat z.B. folgende Struktur:




Bei dieser Variante sind das Entfernen und Vertauschen von Elementen sehr einfach und mit konstantem Zeitbedarf durchzuführen. Der Preis dafür ist der höhere Speicherplatzbedarf für die Zeiger.

## 3.1.4 Spezielle lineare Listen

In vielen Anwendungsbereichen benötigt man lineare Listen mit wenigen speziellen Operationen. Dazu gehören die folgenden Sonderformen:

### Stapel:

Zugriffe sind auf den Anfang der Liste beschränkt. Es sind daher nur die folgenden Operationen zulässig:

- Einfügen eines Elements am Anfang des Stapels (push)
- Entfernen eines Elements am Anfang des Stapels (pop)
- Lesen des ersten Elements (top)
- Leere Liste erzeugen (new_stack)
- Prüfen, ob Liste leer ist (stack_empty)

Stapelspeicher werden z.B. bei der Syntax-Analyse von Programmen während der Compilierung verwendet.

### Schlangen

Zugriffe sind auf den Anfang und das Ende der Schlange beschränkt. Die folgenden Operationen sind erlaubt:

- Einfügen eines Elements am Anfang der Schlange (push_tail)
- Entfernen eines Elements vom Ende der Schlange (pop_head)
- Lesen des ersten Elements (top)
- Leere Liste erzeugen (new_queue)
- Prüfen, ob Liste leer ist (queue_empty)

## 3.2 Matrizen

In der Mathematik werden Matrizen für die verschiedensten Zwecke eingesetzt. Ihre wesentliche Merkmale sind:

- die lineare Anordnung ihrer Komponenten in mehreren Dimensionen und
- die Gleichartigkeit aller Komponenten.

Matrixkomponenten werden durch einen Indexvektor identifiziert. Das folgende Beispiel zeigt dies für eine zweidimensionale Matrix, deren Komponeneten Alphabetzeichen sind:

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | ! | " | § | $ |
| 1 | % | & | / | @ |
| 2 | + | # | * | - |
| 3 | { | [ | ( | \| |
| 4 | > | < | = | ^ |

Wenn wir den Zeilenindex zuerst angeben, dann hat das Matrix-Element mit dem Indexvektor [1,3] den Wert "@".

Der Datentyp Matrix hat für zahlreiche Anwendungen größte Bedeutung (Lösung von Gleichungen, Integrationsverfahren, tabellengesteuerte Verfahren, ... ). Er gehört daher in vielen Programmiersprachen zu den elementaren Datentypen und ist teilweise sogar in der in der Hardware-Architektur berücksichtigt. In C erlauben mehrdimensionale Vektoren eine einfache Implementierung des Datentyps Matrix. Mit

```
char a[5][4]
```

wird eine zweidimensionale Matrix deklariert mit je 5 Zeilen und 4 Spalten. Der Zugriff auf eine Matrix-Komponente wird angegeben durch: `a[1,3]`

Dieser C-Datentyp ist nicht sehr flexibel: die Indexbereiche fangen immer bei 0 an und die Feldgrenzen sind nicht variabel. Andere Programmiersprachen sind in dieser Hinsicht flexibler.

### Implementierungstechniken für Matrizen

In vielen Anwendungen spielt die Art und Weise, wie Maztrizen im Speicher abgelegt werden und wie der Zugriff auf ihre Komponenten erfolgt eine sehr wesentliche Rolle.

**Beispiel:**

Eine Bandmatrix mit *1000x1000* Komponenten benötigt bei der Speicherung als zweidimensionaler Vektor eine Million Speicherplätze. Sind nur die Haupt- und Nebendiagonalen von Null verschieden, so reduziert sich der Platzbedarf bei geeigneter Speicherung auf ca *3000* Plätze.

Wir werden daher im folgenden verschiedene Speichertechniken für die Datenstruktur Matrix und die dazu gehörenden Zugriffsmethoden näher betrachten.

### Lexikographische Speicherung von Matrizen

Viele Compiler speichern die Komponenten einer zweidimensionalen Matrix Zeile für Zeile ab. Die Komponeneten der oben angegebenen 5x4-Matrix liegen dabei wie folgt nacheinander im Speicher:

| ! | " | § | $ | % | & | / | @ | + | # | * | - | { | [ | ( | \| | > | < | = | ^ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

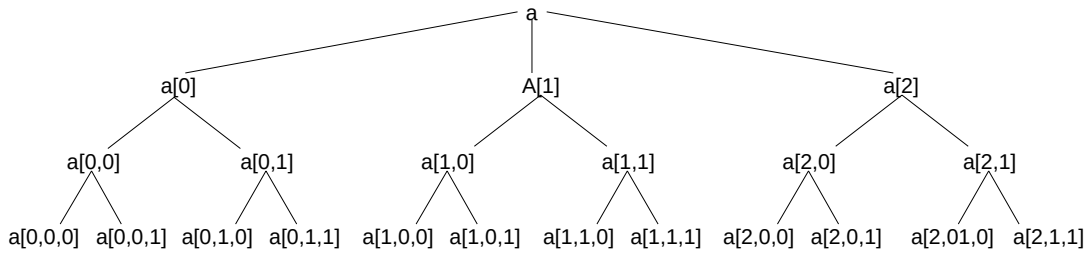
Diese Speicherung entspricht einer lexikographische Anordnung der IndexVektoren:

`a[0,0], a[0,1], a[0,2], a[0,3], a[1,0], a[1,1], a[1,2] ..... a[4,3]`

Dreidimensionale Matrizen kann man sich als Quader vorstellen, ihre zweidimensionalen Untermatrizen als „Scheiben“. Für Matrizen mit mehr als drei Dimensionen versagt die anschauliche Interpretation dieser Anordnung. Wir stellen uns eine Matrix beliebiger Dimension daher zweckmäßiger als einen Baum vor.

**Beispiel**

Eine dreidimensionale Matrix mit der Deklarationchar `A[3][2][2]` besitzt die folgende Darstellung als Baum:




In dieser Darstellung bezeichnet A die gesamte Matrix. Die Teilbäume `A[0]`, `A[1]`, `A[2]` sind gleich große Teilbäume auf dem 1. Hierarchielevel. a[i,j] ist ein Teilbaum auf dem 2. Hierarchielevel und `a[i,j,k]` ist das Element mit dem Index `[i,j,k]`.

### Zugriffsfunktion bei lexikographischer Speicherung

Bei lexikographischer Speicherung ist der Ort einer Matrixkomponente durch ihren Index vollständig bestimmt. Man kann daher durch eine Adreßrechnung auf sie zugreifen. Dazu legen wir eine n-dimensionalen Matrix mit der PASCAL-Deklaration

**VAR** a : **array** [$u_1..o_1$, ... , $u_n..o_n$] **of** <type>;

$u_i$, $o_i$ bezeichnen die unteren bzw. oberen Matrixgrenzen in der i-ten Dimension. $A[i_1,i_2,...,i_n]$ bezeichnet einen Teilbaum auf der k-ten Stufe des Baumes für $0 \le k \le n$.

1. Schritt:
In einem ersten Schritt berechnen wir die Größe von Teilbäume verschiedener Stufen, d.h. von Einzelkomponenten, Zeilen, Scheiben, usw. Bezeichnen wir mit $L$ die Länge einer Matrix-Komponente, dann erhalten wir für Teilbäume der untersten Stufe mit dem Index $n$

$$L_n = L$$

und für Teilbäume auf der $k$-ten Stufe der Matrix:

$$L_k = (o_{k+1} - u_{k+1} + 1) * L_{k+1}$$

Diese Rechnung ist unabhängig von einem bestimmten Zugriff vorab möglich.

2. Schritt:
Die Adresse eines Elementes $a[i_1, i_2, \ldots, i_n]$ erhalten wir, wenn wir von der Anfangsadresse des Gesamtbaumes alle Teilbäume überspringen, die vor $a[i_1, i_2, \ldots, i_n]$ im Speicher liegen.

Die Gesamtlänge dieses Bereichs ist

$$(i_1 - u_1) * L_1 + (i_2 - u_2) * L_2 + (i_3 - u_3) * L_3 + \cdots + +(i_n - u_n) * L_n$$

Damit ist die Adresse des Elements $a[i_1, i_2, \ldots, i_n]$

$$= Adr(a) + \sum (i_k - u_k) * L_k$$

$$= Adr(a) - \sum u_k * L_k + \sum i_k * L_k$$

Der Ausdruck

$$a_0 = Adr(a) - \sum u_k * L_k$$

enthält ausschließlich Größen, die bei der Einrichtung des Feldes bereits bekannt sind und nicht vom speziellen Indexvektor einer Komponente abhängen. Es genügt daher, $a_0$ einmalig zu berechnen und beim Zugriff auf eine Feldkomponente den vorbereiteten Wert zu verwenden, wodurch sich die Adreßrechnung wesentlich vereinfacht. $a_0$ heißt fiktiver Feldanfang. In C stimmt der fiktive Feldanfang mit der Anfangsadresse des Feld-Datenbereichs überein.

Die lexikographische Speichertechnik für den Datentyp Matrix besitzt somit die folgenden Charakteristika:

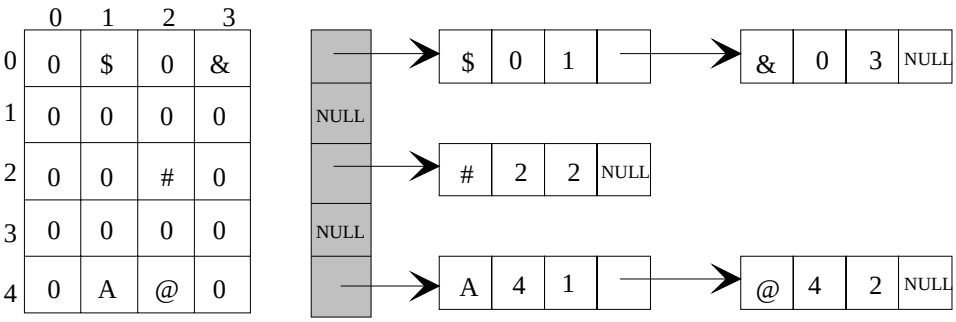
- die Anordnung der Matrix-Elemente im Speicher ist fest und hängt nur vom Indexvektor ab.
- der Zugriff läßt sich durch die Vorabauswertung des fiktiven Feldanfangs effizient gestalten; in C stimmt der effektive Feldanfang mit der Startadresse der Felddaten überein.
- Der Platzbedarf der Matrix hängt nur von den Indexbereichen ab.

### Speichertechniken für schwach besetzte Matrizen

In vielen Anwendungsbereichen fällt als besonderer Nachteil der lexikographischen Speicherung ins Gewicht, daß der Platzbedarf auch dann maximal ist, wenn sie nur schwach besetzt ist, wie es bei Bandmatrizen der Fall ist. Man hat daher Datenstrukturen für Matrizen entwickelt mit denen der Platzbedarf schwach besetzter Matrizen wesentlich reduziert werden kann.

Als ersten Ansatz zu einer Datenstruktur für schwach besetzte Matrizen verwenden wir für jede Zeile eine Liste, die nur die von 0 verschiedenen Elemente explizit enthält. Die folgende Abbildung stellt dies dar für eine Matrixstruktur der Form

```
char a[5][4];
```

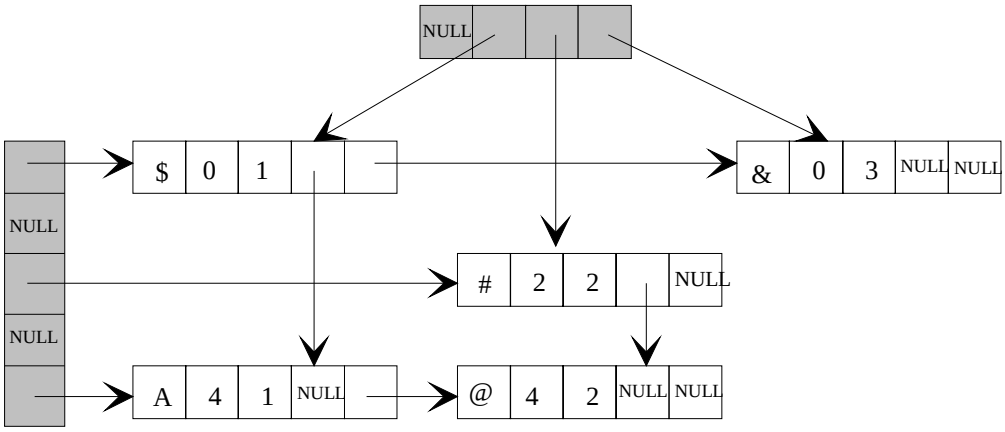


Wir stellen fest, daß außer den Werten von Matrix-Komponenten nun auch in jeder Matrixkomponente Verwaltungsinformation mitgespeichert werden muß, nämlich

- bei jedem Listenelement sein Index und die Verkettung mit dem nächsten Listenelement,
- ein Ankervektor mit Zeigern auf die Matrix-Zeilen

Dieser Ansatz reicht in der Regel noch nicht aus. Wenn wir zwei solche Matrizen miteinander multiplizieren, ist der Zugriff zu aufeinanderfolgenden Elementen einer Zeile einfach, Die Elemente einer Matrix-Spalte lassen sich aber nur durch einen umständlichen Suchprozeß der Reihe nach beschaffen.

Die folgende Datenstruktur erhält daher eine zusätzliche Verkettung aller Matrixelemente, die zur gleichen Spalte gehören.




Bei dieser Speicherungsform ist jedes Listenelement um einen weiteren Zeiger ergänzt worden, der die Elemente der selben Spalte miteinander verkettet. Damit lassen sich auch die Elemente einer Spalte schnell finden, falls sie der Reihe nach gebraucht werden.

# 4 Sortierverfahren

Das Ziel eines Sortiervorgangs ist es stets, eine Menge von Elementen nach vorgegebenen Kriterien zu ordnen. Häufig liegt auf der Menge eine lineare Halbordnung vor und die Elemente sind mit Bezug diese auf- oder absteigend zu ordnen.

Nach verschiedenen empirischen Untersuchungen werden zwischen *20 %* und *50 %* der gesamten CPU-Zeit aller Rechner für Sortierprozesse verbraucht. Dies mag ein Argument dafür sein, daß Methoden zum Sortieren sehr intensiv untersucht wurden. Die große Zahl von Algorithmen läßt sich jedoch auf wenige grundlegende Methoden zurückführen.

Bei der Auswahl eines Sortierverfahrens für einen bestimmten Zweck ist seine Eignung dafür zu beurteilen. Wir werden daher zu allen betrachteten Algorithmen Abschätzungen über ihre Komplexität aufstellen. Dabei interessieren im wesentlichen

- die benötigte Rechenzeit und
- der benötigte Speicherplatz für das Verfahren.

Einige Beispiele verdeutlichen die Vielfalt der Anwendungsfälle, in denen Sortieren eine Rolle spielt:

### Beispiel 1

Um herauszufinden, welche Merkmale "gute" von weniger guten Informatikstudenten unterscheiden, können wir sie z.B. nach dem Notendurchschnitt sortieren und im ersten und letzten Viertel der sortierten Liste nach gemeinsamen Merkmalen suchen (Lebensalter, Schulausbildung, Berufserfahrung, Studiendauer, Größe, Gewicht, Haarfarbe, Geschlecht, ... ). Man muß sich natürlich bewußt sein, daß damit keine Kausalbeziehungen nachzuweisen sind.

### Beispiel 2

Es soll festgestellt werden, wieviele verschiedene Wörter ein Text enthält. Dazu sortiert man alle Wörter alphabetisch und kann dann in einem Durchgang abzählen, wieviele verschiedene vorkommen.

### Beispiel 3

Es sollen Steuersünder ermittelt werden. Dafür liegen zwei Datenbestände vor:

- einige Millionen Steuererklärungen und
- einige Millionen Gehalts-Auszahlungsbelege.

Das Problem wird dadurch gelöst, daß jeder Steuerpflichtige eine Steuernummer erhält, die sowohl in den Steuererklärungen als auch auf den Auszahlungsbelegen angegeben ist. Dann sortiert man beide Datenbestände und kann bei einem einmaligen Durchlauf feststellen

- welche Steuererklärungen unvollständig sind und
- welche Gehaltsempfänger keine korrekte Steuererklärung abgegeben haben.

### Beispiel 4

Bei der Präsentation von Datenmaterial werden Zusammenhänge oft erst durch eine geeignete Sortierung erkennbar. Sortieren nach unterschiedlichen Kriterien kann Hinweise auf unbekannte Zusammenhänge geben, etwa:

- Wenn die Krebsrate in einer Region sich vom Durchschnitt unterscheidet, kann man dort gezielt nach den Ursachen suchen.
- Die Umsatzzahlen eines Produkts werden aufgeschlüsselt nach Kundenmerkmalen, Vertriebspersonal, Regionen, Jahreszeit usw. Daraus lassen sich Anhaltspunkte für die Absatz-Optimierung gewinnen.

# 4.1 Grundsätzliche Gesichtpunkte des Sortierens

### Präzisierung der Problemstellung

Zunächst wollen wir die Problemstellung des Sortierens präziser formulieren. Wir gehen aus von einer endlichen Menge von Items

$$\{i_1, i_2, \cdots, i_n\}$$

Jedes Item $i_j$ soll einen Schlüssel $k_j$ (Key) besitzen. Die Menge der möglichen Schlüssel bildet eine lineare Halbordnung, d.h. je zwei Schlüssel sind stets vergleichbar.

Die meisten Sortierverfahren benutzen außer der Halbordnung keine weitere Struktur-Eigenschaft der Schlüssel. Eine Ausnahme machen die Radix-Sortierverfahren, die ganze Zahlen als Schlüssel verwenden und dafür eine Zifferndarstellung über einer festen Basis voraussetzen.

Das Egebnis eines Sortierprozesses ist eine Permutation der Item-Indizes

$$\pi: [1..N] \rightarrow [1..N]$$

für die die permutierte Folge der Schlüssel bezüglich der Halbordnung eine aufsteigende Folge bildet:

$$k_{\pi_{(1)}} \leq k_{\pi_{(2)}} \leq k_{\pi_{(3)}} \leq \cdots \leq k_{\pi_{(N)}}$$

### Datenstrukturen zum Sortieren

Für die Darstellung und Untersuchung von Sortierverfahren legen wir als Itemstruktur einen Record der folgenden Art zugrunde:

```
typedef struct tagITEM
{ int      key;         // Schlüsselfeld
  char     info[35];    // Informationsfeld
} ITEM;
```

Die zu sortierende Menge nehmen wir als ein Feld an, in dem die Items zusammenhängend abgelegt sind:

```
ITEM A[101];
```

Aus programmtechnischen Gründen verwenden wir für die eigentliche Liste nur den Indexbereich `[1..100]` . Das Erste Element `A[0]` wird für z.B. als Stopper-Element bei Suchschleifen benutzt.

Beim Sortieren werden die Items in die Reihenfolge gebracht, die von der Permutation π vorgeschrieben ist. In der sortierten Liste sind die Items also nach Schlüsseln aufsteigend geordnet:

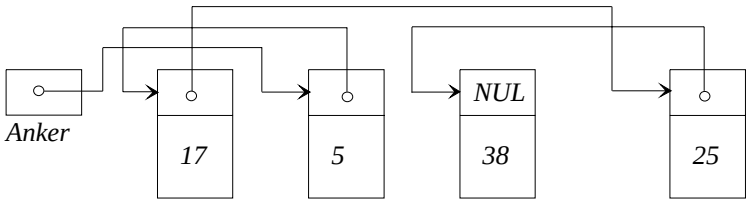
```
A[1].key  ≤  A[2].key  ≤  ...  ≤ A[N].key
```

Bei der praktischen Anwendung der Verfahren werden Modifikationen dieser Annahmen notwendig, die jedoch keine prinzipiellen Auswirkungen haben:

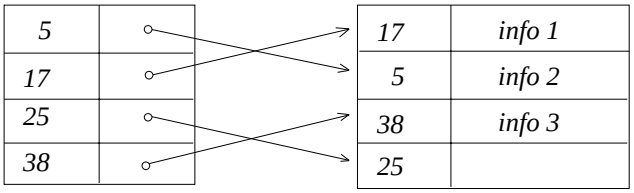
- Außer ganzen Zahlen werden auch andere Schlüsseltypen verwendet, z.B. alphanumerische Felder.
- Die Schlüssel sind oft integriert in den Informationsteil der Items.
- Die Item-Information kann getrennt vom Item gespeichert sein. In diesem Fall enthält das Item nur den Schlüssel und einen Zeiger zur Item-Information.
- Statt in einem Feld `A` kann die Item-Menge auch als Liste vorliegen. Indiesem Fall ist die Itemstruktur um die notwendige Verkettung zu erweitern

Das explizite Umspeichern von Items ist nur für kurze Records sinnvoll. Effektiver sind oft die folgenden Techniken:

- Sortieren verketteter Listen
  Dabei werden die Items um einen Kettungszeiger erweitert. Statt die Items physikalisch umzuordnen muß nur die Reihenfolge in der Verkettung geändert werden.




- Sortieren einer Schlüssel-Tabelle
  Es wird eine Tabelle aufgebaut, die die Schlüssel der Items und Zeiger zu diesen enthält. Beim Sortieren werden dann nur die Elemente der Schlüsseltabelle umgeordnet.




### Elementare und Spezielle Sortierverfahren

Eine häufig benutzte Klassifizierung teilt Sortieralgorithmen in elementare und spezielle Verfahren ein. Die elementaren Sortierverfahren machen keine Annahmen über die Datenstruktur, in der die Items vorliegen. Die Komplexität dieser Verfahren ist daher auch relativ hoch, nämlich $\mathbf{O}\left(N^2\right)$. Verfahren dieses Typs sind z.B.

- das Selection Sort-Verfahren
- das Insertion Sort-Verfahren
- das Shell Sort-Verfahren
- das Bubble Sort-Verfahren

Spezielle Sortierverfahren nutzen im Gegensatz dazu die Eigenschaften bestimmter Datenstrukturen oder zusätzliche Strukturen auf der Schlüsselmenge aus und erreichen dadurch auch eine geringere Komplexität, z.B. $\mathbf{O}\left(N*\log(N)\right)$. Verfahren aus dieser Gruppe sind:

- das Quicksort-Verfahren
- das Heapsort-Verfahren
- das Merge Sort-Verfahren
- das Radix Sort-Verfahren

### Interne und externe Sortierverfahren

*Interne Sortierverfahren* gehen davon aus, daß die zu sortierenden Items vollständig in den Arbeitsspeicher passen. *Externe Sortierverfahren* lesen die Items von Externspeichern, weil diese nicht vollständig in den Arbeitsspeicher passen. Als Externspeicher werden in der Regel sequentielle Speichermedien, z.B. Magnetbänder angenommen.

### Sequentielle und parallele Verfahren

Alle klassischen Sortiermethoden setzen eine von Neumann-Architektur des Rechners voraus. Neuere Arbeiten versuchen, auch Verfahren für Parallelrechner-Architekturen zu entwickeln.

### Vorsortierung

Oft ist der zu sortierende Datenbestand bereits in bestimmter Weise vorsortiert. Es wäre daher zu wünschen, daß dies bei der Sortierung berücksichtigt wird. Bei den meisten Sortierverfahren wird die Vorsortierung nicht ausgenutzt. Quicksort verhält sich bei einer schon vollständig sortierten Liste sogar besonders ungünstig.

### Stabilität

Sortierverfahren heißen stabil, wenn sie die Reihenfolge von Items mit gleichem Schlüssel nicht verändern. Diese Eigenschaft spielt dann eine Rolle, wenn nacheinander nach verschiedenen Schlüsseln sortiert wird: Eine Notenliste wird zunächst alphabetisch nach dem Familiennamen sortiert und danach nach den Noten. Dabei sollen Studenten mit gleicher Note immer noch alphabetisch geordnet bleiben.

### Komplexitätsmaße für Sortierverfahren

Die Laufzeitkomplexität von Sortierverfahren messen wir durch

- die Anzahl $C$ von Schlüsselvergleichen (Comparisons) und
- die Anzahl $M$ von Item-Bewegungen (Movements).

Beide Maßzahlen beziehen wir auf die Anzahl der zu sortierenden Items. Außerdem interessieren uns Maximal-, Minimal- und Durchschnittswerte dieser Größen. Wir bestimmen also:

$C_{min}(N)$ = minimale Anzahl von Vergleichen bei $N$ Items,

$C_{max}(N)$ = maximale Anzahl von Vergleichen bei $N$ Items,

$C_{av}(N)$ = durchschnittliche Anzahl von Vergleichen bei $N$ Items

und

$M_{min}(N)$ = minimale Anzahl von Bewegungen bei $N$ Items,

$M_{max}(N)$ = maximale Anzahl von Bewegungen bei $N$ Items,

$M_{av}(N)$ = durchschnittliche Anzahl von Bewegungen bei $N$ Items.

# 4.2 Elementare Sortiermethoden

Die elementaren Sortierverfahren machen keine Annahmen über die Datenstruktur, in der die Items abgespeichert sind. Von den Schlüsseln wird nur vorausgesetzt, daß sie eine lineare Halbordnung bilden. Wir betrachten in diesem Abschnitt vier typische Methoden dieser Klasse:

- Selection Sort,
- Insertion Sort,
- Shell Sort und
- Bubble Sort

## 4.2.1 Selection Sort

Andere Bezeichnungen für diese Methode sind *Sortieren durch Austauschen* und S*ortieren durch Auswählen*. Die Grundidee besteht darin, zunächst das kleinste, und danach sukzessiv die nächst größeren Elemente auszusuchen und diese zu einer geordneten Liste zusammenzustellen.

### Methode

- Wir wählen zunächst unter den Items `A[1], ... , A[N]` dasjenige mit dem kleinsten Schlüssel aus und vertauschen es mit `A[1]`. Dieses Item hat damit bereits seinen endgültigen Platz gefunden.
- Nun verfahren wir mit der Restliste analog, d.h. falls die Teilliste `<A[1], ... , A[k]>` bereits die *k* kleinsten Elemente in der richtigen Reihenfolge enthält, dann suchen wir in der Restliste `<A[k+1], ... , A[N]>` das kleinste Item und vertauschen es mit `A[k+1]`.

### Beispiel

Die unsortierte Itemliste ist

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 33 | 48 | 3 | 75 | 17 | 27 | 11 | 81 | 5 | 23 |

Im ersten Verfahrensschritt wird `A[3]` mit `A[1]` vertauscht. Es ergibt sich dann der folgende Zustand, wobei die bereits endgültig geordnete Teilliste durch ein Raster unterlegt ist:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 48 | 33 | 75 | 17 | 27 | 11 | 81 | 5 | 23 |

Die Ergebnisse der weiteren Schritte werden nun in gleicher Weise zusammenfassend dargestellt:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 5 | 33 | 75 | 17 | 27 | 11 | 81 | 48 | 23 |
| 3 | 5 | 11 | 75 | 17 | 27 | 33 | 81 | 48 | 23 |
| 3 | 5 | 11 | 17 | 75 | 27 | 33 | 81 | 48 | 23 |
| 3 | 5 | 11 | 17 | 23 | 27 | 33 | 81 | 48 | 75 |
| 3 | 5 | 11 | 17 | 23 | 27 | 33 | 81 | 48 | 75 |
| 3 | 5 | 11 | 17 | 23 | 27 | 33 | 81 | 48 | 75 |
| 3 | 5 | 11 | 17 | 23 | 27 | 33 | 48 | 81 | 75 |
| 3 | 5 | 11 | 17 | 23 | 27 | 33 | 48 | 75 | 81 |

### C - Routine

Die folgende C-Routine demonstriert das Verfahren an einem einfachen Zahlenfeld:

```
//   Sortieren durch Auswaehlen
void SelectionSort (int *A, int n)
{
  int  i, j;                // Laufvariablen
  int  min;                 // Minimum im Restfeld
  int  item;                // Hilfsfeld zum Vertauschen von Items

  for (i=1;i<n;i++)         // durchlaeuft die zu sortierende Liste
  {
    min = i;                // Index des bisher kleinesten Elements
    for (j=i+1;j<=n;j++)    // Suche nach noch kleineren Elementen
      if (A[j]<A[min]) min = j;
    item   = A[min];        // jetzt werden das erste und das kleinste
    A[min] = A[i];          // Item der Restliste vertauscht
    A[i]   = item;
  }
}
```

### Analyse

Die Anzahl von Vergleichen, die zur Bestimmung des Minimums der jeweiligen Restliste notwendig sind, beträgt *N-1* beim ersten Durchlaufen der `i`-Schleife, dann *N-2* und schließlich *1*. Diese Anzahl ist für alle Listen die selbe, daher ist

$$\boldsymbol{C}_{min}(N) = \boldsymbol{C}_{max}(N) = \boldsymbol{C}_{av}(N) = \sum_{i=1}^{N-1} i \;= \frac{N(N-1)}{2} = \boldsymbol{O}(N^2)$$

Die Anzahl der Item-Vertauschungen ist in dieser Form des Verfahrens ebenfalls unabhängig von der Belegung der Liste. Sie beträgt *N-1*. Da jede Vertauschung drei Item-Bewegungen impliziert, ergibt sich

$$\boldsymbol{M}_{min}(N) = \boldsymbol{M}_{max}(N) = \boldsymbol{M}_{av}(N) = 3(N-1)$$

Das Verfahren läßt sich natürlich in mancher Hinsicht verbessern, z.B. kann es sinnvoll sein, Item-Vertauschungen nur auszuführen, wenn die beiden Item-Schlüssel wirklich verschieden sind. Die Selection Sort-Methode erfordert zwar $\mathbf{O}(N^2)$ Vergleiche. Sie kann aber trotzdem günstig sein, wenn die Item-Transporte in Relation zu den Schlüssel-Vergleichen sehr aufwendig sind, weil nur linear viele solcher Transporte gebraucht werden.

**Selection Sort**

| | | |
|---|---|---|
| Speicherbedarf | : | in-situ-Verfahren |
| Stabilität | : | ist **nicht** gewährleistet |
| Datenstruktur | : | Array, Liste |
| Vorsortierung | : | keine Auswirkung auf das Laufzeitverhalten |
| Laufzeitverhalten | : | Vergleiche: $\boldsymbol{C}_{min}(N) = \boldsymbol{C}_{max}(N) = \boldsymbol{C}_{av}(N) = \boldsymbol{O}(N^2)$ |
| | | Item-Transporte: $\boldsymbol{M}_{min}(N) = \boldsymbol{M}_{max}(N) = \boldsymbol{M}_{av}(N) = 3(N-1)$ |
| Anwendung | : | günstig, wenn Item-Transporte aufwendig sind |

## 4.2.2 Insertion Sort

Dieses Verfahren wird auch *Sortieren durch Einfügen* genannt. Die Grundidee besteht darin, in eine bereits sortierte Ziel-Liste, die zunächst leer ist, neue Elemente an der richtigen Position einzufügen, solange bis die Restliste aufgebraucht ist.

### Methode

- Wir bilden zunächst mit dem Item `A[1]` die erste Teilliste der Länge `1`. Diese ist natürlich sortiert.
- Wenn die Items `<A[1], ..., A[k]>` bereits eine sortierte Teilliste bilden, dann fügen wir `A[k+1]` an der richtigen Stelle darin ein, wobei alle größeren Elemente der Teilliste um eine Position nach rechts verschoben werden.

### Beispiel

Die unsortierte Itemliste ist

| *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* |
|---|---|---|---|---|---|---|---|---|---|
| *33* | *48* | *3* | *75* | *17* | *27* | *11* | *81* | *5* | *23* |

Im ersten Verfahrensschritt wird `A[1]` als erste Teilliste abgetrennt:

| *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* |
|---|---|---|---|---|---|---|---|---|---|
| *33* | *48* | *3* | *75* | *17* | *27* | *11* | *81* | *5* | *23* |

Im zweiten Schritt wird `A[2]` in die Teilliste aufgenommen. Wegen *33<48* sind dabei keine Verschiebungen notwendig:

| *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* |
|---|---|---|---|---|---|---|---|---|---|
| *33* | *48* | *3* | *75* | *17* | *27* | *11* | *81* | *5* | *23* |

Die nächsten Einfügeschritte werden nun zusammengefaßt dargestellt. Die bereits sortierte Ziel-Liste ist mit einem Raster unterlegt:

| *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* |
|---|---|---|---|---|---|---|---|---|---|
| *3* | *33* | *48* | *75* | *17* | *27* | *11* | *81* | *5* | *23* |
| *3* | *33* | *48* | *75* | *17* | *27* | *11* | *811* | *5* | *23* |
| *3* | *17* | *33* | *48* | *75* | *27* | *11* | *81* | *5* | *23* |
| *3* | *17* | *27* | *33* | *48* | *75* | *11* | *81* | *5* | *23* |
| *3* | *11* | *17* | *27* | *33* | *48* | *75* | *81* | *5* | *23* |
| *3* | *11* | *17* | *27* | *33* | *48* | *75* | *81* | *5* | *23* |
| *3* | *5* | *11* | *17* | *27* | *33* | *48* | *75* | *81* | *23* |
| *3* | *5* | *11* | *17* | *23* | *27* | *33* | *48* | *75* | *81* |

### C - Routine

```
void InsertionSort (int *A, int n)
{
  int  i, j;              // Laufvariablen
  for (i=2;i<=n;i++)      // Schleife über alle Restlisten
  {
    // Sichere einzufügendes Item. A[0] ist dabei auch Stopper-Element
    A[0] = A[i];

    // Suche die richtige Einfügepostion in der sortierten Teilliste
    // Verschiebe dabei größere Items um eine Position nach rechts
    j = i;
    while (A[j-1] > A[0])
    {
      A[j] = A[j-1];
      j--;
    }
    // Füge das Element A[i] an der richtigen Einfügeposition ein
    A[j] = A[0];
  }
}
```

### Analyse

Die Anzahl von Schlüsselvergleichen, die zur Bestimmung der Einfügeposition in einer Teilliste der Länge $i$ notwendig sind, beträgt minimal $1$, maximal $i+1$ (weil evtl. auch noch mit dem Stopper-Element verglichen wird). Es sind $N-1$ Teillisten zu bearbeiten mit den Längen $1, 2, ..., N-1$. Wir erhalten also

$$\boldsymbol{C}_{min}(N) = N-1$$
$$\boldsymbol{C}_{max}(N) = \sum_{i=1}^{N-1}(i+1) = \boldsymbol{O}\left(N^2\right)$$

Die minimale und maximale Anzahl von Item-Transporten ist analog zu ermitteln: Es sind für jede Ziel-Liste mindestens $2$ Item-Transporte nötig, um das einzufügende Element nach `A[0]` und von dort in die Ziel-Liste zu bringen. Im schlechtesten Fall müssen zusätzlich noch alle $i$ Elemente der Ziel-Liste nach rechts verschoben werden $(i=1, 2,...,N-1)$. Damit erhalten wir:

$$\boldsymbol{M}_{min}(N) = 2(N-1)$$
$$\boldsymbol{M}_{max}(N) = \sum_{i=1}^{N-1}(i+2) = \boldsymbol{O}\left(N^2\right)$$

Wenn keine weiteren Informationen über die Vorsortierung vorliegen, ist zu erwarten, daß bei einer Ziel-Liste der Länge $i$ die Hälfte der Elemente zu durchsuchen und zu verschieben ist, so daß man für die mittlere Gesamtzahl von Vergleichen und Item-Transporten erhält:

$$\boldsymbol{C}_{av}(N) = \sum_{i=1}^{N-1}\frac{i}{2} = \boldsymbol{O}\left(N^2\right)$$
$$\boldsymbol{M}_{av}(N) = \sum_{i=1}^{N-1}\left(\frac{i}{2}+2\right) = \boldsymbol{O}\left(N^2\right)$$

Das Verfahren läßt sich geringfügig verbessern, wenn man zum Suchen der Einfügeposition ein binäres Suchverfahren benutzt. Damit erspart man sich allerdings nicht die ebenfalls mit $\boldsymbol{O}\left(N^2\right)$ wachsende Zahl von Item-Transporten. Eine wesentliche Verbesserung der Methode wird möglich, wenn man statt des Array `A` als Datenstruktur eine Liste verwendet.

**Insertion Sort**

| | | |
|---|---|---|
| Speicherbedarf | : | in-situ-Verfahren |
| Stabilität | : | ist gewährleistet |
| Datenstruktur | : | Array, Liste |
| Vorsortierung | : | wirkt positiv, wenn der *globale* Sortierzustand gut ist |
| Laufzeitverhalten | : | $\boldsymbol{C}_{min}(N) = \boldsymbol{O}(N)$ |
| | | $\boldsymbol{C}_{max}(N) = \boldsymbol{C}_{av}(N) = \boldsymbol{O}(N^2)$ |
| | | $\boldsymbol{M}_{min}(N) = \boldsymbol{O}(N)$ |
| | | $\boldsymbol{M}_{max}(N) = \boldsymbol{M}_{av}(N) = \boldsymbol{O}(N^2)$ |
| Anwendung | : | als Sub-Methode beim Shell Sort-Verfahren |

## 4.2.3 Shell Sort

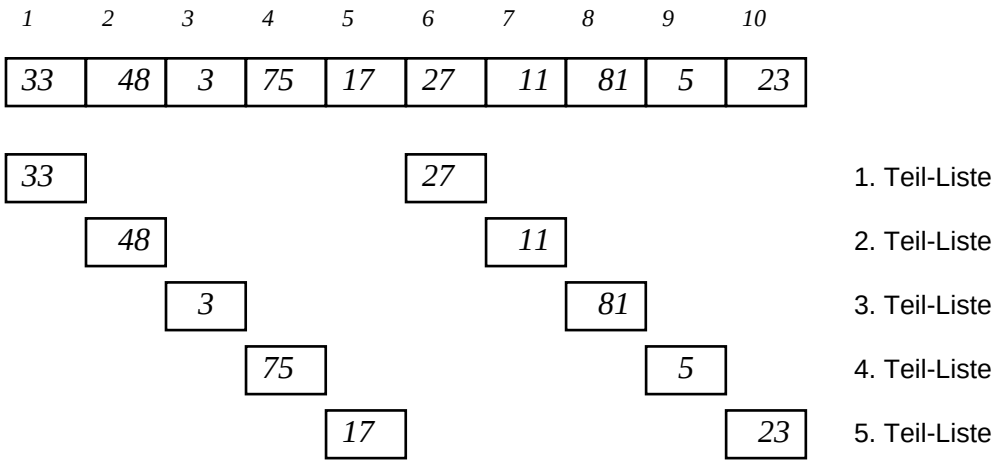
Das Shell Sort-Verfahren geht auf einen Vorschlag von D.L. Shell zurück. Es basiert auf der selben Idee wie die Insertion Sort-Methode, versucht jedoch ihren wesentlichen Nachteil zu vermeiden, daß nämlich bei jedem Einfügeschritt die Items durch Verschieben der Ziel-Liste nach rechts ihrer endgültigen Position um maximal eine Stelle näher rücken können. Es ist heute eher von akademischem Interesse.

### Methode

- Wir wählen zunächst eine abnehmende Folge ganzzahliger Inkremente aus:

$$h_t > h_{t-1} > \cdots > h_1 = 1$$

- Zuerst zerlegen wir in die Originalliste $\boldsymbol{L}$ zunächst in $h_t$ einzelne Teil-Listen. Die Items einer Teil-Liste sind um je $h_t$ Indexpositionen voneinander entfernt. Die folgende Skizze zeigt dies für den Fall $h_t = 5$. Die Teil-Listen sind zur Verdeutlichung einzeln untereinander geschrieben:




- Jede dieser Teillisten wird für sich mit dem Insertion Sort-Verfahren sortiert. Dabei ergeben sich bei den Einfügeschritten Verschiebungen über jeweils $h_t$ Indexpositionen.
- Nun wiederholt man das Ganze für die Inkremente $h_{t-1}$ , $h_{t-2}$ bis $h_1 = 1$ . Danach ist die Liste vollständig sortiert.

### Beispiel

Wir gehen von der obigen unsortierten Liste aus:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 33 | 48 | 3 | 75 | 17 | 27 | 11 | 81 | 5 | 23 |

Zunächst beginnen wir mit einem Inkrement $h=5$ und müssen daher die folgenden Teillisten durch Einfügen sortieren:

$$<A_1, A_6> \; <A_2, A_7> \; <A_3, A_8> \; <A_4, A_9> \; <A_5, A_{10}>$$

Das Ergebnis nach diesen fünf unabhängigen Sortierprozessen ist die folgende Liste:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 27 | 11 | 3 | 5 | 17 | 33 | 48 | 81 | 75 | 23 |

Als nächstes wählen wir das Inkrement $h=3$ und erhalten drei Teillisten, nämlich

$$<A_1, A_4, A_7, A_{10}> \quad <A_2, A_5, A_8> \quad <A_3, A_6, A_9>$$

Diese sortieren wir wieder unabhängig voneinander und erhalten als Gesamtliste:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 11 | 3 | 23 | 17 | 33 | 27 | 81 | 75 | 48 |

Zuletzt wählen wir als Inkrement $h=1$, d.h. wir führen ein ganz normales Insertion Sort-Verfahren durch, das wegen der schon erreichten Vorsortierung jedoch mit weniger Aufwand auskommt. Es verläuft im einzelnen wie folgt:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 11 | 3 | 23 | 17 | 33 | 27 | 81 | 75 | 48 |
| 5 | 11 | 3 | 23 | 17 | 33 | 27 | 81 | 75 | 48 |
| 3 | 5 | 11 | 23 | 17 | 33 | 27 | 81 | 75 | 48 |
| 3 | 5 | 11 | 23 | 17 | 33 | 27 | 81 | 75 | 48 |
| 3 | 5 | 11 | 17 | 23 | 33 | 27 | 81 | 75 | 48 |
| 3 | 5 | 11 | 17 | 23 | 33 | 27 | 81 | 75 | 48 |
| 3 | 5 | 11 | 17 | 23 | 27 | 33 | 81 | 75 | 48 |
| 3 | 5 | 11 | 17 | 23 | 27 | 33 | 81 | 75 | 48 |
| 3 | 5 | 11 | 17 | 23 | 27 | 33 | 75 | 81 | 48 |
| 3 | 5 | 11 | 17 | 23 | 27 | 33 | 48 | 75 | 81 |

Die durch das Einfügen entstehenden sortierten Anfangs-Listen sind mit einem Raster unterlegt.

### C - Routine

```
void ShellSort (int *A, int n)
{
  int  i, j, h;                    // Laufvariablen
  int  item;                       // Hilfsfeld zum Vertauschen

   for (h=1; h<=n/9; h=3*h+1) ;    // h fuer n/9 Teillisten einstellen
   for ( ; h>0 ; h /= 3)           // Schleife über die Inkremente h
    for (i=h+1; i<=n ;i++)
    {
      item = A[i]; j=i;            // Teil-Liste mit Insertion Sort
      while (j>h && A[j-h]>item)   // sortieren
        {A[j] = A[j-h]; j -=h;}
      A[j] = item;
    }
}
```

### Analyse

Eine Leistungsanalyse für das Shellsort-Verfahren müßte vor allem eine Anwort auf die Frage nach einer günstigen Wahl der Inkremente geben. Man kann z.B. zeigen ,daß bei Inkrementen der Form $2^p\,3^q$ die Laufzeit von der Ordnung $\mathbf{O}\left(N*\log^2(N)\right)$ ist. Allerdings gibt es keine generell brauchbaren Aussagen zu dieser Frage.

**Shell Sort**

| | | |
|---|---|---|
| Speicherbedarf | : | in-situ-Verfahren |
| Stabilität | : | ist nicht gewährleistet |
| Datenstruktur | : | Array |
| Vorsortierung | : | wirkt sich nur für die Teilprozesse positiv aus |
| Laufzeitverhalten | : | $\mathbf{O}\left(N*\log^2(N)\right)$ für Inkremente der Form $2^p\,3^q$ |
| Anwendung | : | nur noch von akademischem Interesse |

## 4.2.4 Bubble Sort

Das Bubble Sort-Verfahren beruht auf der Grundidee, benachbarte Items solange zu vertauschen, bis alle in der richtigen Reihenfolge stehen. Es wird deshalb auch *Sortieren durch lokales Vertauschen* genannnt.

### Methode

- Wir durchlaufen die Item-Liste von links nach rechts und vertauschen dabei immer das aktuelle Item mit seinem rechten Nachbarn, falls dieser kleiner ist. So gelangt das größte Element ganz ans rechte Ende der Liste, wo es seinen endgültigen Platz erreicht hat. Die Analogie zu aufsteigenden Blasen in Wasser hat dem Verfahren seinen Namen gegeben.
- Nun durchlaufen wir die Rest-Liste nach der selben Methode wieder von Anfang an und bringen damit das zweitgrößte Element auf seinen endgültigen Platz, nämlich auf der Position *N-1*.
- Das Ganze wird so lange für immer kürzere Restlisten wiederholt, bis alle Items in der sortierten Reihenfolge stehen.

### Beispiel

Für den ersten Durchlauf des Verfahrens ergeben sich die folgenden Zwischenzustände. Die beiden zu vertauschenden Items sind durch ein Raster unterlegt:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *33* | *48* | *3* | *75* | *17* | *27* | *11* | *81* | *5* | *23* |
| *33* | *48* | *3* | *75* | *17* | *27* | *11* | *81* | *5* | *23* |
| *33* | *3* | *48* | *75* | *17* | *27* | *11* | *81* | *5* | *23* |
| *33* | *3* | *48* | *75* | *17* | *27* | *11* | *81* | *5* | *23* |
| *33* | *3* | *48* | *17* | *75* | *27* | *11* | *81* | *5* | *23* |
| *33* | *3* | *48* | *17* | *27* | *75* | *11* | *81* | *5* | *23* |
| *33* | *3* | *48* | *17* | *27* | *11* | *75* | *81* | *5* | *23* |
| *33* | *3* | *48* | *17* | *27* | *11* | *75* | *81* | *5* | *23* |
| *33* | *3* | *48* | *17* | *27* | *11* | *75* | *5* | *81* | *23* |
| *33* | *3* | *48* | *17* | *27* | *11* | *75* | *5* | *23* | *81* |

In der nachfolgenden Übersicht ist jeweils der letzte Zustand nach einem weiteren Durchlauf durch die Restliste dargestellt. Die bereits sortierte Ziel-Liste am rechten Ende ist mit einem Raster unterlegt.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *3* | *33* | *17* | *27* | *11* | *48* | *5* | *23* | *75* | *81* |
| *3* | *17* | *27* | *11* | *33* | *5* | *23* | *48* | *75* | *81* |
| *3* | *17* | *11* | *27* | *5* | *23* | *33* | *48* | *75* | *81* |
| *3* | *11* | *17* | *5* | *23* | *27* | *33* | *48* | *75* | *81* |
| *3* | *11* | *5* | *17* | *23* | *27* | *33* | *48* | *75* | *81* |
| *3* | *5* | *11* | *17* | *23* | *27* | *33* | *48* | *75* | *81* |
| *3* | *5* | *11* | *17* | *23* | *27* | *33* | *48* | *75* | *81* |

Beim letzten Durchlauf durch die Restliste mit der Länge 3 werden keine Vertauschungen mehr vorgenommen, so daß das Verfahren vorzeitig beendet werden kann.

### C - Routine

```
void BubbleSort (int *A, int n)
{  int  i, j;              // Laufvariablen
   int  item;              // Hilfsfeld zum Vertauschen zweier Elemente
   int  xchg;              // Flag = 1   wenn eine Vertauschung vorkam
                           //  Flag = 0   sonst
   i = n;                  // i = letzte Position in der Restliste
   do
   {  xchg = 0;            // Flag = 0: bisher noch keine Vertauschung
      for (j=1;j<i;j++)    // Restliste durchlaufen
        if (A[j]>A[j+1])   // Nachbarn in richtige Reihenfolge bringen
        {
          item   = A[j];
          A[j  ] = A[j+1];
          A[j+1] = item;
          xchg   = 1;      // Flag = 1: es wurde vertauscht !
        }
      i--;
    } while (xchg == 1);   // wiederhole bis im letzten Durchlauf
                           // keine Vertauschung mehr aufgetreten ist
}
```

### Analyse

Der günstigste Fall beim Bubble Sort-Verfahren liegt dann vor, wenn die Liste bereits sortiert vorgegeben ist. Dann ist die **for**-Schleife nur ein einziges Mal zu durchlaufen und damit

$$\boldsymbol{C}_{min}(N) = N - 1 \quad \text{und} \quad \boldsymbol{M}_{min}(N) = 0$$

Der ungünstigste Fall liegt vor, wenn die Liste in absteigender Folge sortiert vorgegeben wird. Dann sind *N-1* Durchläufe durch die **do**-Schleife notwendig, um das kleinste Element nach vorne zu bringen und ein weiterer, der feststellt, daß nichts mehr zu vertauschen ist. Der *i*-te Durchlauf erfordert *N-i* Vergleiche für *i=1,2, ..., N-1* und der letzte Durchlauf noch einmal einen Vergleich, d.h. wir haben insgesamt

| **do**-Läufe für: | **for**-Schleifen mit: |
|---|---|
| i = N | j = 1, ... , N-1 |
| i = N-1 | j = 1, ... , N-2 |
| i = N-2 | j = 1, ... , N-3 |
| ..... | ..... |
| i = 2 | j = 1 |

sowie einen letzten Durchlauf der **do**-Schleife für `i=1`, der keine Vertauschung mehr erfordert und nur das Verfahren beendet. Es ist daher:

$$\boldsymbol{C}_{max}(N) = 1 + \sum_{i=1}^{N-1}(N-i) = 1 + \frac{N(N-1)}{2} = \boldsymbol{O}\left(N^2\right)$$

Bis auf den letzten Durchlauf sind pro Vergleich drei Item-Transporte notwendig, d.h.

$$\boldsymbol{M}_{max}(N) = 3 * \frac{N(N-1)}{2} = \boldsymbol{O}\left(N^2\right)$$

Auch für den durchschnittlichen Aufwand kann man zeigen:

$$\boldsymbol{C}_{av}(N) = \boldsymbol{M}_{av}(N) = \boldsymbol{O}\left(N^2\right)$$

Wir entnehmen diesen Abschätzungen, daß das Bubble Sort-Verfahren sowohl in Bezug auf die Schlüssel-Vergleiche als auch in Bezug auf die Item-Transporte quadratischen Aufwand erfordert. Es verhält sich im Vergleich zum Insertion Sort-Verfahren daher wesentlich ungünstiger, wenn die Item-Transporte ins Gewicht fallen.

**Bubble Sort**

| | | |
|---|---|---|
| Speicherbedarf | : | in-situ-Verfahren |
| Stabilität | : | ist gewährleistet |
| Datenstruktur | : | Array, Liste |
| Vorsortierung | : | globale Vorsortierung wirkt sich positiv aus. |
| Laufzeitverhalten | : | $\boldsymbol{C}_{min}(N) = \boldsymbol{O}(N)$<br>$\boldsymbol{M}_{min}(N) = 0$<br>$\boldsymbol{C}_{max}(N) = \boldsymbol{M}_{max}(N) = \boldsymbol{C}_{av}(N) = \boldsymbol{M}_{av}(N) = \boldsymbol{O}\left(N^2\right)$ |
| Anwendung | : | nur innerhalb anderer Verfahren (z.B. Quicksort) |

# 4.3 Spezielle Sortiermethoden

Im Gegensatz zu den elementaren machen die speziellen Sortierverfahren Annahmen über die Datenstruktur, in der die Items abgespeichert sind oder über die Struktur der Schlüssel. Sie besitzen daher in der Regel auch eine niedrigere Komplexität. Wir betrachten in diesem Abschnitt vier bekannte Methoden dieser Klasse:

- das Quicksort-Verfahren,
- das Heap-Sort-Verfahren,
- verschiedene Merge-Sort-Verfahren und
- einige Radix-Sort-Verfahren.

## 4.3.1 Quicksort

Diese 1962 von C.A.R. Hoare veröffentlichte Methode ist auch als "Partition Exchange"-Verfahren oder als "Vertauschen über große Entfernungen" bekannt. Sie wird allgemein als das schnellste Sortierverfahren für zufällig verteilte Items betrachtet. Seine Laufzeitkomplexität beträgt im Mittel $\mathbf{O}(N * \log(N))$ Schritte.

Dem Verfahren liegt das Divide et Impera-Prinzip zugrunde: Man zerlegt die Original-Liste $\boldsymbol{L}$ in zwei Teillisten $\boldsymbol{L_1}$ und $\boldsymbol{L_2}$ , dabei enthält $\boldsymbol{L_1}$ alle „kleinen“ und $\boldsymbol{L_2}$ alle „großen“ Items. Beide Teil-Listen können dann getrennt sortiert werden nach dem selben Schema.

### Methode

- Wähle aus der Originalliste $\boldsymbol{L} = \langle A_1, A_2, \ldots, A_N \rangle$ ein Pivot-Element $A_p$ aus und zerlege damit $\boldsymbol{L}$ in zwei disjunkte Teillisten $\boldsymbol{L_1}$ und $\boldsymbol{L_2}$:
  - $\boldsymbol{L_1}$ enthält nur Items $A \in \boldsymbol{L} - \{A_p\}$ mit $A \le A_p$
  - $\boldsymbol{L_2}$ enthält nur Items $A \in \boldsymbol{L} - \{A_p\}$ mit $A \ge A_p$ .
- Sortiere $\boldsymbol{L_1}$ und $\boldsymbol{L_2}$ getrennt nach dem selben Schema. Entstehen dabei Teillisten mit einem Item, so sind diese natürlich sortiert.
- Setze die sortierte Gesamtliste $\boldsymbol{L_s}$ zusammen aus den sortierten Teillisten $\boldsymbol{L_1}$ und $\boldsymbol{L_2}$ sowie dem Pivot-Element $A_p$ :

$$\boldsymbol{L} = \langle \boldsymbol{L_1}, A_p, \boldsymbol{L_2} \rangle .$$

Dabei haben wir noch zwei wichtige Aspekte offen gelassen: die Kriterien zur Auswahl des Pivot-Elements und das Vorgehen bei der Zerlegung in Teillisten.

### Zur Wahl des Pivot-Elements

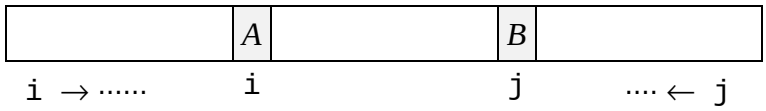
Von der Auswahl eines Pivot-Elements hängt die Größe der Teillisten $\boldsymbol{L_1}$ und $\boldsymbol{L_2}$ ab und damit der Aufwand für den gesamten Sortiervorgang. Zur Bestimmung von $A_p$ werden in der Praxis hauptsächlich heuristische Methoden benutzt, z.B.

- $A_p$ wird der ersten oder letzten Position der Liste L entnommen.
- $A_p$ wird zufällig ausgewählt.
- $A_p$ wird als Medianwert von drei Listenelementen gewählt, z.B. von $A_1$, $A_{N/2}$ und $A_N$ .

### Die Bestimmung der Teillisten

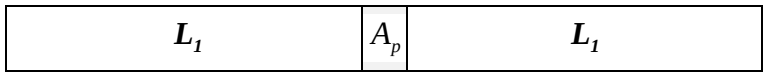
Wir gehen davon aus, daß die Liste $L$ in einem Feld `A[1..N]` gespeichert ist und wollen die Zerlegung in die beiden Teillisten $L_1$ und $L_2$ so durchführen, daß $L_1$ im vorderen Teil von `A` entsteht und $L_2$ im hinteren. Dazu arbeiten wir mit zwei Zeigern `i` und `j`.

Den Zeiger `i` lassen wir von links solange nach rechts wandern, bis erstmals ein Item $A$ angetroffen wird mit $A > A_p$ , das also nicht $L_1$ zugeordnet werden darf. Danach lassen wir den Zeiger `j` solange von rechts her laufen, bis erstmals ein Item $B$ angetroffen wird mit $B < A_p$ ist, das also nicht $L_2$ zugeordnet werden darf:




In dieser Situation vertauschen wir die beiden Listenelemente, auf die `i` und `j` zeigen. Sie wandern damit in die „richtige“ Teilliste.

Wir lassen nun `i` und `j` weiter gegeneinander laufen und vertauschen immer dann, wenn wir ein Item-Paar antreffen, das den „falschen“ Teillisten angehört. Sobald sich `i` und `j` überholen, vertauschen wir das Pivot-Item mit dem Item auf der Position `i`. Damit hat $A_p$ seinen endgültigen Platz erreicht: Links von $A_p$ steht die Teilliste $L_1$ und rechts davon die Teilliste $L_1$, die wir nun beide nach dem selben Schema weiterbearbeiten:




### Beispiel

Wir gehen von der folgenden unsortierten Liste aus und benutzen zunächst `A[N]=23` als Pivot-Element (mit einem Raster unterlegt):

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 33 | 8 | 3 | 25 | 17 | 27 | 11 | 81 | 5 | 23 |
| i | | | | | | | | | j |

Wenn wir `i` und `j` solange laufen lassen, bis zum ersten Mal vertauscht werden muß, erhalten wir die folgende Situation:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 33 | 8 | 3 | 25 | 17 | 27 | 11 | 81 | 5 | 23 |
| i | | | | | | | | j | |

Wir vertauschen `A[i]` mit `A[j]` und erhalten

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 8 | 3 | 25 | 17 | 27 | 11 | 81 | 33 | 23 |
| | i | | | | | | j | | |

Wir lassen nun `i` und `j` erneut laufen, bis zu vertauschen ist:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 8 | 3 | 25 | 17 | 27 | 11 | 81 | 33 | 23 |
| | | | i | | | j | | | |

Nach der Vertauschung von `A[i]` mit `A[j]` ergibt sich:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 8 | 3 | 11 | 17 | 27 | 25 | 81 | 33 | 23 |
| | | | | i | j | | | | |

Wenn nun i und j noch einmal laufen, dann überholen sie sich:

| 5 | 8 | 3 | 11 | 17 | 27 | 25 | 81 | 33 | 23 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | j | i | | | | |

Schließlich vertauschen wir `A[i]` mit dem Pivot-Element `A[N]`:

| 5 | 8 | 3 | 11 | 17 | 23 | 25 | 81 | 33 | 27 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | j | i | | | | |

Jetzt hat das erste Pivot-Element mit dem Wert *23* seine endgültige Position erreicht. Davor stehen die „kleineren“ Items der Teilliste $L_1$ und dahinter die "größeren" Items der Teilliste $L_2$. Im folgenden ist vom weiteren Verlauf des Verfahrens nur noch die Situation nach der Aufspaltung in Teillisten gezeigt. Items, die bereits ihren endgültigen Platz gefunden haben sind mit einem Raster unterlegt.

| 5 | 8 | 3 | 11 | 17 | 23 | 25 | 81 | 33 | 27 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 8 | 3 | 11 | 17 | 23 | 25 | 27 | 33 | 81 |
| 5 | 8 | 3 | 11 | 17 | 23 | 25 | 27 | 33 | 81 |
| 3 | 8 | 5 | 11 | 17 | 23 | 25 | 27 | 33 | 81 |
| 3 | 5 | 8 | 11 | 17 | 23 | 25 | 27 | 33 | 81 |

### C - Routine

Die folgende C-Routine stellt eine rekursive Implementierung des Quicksort-Verfahrens dar:

```
void QuickSort (int *A, int left, int right)
{  int  i, j;                     // Laufvariablen
   int  item;                     // Hilfsfeld zum Vertauschen
   int  pivot;                    // Pivot-Element

   if (right>left)
   {  pivot = A[right];           // Pivot-Element festhalten
      i = left-1, j = right;      // Laufzeiger initialisieren
      for (;;)
①     {  while (A[++i] < pivot);  // von links "grosses" Item suchen
②        while (A[--j] > pivot);  // von rechts "kleines" Item suchen
         if (i >= j) break;       // Abbruch,wenn sich Zeiger überholen
         item = A[i];             // grosses u. kleines El. vertauschen
         A[i] = A[j];
         A[j] = item;
      }
      A[right] = A[i];            // Pivotelement auf endgültigen Platz
      A[i] = pivot;
      QuickSort (A,left, i-1  );  // Teil-Listen getrennt sortieren
      QuickSort (A,i+1,  right);
   }
}
```

Das Quicksort-Verfahren ist sehr empfindlich gegen minimale Programm-Modifikationen. Einige Besonderheiten der oben angegebenen Implementierung sind:

- Die **`while`**-Schleife ① muß mit Vergleichen der Form < arbeiten, damit sie spätestens beim Pivot-Element der aktuellen Teil-Listen abbricht, das an der rechtesten Position steht.
- Die **`while`**-Schleife ② bricht beim ersten Aufruf der Prozedur nur dann sicher ab, wenn vorher auf Position `A[0]` ein Stopper-Item abgelegt wurde, dessen Schlüssel kleiner als alle in der Liste ***L*** vorkommenden Schlüssel ist.

  Vor dem ersten Aufruf muß `A[0]` daher mit einem Stop-Wert besetzt werden, der kleiner als der kleinstmögliche Schlüssel ist.

  In diesem Fall terminiert die Schleife ② auch für alle weiteren Aufrufe von Quicksort: `A[0]` ist weiterhin Stopper-Element für die linkeste Teilliste. Für alle anderen Teil-Listen wirkt das unmittelbar vor ihnen stehende Pivot-Element als Stopper-Item.

### Analyse

Die erste Teilungsphase für eine Liste der Länge *N* benötigt *(N-1)* Schlüsselvergleiche in den Zeilen ① und ② . Die nächste Phase benötigt zur Aufspaltung ihrer Teil-Listen zusammen *(N-2)* Schlüsselvergleiche, usw.

Der ungünstigste Fall mit einer maximalen Zahl von Phasen liegt vor, wenn bei jeder Teilung eine leere Liste entsteht und die andere außer dem Pivot-Element den Rest enthält. Damit erhalten wir für die Maximalzahl von Vergleichen:

$$\boldsymbol{C}_{max}(N) = (N-1) + (N-2) + \ldots + 1 = \boldsymbol{O}(N^2)$$

Im ungünstigsten Fall müssen für jeden dieser Vergleiche die Elemente paarweise miteinander vertauscht werden. Dies erfordert daher

$$\boldsymbol{M}_{max}(N) = 3 * [(N-1) + (N-2) + \cdots] = \boldsymbol{O}(N^2)$$

Transportoperationen für Items.

Der optimale Fall liegt dann vor, wenn die Anzahl der Teilungsphasen minimal wird. Dies trifft zu, wenn die entstehenden Teillisten stets gleich groß werden. Dann ergeben sich nämlich $ld(N)$ Teilungsphasen, d.h. die minimale Anzahl von Vergleichen ist in diesem Fall

$$\boldsymbol{C}_{min}(N) = \boldsymbol{O}(N * \log(N))$$

Um den mittleren Sortieraufwand zu bestimmen, setzen wir voraus, daß alle Schlüssel verschieden und zufällig verteilt sind. Durch die Aufteilungsschritte bleibt die zufällige Verteilung innerhalb der Teillisten gewahrt.

Wenn wir eine Liste mit $N$ Elementen an der Position $i$ aufteilen, dann beträgt die notwendige mittlere Zahl von Vergleichen

$$(N-1) + A_{i-1} + A_{N-i}$$

Dabei ist $A_k$ der mittlere Sortieraufwand zum Sortieren einer Liste der Länge *k*. Um den mittleren Aufwand für die Liste der Länge $N$ zu erhalten, mitteln wir über alle möglichen Aufteilungspositionen $i$ und erhalten:

$$A_N = \frac{1}{N} * \sum_{1}^{N} [(N-1) + A_{i-1} + A_{N-i}]$$

$$A_N = (N-1) + \frac{1}{N} * \sum_{1}^{N} A_{i-1} + \frac{1}{N} * \sum_{1}^{N} A_{N-i}$$

$$A_N = (N-1) + \frac{1}{N} * \sum_{0}^{N-1} A_i + \frac{1}{N} * \sum_{0}^{N-1} A_i$$

$$A_N = (N-1) + \frac{2}{N} * \sum_{0}^{N-1} A_i$$

Zur Lösung dieser Rekursionsgleichung machen wir den gleichen Ansatz für $A_{N-1}$:

$$A_{N-1} = (N-2) + \frac{2}{N-1} * \sum_{0}^{N-2} A_i$$

Wir formen beide Ansätze um zu

$$NA_N = N(N-1) + 2 * \sum_{0}^{N-2} A_i + 2A_{N-1}$$

$$(N-1)A_{N-1} = (N-1)(N-2) + 2 * \sum_{0}^{N-2} A_i$$

und bilden ihre Differenz

$$NA_N - (N-1)A_{N-1} = N(N-1) - (N-1)(N-2) + 2 * A_{N-1}$$

Die Differenz formen wir weiter um:

$$NA_N = 2(N-1) + (N+1)A_{N-1}$$

$$\frac{1}{N+1} A_N = \frac{1}{N} A_{N-1} + \frac{2(N-1)}{N(N+1)}$$

Mit der Substitution

$$Z_N = \frac{1}{N+1} A_N$$

und der Aufspaltung

$$\frac{2(N-1)}{N(N+1)} = \frac{4}{(N+1)} - \frac{2}{N}$$

erhalten wir daraus:

$$Z_N = Z_{N-1} + \frac{4}{N+1} - \frac{2}{N}$$

Als Lösung dieser Rekursionsgleichung in geschlossener Form erhalten wir

$$Z_N = 2 * \sum_{1}^{N} \left[ \frac{2}{i+1} - \frac{1}{i} \right]$$

Durch einfache Umrechnung erhalten wir daraus nun

$$Z_N = 2 * \sum_{2}^{N+1} \frac{2}{i} - 2 * \sum_{1}^{N} \frac{1}{i}$$

$$Z_N = 2 * \sum_{1}^{N} \frac{1}{i} + \frac{4}{N+1} - 4$$

$$Z_N = 2 * \sum_{1}^{N} \frac{1}{i} - \frac{4N}{N+1}$$

Aus der harmonischen Summenformel

$$H_L = \sum_{1}^{N} \frac{1}{i} \quad = \ln(N) + Eulerkonstante + \text{Re}\, stterme$$

erhalten wir $Z_N = \boldsymbol{O}((\log(N))$

und $A_N = (N+1) * Z_N = \boldsymbol{O}(N \log(N))$

Für die mittlere Komplexität des Quicksort-Verfahrens gilt also:

$$\boldsymbol{C_{av}}(N) = \boldsymbol{M_{av}}(N) = \boldsymbol{O}(N \log(N))$$

### Nicht-Rekursives Quicksort-Verfahren

Die oben dargestellte rekursive Implementierung des Quicksort-Verfahrens ist in Bezug auf die Formulierung des Algorithmus sehr elegant. Sie hat jedoch noch Nachteile:

- Die rekursiven Aufrufe implizieren einen relativ hohen Laufzeit-Overhead.
- Im ungünstigsten Fall wächst der Laufzeitkeller, in dem die Rekursiven Aufrufe verwaltet werden, linear mit der Länge der Liste.

Wir betrachten daher noch eine weitere Implementierung, die statt der rekursiven Aufrufe eine explizite Verwaltung der Teil-Listen durchführt und damit diese Nachteile vermeidet:

- Die Zerlegung in Teil-Listen wird wie bisher mit Hilfe eines Pivot-Elements durchgeführt. Die Pivot-Elemente finden dabei stets sofort ihren endgültigen Platz.
- Nach jeder Zerlegung werden die Grenzen der größeren Teil-Liste in einen Stack gelegt. Dieser kann daher maximal $\boldsymbol{O}((\log(N))$ Einträge groß werden.
- Die kleinere Teil-Liste wird sofort weiter zerlegt, wodurch der zweite rekursive Aufruf (End-Rekusion) überflüssig wird.

### C-Routine

```
void QuickSortNr (int *A, int n)
{
  int  i, left, right;           // Laufvariablen

  StackInit();                   // Stack f. Teillisteninitialisieren
  left  = 1;                     // linke  Feldgrenze
  right = n;                     // rechte Feldgrenze
  A[0]  = -MAXINT;               // Stopper-Element auf Position 0 !
  for (;;)
  {
    while (right > left)         // Abbruchkriterium: Listenlänge <= 1
    {
      i = Partition (A, left, right);  // Liste in 2 Teile zerlegen
      if (i-left > right-i)
      {
        push(left);  push(i-1); // größere Teil-Liste in den Stack
        left = i+1;             // größere Teil-Liste in den Stack
      }
      else
      {
        push(i+1); push(right); // größere Teil-Liste in den Stack
        right = i-1;            // größere Teil-Liste in den Stack
      }
    }

    if (StackEmpty()) break;     // fertig !
    right = pop ();              // Teil-Liste vom Stack holen
    left  = pop ();
  }
}

int  Partition  (int *A, int left, int right)
{
  int  i, j;                   // Laufvariablen
  int  item;                   // Hilfsfeld zum Vertauschen
  int  pivot;                  // Pivot-Element

  pivot = A[right];            // Pivot-Element festhalten
  i = left-1, j = right;       // Laufzeiger initialisieren
  for (;;)
  {
    while (A[++i] < pivot);    // von links  her "grosses" Item suchen
    while (A[--j] > pivot);    // von rechts her "kleines" Item  suchen
    if (i >= j) break;         // Abbruch, wenn sich Zeiger überholen
    item = A[i];               // grosses und kleines Item vertauschen
    A[i] = A[j];
    A[j] = item;
  }
  A[right] = A[i] ;            // Pivot-Element auf endgültigen Platz
  A[i] = pivot;
  return(i);
}
```

### Weitere Optimierung des Quicksort-Verfahrens

Zusätzliche Verbesserungen des Quicksort-Verfahrens setzen an den folgenden Stellen an:

- Wahl des Pivot-Elements: Statt des rechtesten Elements der Teil-Listen kann man z.B. zufäl- lig eines wählen oder von drei zufällig gewählten das mitlere verwenden. Damit werden systematisch ungünstige Zerlegungen sehr unwahrscheinlich.
- Sonderbehadlung kleiner Restlisten: Diese können besser mit Insertion Sort sortiert werden. Eine elegante Möglichkeit für das rekursive Quicksort-Verfahren besteht darin, die Abfrage `if (r>l)` am Anfang der Prozedur in `r-l>M` abzuändern. Dabei entsteht eine fast sortierte Liste, die sich danach effizient mit Insertion Sort nachbehandeln läßt.

**Quicksort**

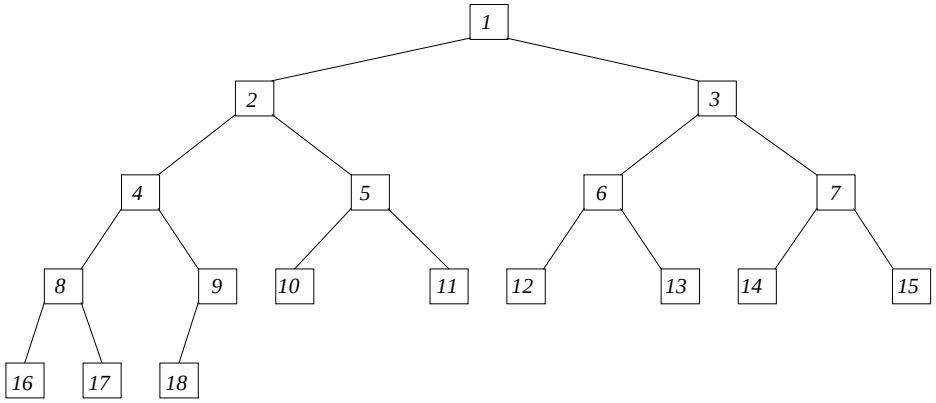
| | | |
|---|---|---|
| Speicherbedarf | : | in-situ-Verfahren, aber zusätzlicher Speicher zur Verwaltung der Teilaufträge notwendig (maximal $\boldsymbol{O}(N)$ Speicherplätze). |
| Stabilität | : | ist **nicht** gewährleistet |
| Datenstruktur | : | Array |
| Vorsortierung | : | globale Vorsortierung wirkt sich eher negativ aus. |
| Laufzeitverhalten | : | $\boldsymbol{C}_{av}(N) = \boldsymbol{C}_{min}(N) = \boldsymbol{O}(N \log(N))$ |
| | | $\boldsymbol{C}_{max}(N) = \boldsymbol{O}(N^2)$ |
| | : | $\boldsymbol{M}_{av}(N) = \boldsymbol{M}_{min}(N) = \boldsymbol{O}(N \log(N))$ |
| | | $\boldsymbol{M}_{max}(N) = \boldsymbol{O}(N^2)$ |
| Anwendung | : | schnellstes internes Sortier-Verfahren |

## 4.3.2 Heap Sort

Das Heap Sort-Verfahren zeichnet sich dadurch aus, daß es die einzige In-situ-Methode ist, deren Laufzeitkomplexität auch im schlechtesten Fall nicht größer wird als das theoretische Optimum für zufällig angeordnete Ausgangslisten, nämlich $\boldsymbol{O}(N \log(N))$. Bei allen anderen bisher behandelten Verfahren muß man bei ungünstiger Anordnung der Ausgangsliste mit $\boldsymbol{O}(N^2)$ Verfahrensschritten rechnen.

### Halden (Heaps)

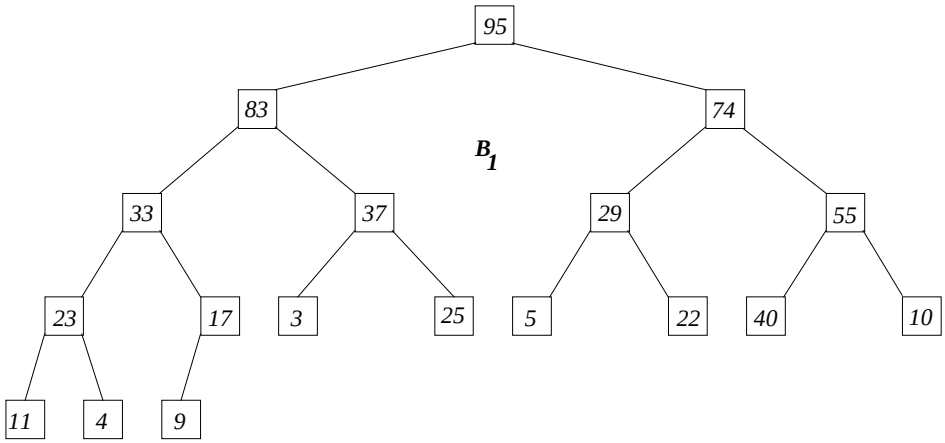
Die zu sortierende Liste $\boldsymbol{L} = \langle A_1, A_2, \ldots, A_N \rangle$ wird als ein binärer Baum in einem Array abgespeichert, so daß man auf die Items gezielt über eine einfache Indexrechnung zugreifen kann. Wir nehmen eine Numerierung der Knoten des Baums in der folgenden Weise vor:




Beim Abspeichern der Liste in einem Array wird die Knotennummer als Array-Index verwendet. Der obigen Darstellung entnehmen wir:

- Die Nummer eines Knotens ist gleichzeitig die Anzahl der Elemente im Baum bis zu dieser Position einschließlich.
- Die beiden Söhne eines Knotens mit der Nummer *i* besitzen die Nummern $2*i$ und $2*i+1$.
- der Vaterknoten zum Knotens $i$ hat die Nummer $\left\lfloor \frac{i}{2} \right\rfloor$.

Ein binärer Baum, wird als eine Halde (Heap) bezeichnet, wenn jeder Knoten größer als seine Söhne ist. In dem folgenden Baum $\boldsymbol{B_1}$ ist diese Bedingung über die Anordnung der Elemente erfüllt:



### Prinzip des Heap Sort-Verfahrens

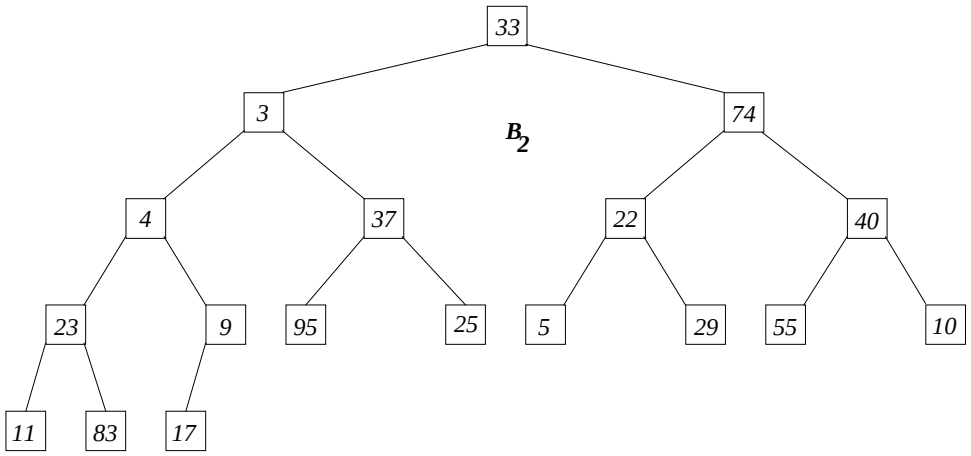
Die Idee des Heap Sort-Verfahren beruht darauf, daß zunächst die Liste als Heap angeordnet wird. Damit steht das größte Element an der Wurzel des Baumes zur Verfügung. Die folgenden beiden Schritte werden nun solange wiederholt, bis der Heap leer ist:

- Übertrage das Wurzel-Element vom Heap in die Ausgabeliste.
- Stelle für die Restliste wieder die Heap-Bedingung her.

Damit erhält man die Elemente der Liste in absteigender Ordnung. Durch weitere organisatorische Maßnahmen kann man dafür sorgen, daß für die Ausgabeliste kein zusätzlicher Speicher benötigt wird und daß die Elemente aufsteigend angeordnet sind.

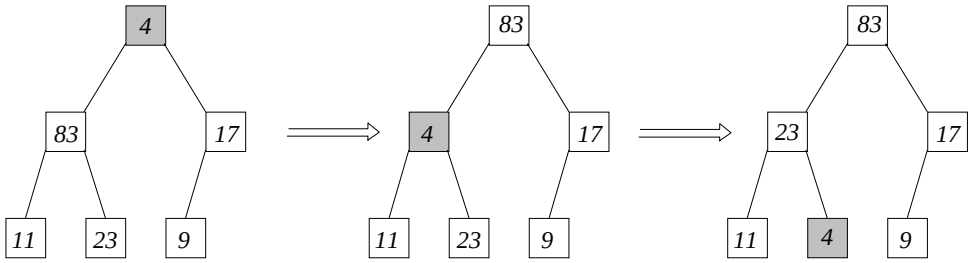
### Erstmaliges Herstellen der Heap-Bedingung

Einen binären Baum, der als Liste $\boldsymbol{L} = \langle A_1, A_2, \ldots, A_N \rangle$ vorliegt, z.B. den folgenden Baum $\boldsymbol{B_2}$




ordnen wir zu einem Heap um, indem wir von der untersten Ebene zur Wurzel hin Teilbäume zu Heaps umordnen. Die Teilbäume der untersten Ebene können wir zu Heaps machen, indem wir das größte Element an die Spitze stellen


Wenn wir auf den k untersten Ebenen des Baumes bereits alle Teilbäume zu Heaps umgeordnet haben, dann können wir die Heap-Bedingung auf der nächst höheren Ebene herstellen, indem wir das Wurzel-Element eines Teilbaumes solange in Richtung des jeweils größten Sohnes nach unten sinken lassen lassen, bis kein größerer Sohn mehr vorliegt. Dies zeigt die folgende Abbildung für den Wurzelknoten mit dem Item 4:




Durch Fortsetzung dieses Verfahrens bis zur Wurzel-Ebene erhalten wir schließlich einen Heap und zwar den bereits am Anfang dieses Abschnitts abgebildeten Baum $B_1$ .

Die entscheidende Idee der Heap-Sort-Methode ist es also, daß aus einem Baum, der bis auf das Wurzel-Element bereits der Heap-Bedingung genügt, ein vollständiger Heap durch das Versinken des Wurzelelements entsteht.

### Sortieren eines Heap

Wenn der Heap ein erstes Mal hergestellt ist, dann ist der eigentliche Sortiervorgang einfach:

- Wir vertauschen das letzte Element des Baumes mit der Wurzel. Dadurch kommmt das größte Element der Liste auf den letzten Platz.
- Danach lassen wir das neue Wurzel-Element in den Heap einsinken. Dadurch wird die Restliste wieder zu einem Heap.

Dies wiederholen wir so lange, bis keine Restliste mehr übrig bleibt.

### Verfahren

Wir führen den oben mit Hilfe von Bäumen veranschaulichten Prozeß nun mit einer Liste durch, die in einem Array gespeichert vorliegt. Auf dieser Datenstruktur basiert der endgültige Algorithmus. Die vorgegebene Liste für den Baum $B_2$ hat die Form:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 3 | 74 | 4 | 37 | 22 | 40 | 23 | 9 | 95 | 25 | 5 | 29 | 55 | 10 | 11 | 83 | 17 |

Der letzte Teilbaum auf der untersten Schicht besitzt die Knoten-Indizes 9 und 18. Wenn wir den größeren der beiden Knoten zum Vaterknoten machen, erhalten wir die Liste:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 3 | 74 | 4 | 37 | 22 | 40 | 23 | 17 | 95 | 25 | 5 | 29 | 55 | 10 | 11 | 83 | 9 |

Außerdem enthält die unterste Schicht den Teilbaum mit den Knoten-Indizes 8, 16 und 17:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 3 | 74 | 4 | 37 | 22 | 40 | 23 | 17 | 95 | 25 | 5 | 29 | 55 | 10 | 11 | 83 | 9 |

Wenn wir den größeren der beiden Söhne zum Vaterknoten machen, erhalten wir die Liste:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 3 | 74 | 4 | 37 | 22 | 40 | 83 | 17 | 95 | 25 | 5 | 29 | 55 | 10 | 11 | 23 | 9 |

Auf der nächsten Ebene haben wir die Teilbäume mit den Knoten-Indizes <7,14,15>, <6,12,13>, <5,10,11> und <4,8,9,16,17,18> zu Heaps umzuordnen. Umordnen des ersten dieser Teilbäume ergibt die folgende Liste:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 3 | 74 | 4 | 37 | 22 | 55 | 83 | 17 | 95 | 25 | 5 | 29 | 40 | 10 | 11 | 23 | 9 |

Umordnen der zweiten Teilliste liefert analog:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 3 | 74 | 4 | 37 | 29 | 55 | 83 | 17 | 95 | 25 | 5 | 22 | 40 | 10 | 11 | 23 | 9 |

Umordnen der dritten Teil-Liste liefert:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 3 | 74 | 4 | 95 | 29 | 55 | 83 | 17 | 37 | 25 | 5 | 22 | 40 | 10 | 11 | 23 | 9 |

Umordnen der letzten Teilliste dieser Ebene ergibt schließlich:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 3 | 74 | 83 | 95 | 29 | 55 | 23 | 17 | 37 | 25 | 5 | 22 | 40 | 10 | 11 | 4 | 9 |

Auf der nächsten Ebene des Baumes müssen wir die Heap-Bedingung für die Teilbäume mit den folgenden Indizes erreichen

<3,6,7,12,13,14,15> und <2,4,5,8,9,10,11,16,17,18>.

Der erste Teilbaum genügt bereits der Heap-Bedingung.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 3 | 74 | 83 | 95 | 29 | 55 | 23 | 17 | 37 | 25 | 5 | 22 | 40 | 10 | 11 | 4 | 9 |

Wenn wir das Wurzel-Element des zweiten Teilbaums versinken lassen, dann erhalten wir:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 95 | 74 | 83 | 37 | 29 | 55 | 23 | 17 | 3 | 25 | 5 | 22 | 40 | 10 | 11 | 4 | 9 |

Schließlich müssen wir durch Versinken des obersten Wurzel-Elements die Heap-Bedingung endgültig für den ganzen Baum herstellen und erhalten:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 95 | 83 | 74 | 33 | 37 | 29 | 55 | 23 | 17 | 3 | 25 | 5 | 22 | 40 | 10 | 11 | 4 | 9 |

Nun beginnt der eigentliche Sortierprozeß. Wir vertauschen die Wurzel mit dem letzten Element der Liste und lassen die neue Wurzel versinken. Als Ergebnis erhalten wir die folgende Liste:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 83 | 37 | 74 | 33 | 25 | 29 | 55 | 23 | 17 | 3 | 9 | 5 | 22 | 40 | 10 | 11 | 4 | 95 |

Darin ist der bereits endgültig sortierte Bereich des Array mit einem Raster unterlegt.

Wir wiederholen dasselbe für die jeweils um ein Element verkürzten Restlisten, solange bis das ganze Feld sortiert ist. Die Listenzustände jeweils nach dem Versinken sind dann:

| *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* | *15* | *16* | *17* | *18* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *74* | *37* | *55* | *33* | *25* | *29* | *40* | *23* | *17* | *3* | *9* | *5* | *22* | *4* | *10* | *11* | *83* | *95* |
| *55* | *37* | *40* | *33* | *25* | *29* | *11* | *23* | *17* | *3* | *9* | *5* | *22* | *4* | *10* | *74* | *83* | *95* |
| *40* | *37* | *29* | *33* | *25* | *22* | *11* | *23* | *17* | *3* | *9* | *5* | *10* | *4* | *55* | *74* | *83* | *95* |
| *37* | *33* | *29* | *23* | *25* | *22* | *11* | *4* | *17* | *3* | *9* | *5* | *10* | *40* | *55* | *74* | *83* | *95* |
| *33* | *25* | *29* | *23* | *10* | *22* | *11* | *4* | *17* | *3* | *9* | *5* | *37* | *40* | *55* | *74* | *83* | *95* |
| *29* | *25* | *22* | *23* | *10* | *5* | *11* | *4* | *17* | *3* | *9* | *33* | *37* | *40* | *55* | *74* | *83* | *95* |
| *25* | *23* | *22* | *17* | *10* | *5* | *11* | *4* | *9* | *3* | *29* | *33* | *37* | *40* | *55* | *74* | *83* | *95* |
| *23* | *17* | *22* | *9* | *10* | *5* | *11* | *4* | *3* | *25* | *29* | *33* | *37* | *40* | *55* | *74* | *83* | *95* |
| *22* | *17* | *11* | *9* | *10* | *5* | *3* | *4* | *23* | *25* | *29* | *33* | *37* | *40* | *55* | *74* | *83* | *95* |
| *17* | *10* | *11* | *9* | *4* | *5* | *3* | *22* | *23* | *25* | *29* | *33* | *37* | *40* | *55* | *74* | *83* | *95* |
| *11* | *10* | *5* | *9* | *4* | *3* | *17* | *22* | *23* | *25* | *29* | *33* | *37* | *40* | *55* | *74* | *83* | *95* |
| *10* | *9* | *5* | *3* | *4* | *11* | *17* | *22* | *23* | *25* | *29* | *33* | *37* | *40* | *55* | *74* | *83* | *95* |
| *9* | *4* | *5* | *3* | *10* | *11* | *17* | *22* | *23* | *25* | *29* | *33* | *37* | *40* | *55* | *74* | *83* | *95* |
| *5* | *4* | *3* | *9* | *10* | *11* | *17* | *22* | *23* | *25* | *29* | *33* | *37* | *40* | *55* | *74* | *83* | *95* |
| *4* | *3* | *5* | *9* | *10* | *11* | *17* | *22* | *23* | *25* | *29* | *33* | *37* | *40* | *55* | *74* | *83* | *95* |
| *3* | *4* | *5* | *9* | *10* | *11* | *17* | *22* | *23* | *25* | *29* | *33* | *37* | *40* | *55* | *74* | *83* | *95* |

Damit haben wir die Liste vollständig sortiert !

### C - Routine

Das Verfahren benutzt als Unterprogramm die Funktion Sinken, mit der das Ansinken eines Item
in einem Heap realisiert wird.

```
void HeapSort (int *A, int n)
{  int  item;                    // Hilfsfeld zum Vertauschen
   int  left, right;             // linke, rechte Grenze von Teilbäumen

   left  = ( n / 2 ) + 1;        // Zeige hinter den letzten Vaterknoten
   right = n;                    // zeige auf rechtes Listenende
   while ( left > 1 )            // Schleife zum Aufbau des Heap
   {  left--;                    // zeige auf den nächsten Vaterknoten
      Sinken (A,left, right);    // laß ihn absinken
   }
   while ( right > 1 )           // Sortiere jetzt den Heap
   {  item = A[left];            // tausche Wurzel mit letztem Item
      A[left] = A[right];        // in der Restliste
      A[right] = item;
      right--;                   // Restliste verkürzen
      Sinken (A,left, right);    // neues Wurzel-Item absinken lassen
   }
}
```

```
void Sinken ( int *A, int left, int right )
{
  int i, j;              // Laufzeiger
  int item;              // Hilfsfeld zum Vertauschen zweier Elemente

  i     = left;          // zeige zur Wurzel
  j     = 2 * i;         // zeige auf den ersten der Söhne
  item = A[i];           // sinkendes Item zwischenspeichern

  while ( j <= right )   // wiederhole Sinken
  {
    if ( j < right )     // den größeren der beiden Söhne feststellen
      if ( A[j] < A[j+1] )  j++;
    if ( item >= A[j] )  // richtiger Platz für das Item gefunden: i
      j = right + 1;     // setze Abbruchbedingung für while
    else
    {
      A[i] = A[j];       // laß Nachfolger aufsteigen
      i = j;             // zeige auf nächsten "leeren" Platz
      j = 2 * i;         // zeige auf den nächsten Sohn
    }
  }
  A[i] = item;           // Trage Item auf dem richtigen Platz ein
}
```

### Analyse

Zunächst betrachten wir den Aufwand zur Umordnung der Liste in einen Heap:

Es sei $2^{j-1} < N < 2^j$ , d.h. $j$ ist die Anzahl der Level des Baumes. Auf der Stufe $k$ gibt es dann maximal $2^{k-1}$ Items.

Die Anzahl der Vergleiche bzw. Item-Transporte beim Versinken eines Wurzelknotens auf der Stufe k des Baums ist proportional zu *j-k*. Damit erhalten wir für die Zahl Z der Operationen beim Aufbau der Heap-Struktur:

$$Z = \sum_{k=1}^{j-1} 2^{k-1}(j-k)$$

Mit der Substitution $i = j - k$ erhalten wir weiter:

$$Z = \sum_{i=1}^{j-1} i 2^{j-i-1}$$

$$Z = 2^{j-1} \sum_{i=1}^{j-1} \frac{i}{2^i} < N * 2$$

Damit ist der Aufwand zur Erreichung der Heap-Struktur $\boldsymbol{O}(N)$. Dabei wurde die Summen-Formel für eine spezielle binomische Reihe ausgenutzt, wenn darin $x = 1/2$ gesetzt wird:

$$\frac{1}{(1-x)^2} = 1 + 2x + 3x^2 + 4x^3 + \ldots \qquad \text{für } |x| < 1$$

Der Aufwand für das anschließende Sortieren ist von der Größenordnung $\boldsymbol{O}(N \log(n))$, denn jedes Listenelement ist einmal Wurzelknoten und kann dann um maximal $\log(N)$ Stufen versinken.

Der Maximal-Aufwand für das Heapsort-Verfahren beträgt also

$$\boldsymbol{C}_{max}(N) = \boldsymbol{M}_{max}(N) = \boldsymbol{O}(N \log(N))$$

Dies ist sogar das theoretische Optimum für alle möglichen Sortierverfahren bei zufllig verteileter Ausgangsliste.

**Heap Sort**

| | | |
|---|---|---|
| Speicherbedarf | : | in-situ-Verfahren |
| Stabilität | : | ist **nicht** gewährleistet |
| Datenstruktur | : | Array |
| Vorsortierung | : | In der beschriebenen Form hat sie keine Auswirkung. Es gibt aber Modifikationen, die sie nutzen, z.B. Smoooth Sort. |
| Laufzeitverhalten | : | $\boldsymbol{C}_{max}(N) = \boldsymbol{O}(N \log(N))$<br>$\boldsymbol{M}_{max}(N) = \boldsymbol{O}(N \log(N))$ |
| Anwendung | : | Einzige worst case-optimale in situ-Methode |

u

## 4.3.3 Merge Sort

Das Merge Sort-Verfahren geht von der intuitiven Vorstellung aus, daß es einfacher ist, zwei schon sortierte Teillisten zu einer sortierten Gesamtliste zusammenzuführen, als eine Gesamtliste ganz neu zu sortieren. Es wurde schon 1945 von J.v.Neumann vorgeschlagen und gehört damit zu den ältesten Methoden zum Sortieren auf Rechnern.

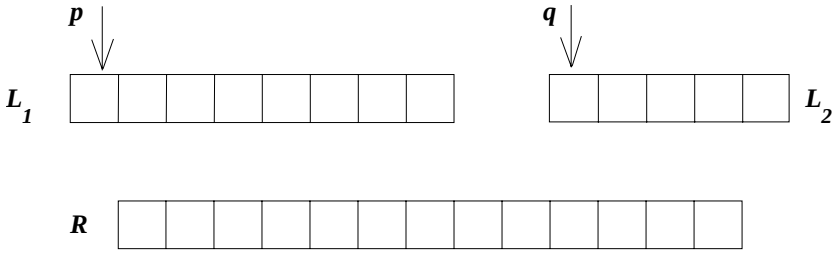
#### Prinzipielles Vorgehen

Der prinzipielle Ansatz des Merge Sort-Verfahrens ähnelt dem von Quicksort: Eine Gesamtliste wird etwa in der Mitte geteilt. Die beiden Teile werden durch rekursives Anwenden der Methode solange weiterbearbeitet bis sie sortiert sind. Danach bildet man aus ihnen eine Gesamtliste durch einen Verschmelzungsprozeß. Dieses explizite Verschmelzen ist notwendig, weil im Gegensatz zu Quicksort jede Teilliste sowohl "kleine" als auch "große" Items enthalten kann.

Es sind verschiedene Varianten des Verfahrens bekannt, von denen wir hier einige betrachten, die Listen im Arbeitsspeicher sortieren (interne Verfahren). Außerdem wird die Methode bei externen Sortierverfahren eingesetzt. Den Kern aller Merge Sort-Varianten bildet das Verschmelzen sortierter Teillisten. Wir betrachten daher vorab diesen Vorgang genauer:

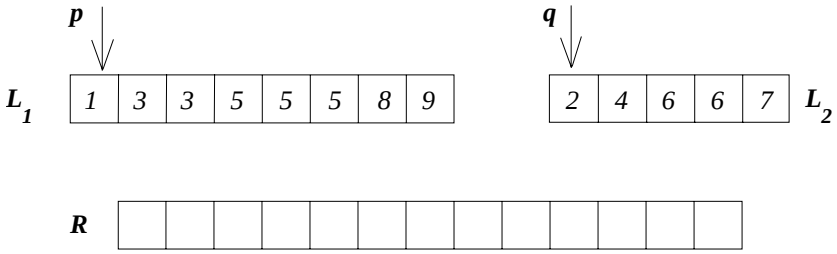
### 4.3.3.1 Verschmelzen sortierter Teillisten

Sind die Listen $L_1$ und $L_2$ bereits aufsteigend sortiert, so stellen wir an den Anfang jeder Liste einen Zeiger (***p*** bzw. ***q***). In einen zusätzlichen Speicherbereich ***R*** übertragen wir das kleinere der durch die Zeiger identifizierten Elemente und stellen dann diesen Zeiger weiter. Diesen Vorgang wiederholen wir solange, bis eine der beiden Listen leer ist. Dann kann die andere direkt in die Zielliste übernommen werden:

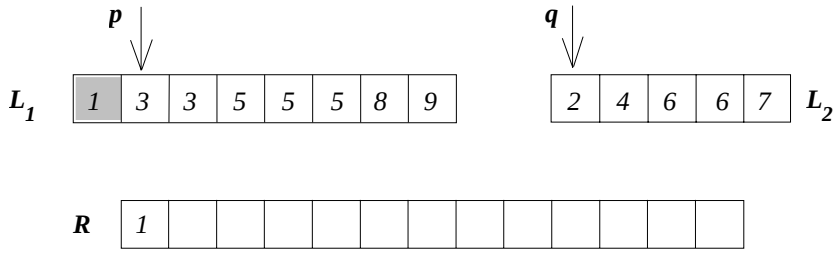



#### Beispiel:

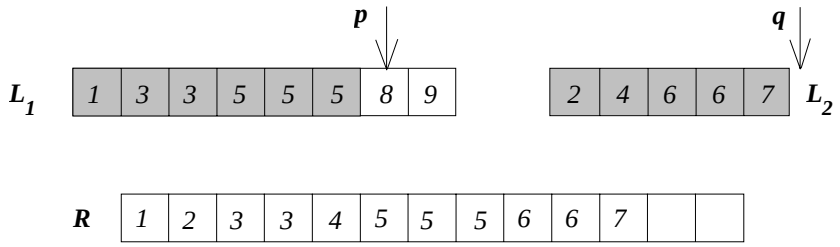
Ausgehend von der folgenden Situation:



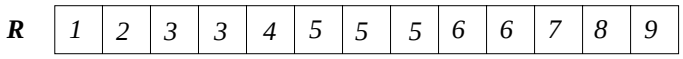

übernehmen wir zunächst das erste Element der linken Liste mit dem Schlüssel *1* in die Zielliste. Der linke Zeiger ***p*** wird eine Position weitergestellt und wir erhalten:




Als nächstes wird das erste Element der rechten Liste in die Resultatliste übernommen usf. Die rechte Teilliste wird schließlich zuerst leer und es ergibt sich:




In dieser Situation kann der Rest der linken Liste direkt übertragen werden und wir erhalten als Ergebnisliste:




### C - Routine

In der folgenden Misch-Routine werden Stooper-Elemente verwendet, um während der Schleifenkontrolle überflüssige Abfragen zu sparen.

```
void Merge ( int *A,                    // Resultatliste
             int *B, int lB, int rB,    // 1. Ausgangsliste
             int *C, int lC, int rC)    // 2. Ausgangsliste
{
   int p, q, r;                         // Laufzeiger

   p       = lB;                        // zeige auf erstes Item der Liste A
   q       = lC;                        // zeige auf erstes Item der Liste B
   B[rB+1] = MAXINT;                    // Stopper-Element für Liste A
   C[rC+1] = MAXINT;                    // Stopper-Element für Liste B

   for (r=1; r <= rB+rC; r++)           // Mischen
      A[r]  = (B[p] < C[q]) ?  B[p++] : C[q++];
}
```

#### Analyse

Das Verschmelzen zweier sortierter Listen bildet den Kern aller Merge Sort-Verfahren. Daher untersuchen wir den Aufwand dafür vorab:

Die Länge der beiden Listen seien $n_1$ und $n_2$. Die Mindestzahl von Vergleichen für das Verschmelzen beträgt $\min(n_1, n_2)$, nämlich dann, wenn zunächst nur aus der kleineren Liste Elemente in das Resultat übernommen werden müssen, bis diese leer ist. Die andere Teilliste wird in diesem Fall ohne weiteren Vergleich angehängt.

Der maximale Aufwand fällt dann an, wenn die Elemente abwechselnd aus beiden Listen in die Resultatliste übernommen werden. Er beträgt dann $n_1 + n_2 - 1$ Vergleiche

Die Anzahl der Item-Transporte in Merge beträgt $2(n_1 + n_2)$, denn es wird jedes Item einmal in die Zielliste $\boldsymbol{R}$ übernommen und zum Schluß wird $\boldsymbol{R}$ zurückgespeichert in den Originalbereich.

### 4.3.3.2 Rekursives 2-Wege Merge Sort-Verfahren

#### Methode

Eine Gesamtliste $\boldsymbol{L} = \langle A_1, A_2, \ldots, A_N \rangle$ zerlegen wir in zwei Teil-Listen

$$\boldsymbol{L_1} = \langle A_1, A_2, \ldots, A_m \rangle \text{ und } \boldsymbol{L_2} = \langle A_{m+1}, A_{m+2}, \ldots, A_N \rangle \quad \text{mit } m = \left\lceil 1/2 \right\rceil .$$

Die beiden Listen $\boldsymbol{L_1}$ und $\boldsymbol{L_2}$ sortieren wir rekursiv und verschmelzen sie danach zu einer sortierten Gesamtliste.

#### C - Routine

```
void MergeSort ( int *A, int left, int right)
{  int p, q, r, m;                  // Laufzeiger

   if (right > left)
   {
      m = (right+ left)/2;             // auf Mitte der Liste zeigen
      MergeSort (A, left, m);          // linke  Teil-Liste sortieren
      MergeSort (A, m+1, right);       // rechte Teil-Liste sortieren
                                       // beide  Teil-Listen zusammenmischen:
      for (p=m+1; p>left;  p--) B[p-1]          = A[p-1];
‹     for (q=m;   q<right; q++) B[right + m -q] = A[q+1];
›     for (r=left; r <=right; r++)
        A[r] = (B[p] < B[q]) ? B[p++] : B[q--];
   }
}
```

Man beachte, daß in der Schleife ‹ die sortierte Teil-Liste in umgekehrter Reihenfolge auf das Feld B gespeichert wird. Dadurch wird erreicht, daß bei der Misch-Schleife › die Zeiger p bzw. q auf dem größten Element der anderen Teil-Liste stehen, sobald ihre eigene erschöpft ist und dann die verbliebene Teil-Liste ganz nach A übernommen wird.

#### Analyse

Da die Aufteilung der Listen stets in der Mitte erfolgt, ist die Rekursionstiefe maximal $\log_2(N)$. Auf jeder dieser Stufen benötigen wir für alle Teillisten zusammen $\boldsymbol{O}(N)$ Vergleiche und $\boldsymbol{O}(N)$ Transporte. Damit beträgt der Gesamtaufwand für das Sortieren $\boldsymbol{O}(N\log(N))$.

Wie unter 4.3.3.1 schon gezeigt, ist diese Größenordnung sowohl im besten, wie im schlechtesten Fall erforderlich, so daß gilt:

$$\boldsymbol{C}_{av}(N) = \boldsymbol{C}_{min}(N) = \boldsymbol{C}_{max}(N) = \boldsymbol{O}(N\log(N))$$

$$\boldsymbol{M}_{av}(N) = \boldsymbol{M}_{min}(N) = \boldsymbol{M}_{max}(N) = \boldsymbol{O}(N\log(N))$$

Merge Sort ist daher, wie auch Heap Sort ein worst-case-optimales Verfahren. Es benötigt im Gegensatz zu Heap Sort aber zusätzlichen Speicher.

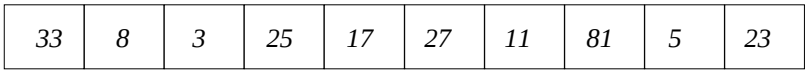
### 4.3.3.3 Bottom-Up Merge Sort-Verfahren

#### Methode

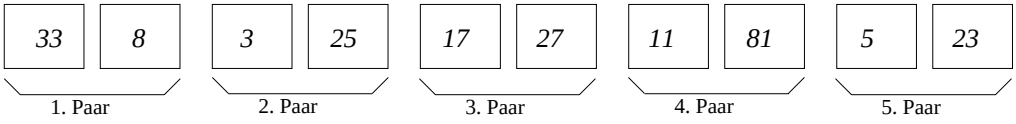
Während das rekursive Verfahren die Gesamtliste von "oben" her rekursiv teilt, geht die iterative Methode den umgekehrten Weg: Es werden zunächst benachbarte Teillisten der Länge *1*, danach der Länge *2* usf. verschmolzen zu sortierten Teil-Listen, bis eine sortierte Gesamtliste mit der vollen Länge $N$ vorliegt.

#### Beispiel

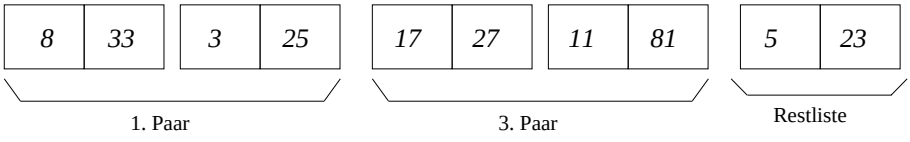
Wir gehen aus von der folgenden unsortierten Liste:



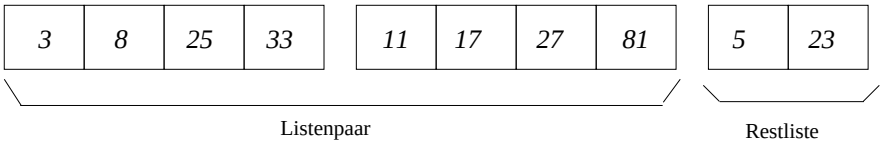

Zunächst betrachten wir je zwei benachbarte Teillisten der Länge 1. Diese sind per se sortiert. Die Listenpaare sind hier durch größere Abstände voneinander getrennt:




Durch Verschmelzen der Listenpaare erhalten wir neue Paare von Listen mit je zwei Items:




Das nächste Verschmelzen liefert zwei Listen mit je 4 Items und eine Restliste mit zweien:



Das nächste Verschmelzen liefert eine Liste mit 8 Items und eine Restliste:

| 3 | 8 | 11 | 17 | 25 | 27 | 33 | 81 |
|---|---|---|---|---|---|---|---|

| 5 | 23 |
|---|---|

Zuletzt wird die Liste der Länge 8 mit der Restliste verschmolzen. Dabei ergibt sich die vollständig sortierte Liste:

| 3 | 5 | 8 | 11 | 17 | 23 | 25 | 27 | 33 | 81 |
|---|---|---|---|---|---|---|---|---|---|

### C -Routine

```
void MergeSortBU ( int *A, int left, int right)
{  int p, q, c;                 // Laufzeiger
   int size;                    // aktuelle Länge der Teil-Listen

   size = 1;                    // mit Teil-Listen der Länge 1 beginnen
   while (size < right - left +1) // solange es mind 2 Teil-Listen gibt
   {  q = left - 1;             // Zeiger vor die Gesamtliste stellen
      while (q + size < right)  // solange es mind. 2 Teil-Listen gibt
      {  p = q+1;               // zeige auf nächstes Item der 1. Folge
         c = p + size -1;       // zeige auf letztes  Item der 1. Folge
         if (c + size <= right) // zeige auf rechten  Rand der 2. Folge
           q = c + size;
         else
           q = right;
         MergeT (A,p,c,q);      // Mische beide Teil-Listen
      }
      size = 2*size;            // nächst größere Listenlänge
   }
}

void MergeT ( int *A, int left, int center, int right)
{
   int p, q, i, k;              // Laufzeiger
   int AA[N+2];                 // Hilfsfeld für die Ziel-Liste

   p = left;                    // Zeiger auf 1. Item der linken  Liste
   q = center + 1;              // Zeiger auf 1. Item der rechten Liste
   k = left;                    // Zeiger auf 1. Position der Ziel-Liste
   while ((p <=center) && (q <= right))      // Teil-Listen mischen
   {  if (A[p] <= A[q]) AA[k++] = A[p++];
      else              AA[k++] = A[q++];
   }
                                             // Restliste übernehmen
   if (p > center)  for (i=q; i<= right; i++)  AA[k+i-q] = A[i];
   else             for (i=p; i<= center; i++) AA[k+i-p] = A[i];

   for (i=left; i<=right; i++) A[i] = AA[i];  // Resultat zurück nach A
}
```

### 4.3.3.4 Natürliches 2-Wege Merge Sort-Verfahren

#### Methode

Eine bessere Ausnutzung der Vorsortierung erreicht man dadurch, daß man nicht beginnend mit Listen der Länge *1* die Listenlänge immer wieder verdoppelt, sondern benachbarte Teillisten verwendet, die schon möglichst groß und dabei sortiert sind (*Runs*). Das folgende Programm implementiert diese Methode:

#### C - Routine

```
void MergeSortN ( int *A, int left, int right)
{ int p, q, c;                          // Laufzeiger
  do
  {
    q = left - 1;                       // q-Zeiger initialisieren
    while (q < right)                   // solange Runs vorhanden sind
    { p = q + 1;                        // linker Rand des 1. Run
      c = p;                            // Stelle Länge des 1. Run fest
      while ((c < right) && (A[c] <= A[c+1])) c++;
      if (c<right)                      // falls es weitere Items gibt:
      { q = c+1;                        // Stelle Länge des 2. Run fest
        while ((q < right) && (A[q] <= A[q+1])) q++;
        MergeT (A,p,c,q);
      }
      else
        q = c;              // Es gibt keine Elemente mehr für 2. Run
    }
  } while (p != left);      // solange wie p noch verändert wurde
}                           // d.h. bis die Gesamtliste ein Run ist
```

### 4.3.3.5 Zusammenfassung

Bei allen betrachteten Varianten der Merge Sort-Methode ergibt sich die gleiche Leistungs--Charakteristik im besten wie im schlechtesten Fall:

$$\boldsymbol{C}_{av}(N) = \boldsymbol{C}_{min}(N) = \boldsymbol{C}_{max}(N) = \boldsymbol{O}(N\log(N))$$

$$\boldsymbol{M}_{av}(N) = \boldsymbol{M}_{min}(N) = \boldsymbol{M}_{max}(N) = \boldsymbol{O}(N\log(N))$$

Merge Sort ist daher ein worst case-optimales Verfahren. Der Speicherbedarf beträgt *2*N*. Diesen kann man durch zusätzliche Maßnahmen zwar reduzieren, so daß die Methode in situ arbeitet, dann geht jedoch die worst case-Optimalität verloren. Die Merkmale lassen sich wie folgt zusammenfassen:

**Merge Sort**

| | | |
|---|---|---|
| Speicherbedarf | : | *2*N.* |
| Stabilität | : | ist gewährleistet |
| Datenstruktur | : | Array oder Liste |
| Vorsortierung | : | wird beim natürlichen 2-Wege Merge Sort ausgenutzt |
| Laufzeitverhalten | : | $\boldsymbol{C}_{av}(N) = \boldsymbol{C}_{min}(N) = \boldsymbol{C}_{max}(N) = \boldsymbol{O}(N\log(N))$<br>$\boldsymbol{M}_{av}(N) = \boldsymbol{M}_{min}(N) = \boldsymbol{M}_{max}(N) = \boldsymbol{O}(N\log(N))$ |
| Besonderheit | : | worst case-optimale Methode |

## 4.3.4 Radix Sort

Die bisher betrachteten speziellen Sortierverfahren nutzen nur Eigenschaften der Datenstruktur aus, in der die Items gespeichert sind: Quicksort und Heapsort verlangen ein Array als Datenstruktur, Merge Sort ein Array oder eine lineare Liste.

Die in diesem Abschnitt untersuchten Verfahren setzen eine bestimmte Struktur der Schlüsselwerte voraus: Es wird verlangt, daß die Schlüssel n-stellige Zeichenfolgen über einem Alphabet mit b Zeichen sind, zum Beispiel

- *b=2, n=8*: falls es sich um 8-stellige Binärzahlen handelt,
- *b=10, n=4*: falls es sich um 4-stellige Dezimalzahlen handelt,
- *b=26, n=8*: falls es sich um 8-stellige Wörter über dem Alphabet [a, b, ... , z] handelt.

#### Ziffernzugriffsfunktion Z(k,i)

Für die Beschreibung des Verfahrens verwenden wir im eine Funktion `Z(k,i)`, die von einem Schlüssel `k` die i-te Binärziffer liefert. Im Falle eines Schlüssels *k* über der Basis *b* ist `Z(k,i)` die Stelle mit der Wertigkeit $b^i$ .

### 4.3.4.1 Radix-Exchange Sort

Das Radix-Exchange-Verfahren setzt voraus, daß die Schlüssel *Binärzahlen gleicher Länge* sind und die Items in einem Array gespeichert vorliegen.

#### Methode

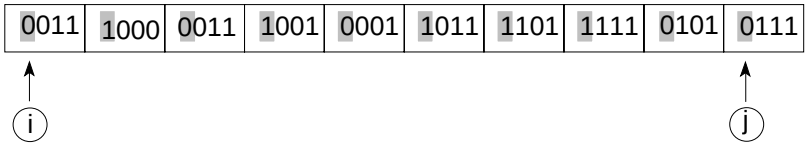
Ähnlich wie bei Quicksort wird die Gesamtliste in zwei Teil-Listen zerlegt, von denen die linke nur "kleine" und die rechte nur "große" Elemente enthält. Als Aufteilungs-Kriterium dient bei der ersten Zerlegung die Ziffernstelle mit der höchsten Wertigkeit.

Die Zerlegung erfolgt genau wie bei Quicksort in situ, so daß eine sortierte Gesamtliste vorliegt, sobald auch die Teil-Listen unabhängig voneinander sortiert sind.

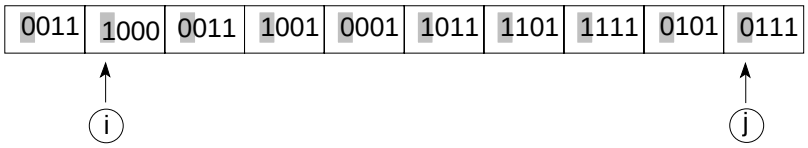
Die Teillisten werden rekursiv weiter zerlegt, wobei nacheinander die Binärstellen mit den nächst niedrigeren Wertigkeiten als Aufteilungskriterium benutzt werden.

#### Beispiel

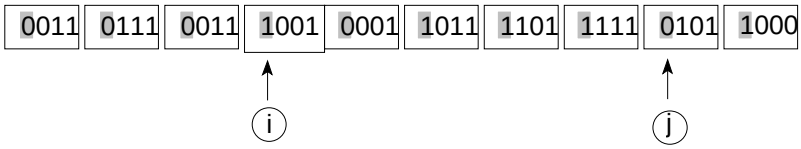
Wir gehen aus von der folgenden Liste, deren Schlüssel 4-stellige Binärzahlen sind. Die beiden Zeiger i und j haben die selbe Bedeutung wie bei Quicksort: Sie laufen von links nach rechts bzw. von rechts nach links, solange bis erstmals ein auszutauschendes Schlüsselpaar angetroffen wird.



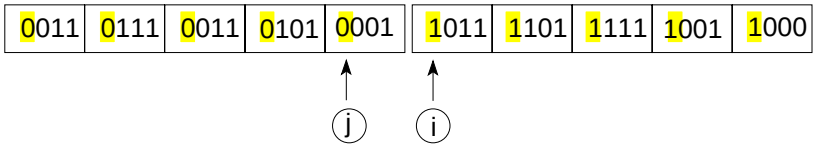

Der erste Durchlauf verwendet die höchstwertige Binärziffer als Aufteilungskriterium. Wir lassen beide Zeiger laufen, bis wir Elemente treffen, die in die jeweils andere Teilliste gebracht werden müssen. Dies ist in der folgenden Situation der Fall:



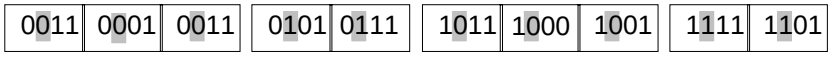
Die beiden Elemente werden vertauscht und die Zeiger laufen weiter, bis wieder zwei Items zu vertauschen sind:



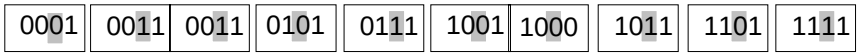

Schließlich erhalten wir die folgenden beiden Teil-Listen: die linke Teilliste enthält auf der höchstwertigen Ziffernposition den Wert 0, die rechte dort den Wert 1.



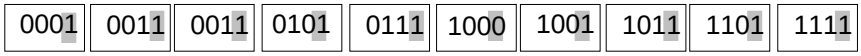

Diese beiden Teil-Listen werden nun analog zerlegt in Abhängigkeit von der zweithöchsten Ziffernposition. Das Ergebnis sind die folgenden vier Teil-Listen:




Die weitere Zerlegung nach dem zweitletzten Bit liefert acht Teil-Listen:




Die letzte Zerlegung stellt schließlich eine sortierte Gesamtliste her. Dabei ist das niedrigstwertige Bit das Kriterium:




### C - Routine

```
// Z(k,i) liefert die i-te Ziffer von rechts in der Zahl k
unsigned Z(unsigned k, int i)
{ return (k >> i) & 0x0001;}


void RadixSort (int *A, int left, int right, int b)
{ int  i, j;                    // Laufvariablen
  int  item;                    // Hilfsfeld zum Vertauschen

  if ((left<right) && (b>=0)) // solange es mindestens 2 Items gibt
  {                           // und die Schlüssel-Länge  b>=0
    i = left; j = right;      // Laufzeiger initialisieren
    while (i != j)
    {
‹     while (Z(A[i],b)==0 && (i<j)) i++;  // von links "1" suchen
›     while (Z(A[j],b)!=0 && (j>i)) j--;  // von rechts "0" suchen
      item = A[i];           // grosses und kleines Item vertauschen
      A[i] = A[j];
      A[j] = item;
    }
fi  if (Z(A[right],b) == 0) j++;
    RadixSort (A,left, j-1,b-1);          // Teil-Listen sortieren
    RadixSort (A,j,  right,b-1);
  }
}
```

Radix Exchange Sort sortiert ein Feld mit Schlüsseln den Länge b, wobei ähnlich wie bei Quicksort zwei Teil.Listen gebildet und diese getrennt sortiert werden. Die Zerlegungsschleifen ‹ und › enden mit i=j. Dabei hat der dort stehende Schlüssel an der Ziffernpsoition b eine 1 und alle links davon stehenden Schlüssel haben an dieser Position eine 0.Eine Ausnahme liegt vor, wenn gar kein Schlüssel auf dieser Position eine 1 hat. Die Abfrage fi dient dazu diesen Fall abzufangen.

### Analyse

Die Anzahl der Vergleiche/Transport-Operationen zum ersten Aufteilen der Gesamtliste in zwei Teil-Listen ist wie bei Quicksort linear von der Listenlänge $N$ abhängig, d.h. von der Ordnung $\boldsymbol{O}(N)$. Auf den nächsten Stufen des Aufteilungsprozesses ist die Gesamtzahl zum Zerlegen der Teillisten einer Stufe ebenfalls $\boldsymbol{O}(N)$.

Die maximale Tiefe der Zerlegungsstufen ist durch die Schlüssel-Länge *n* gegeben. Der Gesamt-Aufwand des Verfahrens ist daher $\boldsymbol{O}(n*N)$. Falls für die Schlüssel-Länge *n* gilt: $n = \log(N)$, dann ist die Effizienz des Radix-Exchange-Verfahrens mit Quicksort vergleichbar. Falls jedoch nur wenige Items mit großer Schlüssel-Länge zu sortieren sind, d.h. $n >> \log(N)$, dann ist die Methode nicht zu empfehlen.

**Radix Exchange Sort**

| | | |
|---|---|---|
| Speicherbedarf | : | in situ-Verfahren. |
| Stabilität | : | ist nicht gewährleistet |
| Datenstruktur | : | Array |
| Vorsortierung | : | hat keine Auswirkungen |
| Laufzeitverhalten | : | $\boldsymbol{C}_{max}(N) = \boldsymbol{O}(n*N)$ (n=Schlüssel-Länge, N=Listenlänge)<br>$\boldsymbol{M}_{max}(N) = \boldsymbol{O}(n*N)$ |
| | : | $\boldsymbol{M}_{max}(N) = \boldsymbol{O}(n*N)$ |
| Besonderheit | : | schlecht für wenige Items mit langem Schlüssel |

### 4.3.4.2 Sortieren durch Fachverteilung

Das Radix Exchange-Verfahren setzte als Schlüssel Binärzahlen voraus. Daher genügte dort bei jedem Zerlegungsschritt eine Aufteilung in zwei Teil-Listen.

Besitzt die Zahlenbasis jedoch mehr als zwei Ziffern, muß man die Methode verallgemeinern und gelangt so zu der als *Sortieren durch Fachverteilung*, *Binsort* oder *Bucketsort* bekannten Methode. Wir setzen zur Vereinfachung dafür voraus, daß es sich um Dezimalzahlen mit gleicher Länge $n$ handelt, d.h. das Alphabet sind die Ziffern $\{0,1,2,\cdots,9\}$ und die Basis ist $b = 10$.

#### Methode

Beginnend mit der *niedrigstwertigen* Schlüssel-Position führen wir nacheinander für jede Ziffernposition eine Verteilungs- und eine Sammelphase durch:

1. In der Verteilungsphase werden die Items auf $b = 10$ Fächer verteilt. Dabei kommt ein Item in das Fach $F_i$ , wenn die Ziffer an der aktuellen Schlüsselposition den Wert $i$ hat. Ein in ein Fach einsortiertes Item wird "oben" auf die bereits dort liegenden gelegt.
2. In der Sammelphase werden die Fachinhalte zu einer neuen Liste zusammengeführt und zwar so, daß zunächst die Items aus Fach $F_0$ , danach die Items aus Fach $F_1$ usw. aneinandergehängt werden.

Mindestens ab der zweiten Verteilungsphase müssen die Items der Reihe nach, beginnend mit dem ersten Item in die Fächer verteilt werden. Nach der letzten Sammelphase (für die höchstwertige Ziffernposition) ist die Liste ganz sortiert.

#### Beispiel

Wir betrachten die folgende Liste zweistelliger Dezimalzahlen:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *33* | *08* | *03* | *25* | *17* | *27* | *11* | *81* | *05* | *23* |

Das Ergebnis der ersten Verteilungsphase, die von der letzten Ziffer gesteuert wird, ist

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | *23* | | | | | | |
| | *81* | | *03* | | *05* | | *27* | | |
| | *11* | | *33* | | *25* | | *17* | *08* | |
| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ | $F_8$ | $F_9$ |

Die anschließende Sammelphase ergibt die folgende Liste:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *11* | *81* | *33* | *03* | *23* | *25* | *05* | *17* | *27* | *08* |

Die zweite Verteilungsphase, diesmal nach der ersten Ziffer von links liefert:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *08* | | *27* | | | | | | | |
| *05* | *17* | *25* | | | | | | | |
| *03* | *11* | *23* | *33* | | | | | *81* | |
| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ | $F_8$ | $F_9$ |

Die anschließende Sammelphase ergibt die sortierte Liste:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 03 | 05 | 08 | 11 | 17 | 23 | 25 | 27 | 33 | 81 |

### Korrektheit der Methode

Falls die Liste bezüglich der letzten t Ziffernpositionen schon geordnet ist, dann bleibt diese Ordnung durch Verteilen nach der nächsten Ziffernposition links davon in jedem der Fächer erhalten und wird auch durch das Sammeln nicht zerstört.

Die erste Sammelphase ergibt eine nach der letzten Ziffer geordnete Folge und die letzte Sammelphase daher eine vollständig geordnete Folge.

### Implementierungsaspekte

Wenn man für jede der *b* Ziffern ein eigenes Fach vorsieht und dieses für die maximal mögliche Anzahl aufzunehmender Items auslegt, dann beträgt der Bedarf an Zusatzspeicher *b*N* Plätze. Dies ist schon für kleine Listen zu viel.

Eine elegantere Technik zählt zunächst ab, wieviele Elemente jedes Fach aufnehmen muß und bringt dann alle Fächer auf einem zusätzlichen Bereich der Länge *N* unter.

Eine andere Alternative realisiert die Fächer als verkettete lineare Listen. In der Verteilungsphase werden neue Elemente jeweils ans Ende einer Liste angehängt. In der Sammelphase werden die Listen vom Angfang her abgebaut und in die Sammel-Liste übernommen.

### C - Routine

Die Schlüssel sind hier **`int`**-Zahlen. Es werden nur die letzten `KeyLen` Dezimalstellen als Key benutzt. Alle Fächer werden auf einem zweiten Feld der Länge *N* untergebracht.

```
// Die Funktion Z10(k,weight) liefert die Dezimalziffer mit dem
// Gewicht weight im Schlüssel
unsigned Z10(unsigned k, int weight)
{  int i;
   i = k / weight;
   i = i - i/10*10;
   return i;
}
void Fachverteilen (int *A, int n)
{
   int  i, j, k;             // Laufvariablen
   int  AA[N+2];             // Hilfsfeld für die Fächer
   int  weight;              // Wertigkeit der aktuellen Ziffernpos.
   int  F[10];               // Zeiger zu den Fächern

   weight = 1;               // Gewicht der 1. Ziffernposition
   for (i=0; i < KEYLEN; i++)// Schleife über die Ziffernpositionen
   {  for (j=0; j<=9; j++) F[j]=0;     // Zählen, wie oft jede Ziffer
      for (j=1; j<=N; j++)             // vorkommt
      {  k = Z10(A[j],weight);
         F[k] = F[k] + 1;
      }
      F[9] = N + 1 - F[9];             // Anfangspos. d.Fächer bestimmen
      for (j=8; j>=0; j--) F[j] = F[j+1] - F[j];

      for (j=1; j<=N; j++)             // Verteilphase
      {  k        = Z10(A[j],weight);
         AA[F[k]] = A[j];
         F[k]     = F[k] + 1;
      }

      for (j=1; j<=N; j++) A[j] = AA[j];// Sammelphase

      weight = weight*10;               // Nächstes Zifferngewicht
   }
}
```

### Analyse

Wenn wir die obigen Bezeichnungen verwenden, nämlich

*b* = Anzahl der verschiedenen Ziffern
*KeyLen* = Länge der Schlüssel
*N* = Länge der Liste

so ist der Zeitaufwand zum Sortieren von der Größenordnung $\boldsymbol{O}\left(KeyLen * \left(b + N\right)\right)$. Der zusätzliche Speicherbedarf ist von der Größenordnung $\boldsymbol{O}\left(b + N\right)$.

In der Regel können wir die Anzahl $b$ der Ziffern gegenüber $N$ vernachlässigen. Falls die Schlüssel verschieden sind, muß $KeyLen > \log_b(N)$ sein.

Solange die Schlüssel aber nicht viel länger als nötig sind, d.h. $KeyLen < c * \log_b(N)$ für eine "kleine" Konstante $c$, erhalten wir für den Zeitaufwand des Verfahrens $\boldsymbol{O}\left(N * \log(N)\right)$ und für den zusätzlichen Speicherplatzbedarf $\boldsymbol{O}\left(N\right)$.

**Sortieren durch Fachverteilen**

| | | |
|---|---|---|
| Speicherbedarf | : | $\boldsymbol{O}\left(N\right)$ zusätzlicher Platz erforderlich |
| Stabilität | : | ist gewährleistet |
| Datenstruktur | : | Array oder Liste |
| Vorsortierung | : | hat keine Auswirkungen |
| Laufzeitverhalten | : | $\boldsymbol{O}\left(N * \log(N)\right)$ |
| Besonderheit | : | Die Komplexität $\boldsymbol{O}\left(N * \log(N)\right)$ gilt für *kurze Schlüssel* und *wenige Ziffern* im Vergleich zur Listenlänge |

# 4.4 Externe Sortierverfahren

## 4.4.1 Voraussetzungen

Alle bisher betrachteten Sortierverfahren gehen von der Annahme aus, daß die zu sortierenden Datenbestände vollständig in den Hauptspeicher passen und damit der Zugriff auf Listenelemente stets gleich lange dauert. Diese Voraussetzung ist für große Datenbestände nicht mehr mit vertretbaren Kosten zu erfüllen. Zwei Auswege bieten sich dafür an:

#### Sortieren in virtuellen Adreßräumen

Man arbeitet auf Rechnern mit einem großen virtuellen Adreßraum. In diesem Fall können alle konventionellen Verfahren ohne Modifikation eingesetzt werden. Die Auswirkungen auf die Laufzeit hängen dann aber stark von der speziellen Speicherverwaltung ab und sind mit Bezug auf Sortierverfahren vermutlich noch nicht untersucht. Es sind auch keine speziell dafür konzipierten Sortierstrategien bekannt.

#### Sortieren auf externen Speichermedien

Eine andere, sehr viel genauer untersuchte Situation liegt vor, wenn die Datenbestände auf sequentiellen, externen Speichermedien sortiert werden. Erste Vorschläge für solche externen Sortier-Algorithmen gehen zurück bis ins Jahr 1945. Dies ist auch darin begründet, daß sequentielle Externspeicher lange Zeit die einzige Möglichkeit zur Speicherung größerer Datenbestände waren. Methodisch sind die externen Sortierverfahren alle verwandt mit den schon betrachteten internen Merge Sort-Algorithmen.

#### Idealisiertes Modell eines externen Speichers

Für die hier betrachteten, externen Sortierverfahren wird ein idealisiertes Modell eines externen Speichers zugrunde gelegt, das sich an den Funktionen orientiert, die bei allen Magnetbandgeräten vorhanden sind. Die charakteristischen Merkmale dieses Modells sind die folgenden:

- Es wird eine unendlich große Datenträgerkapazität angenommen.
- Es besteht nur ein streng sequentieller Zugriff auf die Listenelemente
- Die Zugriffszeiten zum Externspeicher sind sehr viel höher als zum internen Arbeitsspeicher.

Die folgenden Operationen sind in diesem Speichermodell möglich:

- Reset (***T***) Das Band auf die Anfangsmarke stellen und Lesezugriffe erlauben.
- Rewrite(***T***) Das Band auf die Anfangsmarke stellen und Lese/Schreibzugriffe erlauben.
- Read(***T***,Item) Das nächste Listenelement vom Band lesen
- Write(***T***,Item) Das nächste Item auf Band schreiben
- EoF(***T***) Prüfen, ob Ende des Datenbestandes erreicht ist.

Diese Modellvorstellung ist idealisiert, um die für das grundsätzliche Vorgehen wesentlichen Merkmale zu isolieren. Reale Magnetbandgeräte bieten eine Fülle weiterer Funktionen, die im Anwendungsfall natürlich genutzt werden. Dazu gehören z.B.

- geblocktes Lesen und Schreiben,
- kombinierter Lese/Schreib-Modus,
- Streaming-Mode,
- rückwärts lesen.

### Komplexitätsmaß für externe Sortier-Algorithmen

Das Lesen oder Schreiben aller $N$ Datensätze einer zu sortierenden Liste bezeichnen wir als einen Durchgang (Pass), unabhängig davon, ob diese auf einem einzigen Datenträger stehen oder auf mehrere verteilt sind.

Als Maß für den Aufwand eines externen Verfahrens zum Sortieren von $N$ Datensätzen wählen wir die Anzahl der Durchgänge, die dafür notwendig sind, und bezeichnen den minimalen / mittleren / -maximalen Aufwand mit $\boldsymbol{P}_{\min}(N), \boldsymbol{P}_{av}(N), \boldsymbol{P}_{\max}(N)$. Ein Pass beinhaltet in der Regel das Positionieren des externen Datenträgers und das einmalige sequentielle Lesen oder Schreiben der Liste. Dagegen kann der Aufwand für alle internen Operationen vernachlässigt werden.

## 4.4.2 Ausgeglichenes 2 Wege-Mergesort-Verfahren

Diese Methode geht von den folgenden Gegebenheiten aus:

- Es stehen 4 Bandgeräte zur Verfügung, die mit $\boldsymbol{T}_1, \boldsymbol{T}_2, \boldsymbol{T}_3$ und $\boldsymbol{T}_4$ bezeichnet werden.
- Das Band $\boldsymbol{T}_1$ enthält zu Beginn der Verarbeitung die unsortierte Liste.
- Die externe Gesamtliste ist $N_{ext}$ Elemente lang.
- Intern können maximal $N_{\text{int}}$ Listenelemente gleichzeitig gespeichert werden.

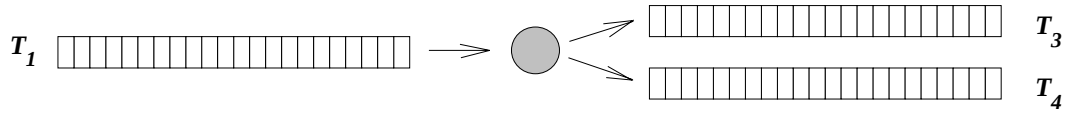
### Runs

Unter einem *Run* verstehen wir eine aufsteigend sortierte Teilfolge innerhalb einer Gesamtliste.

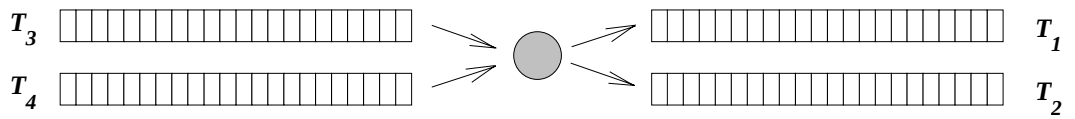
### Methode

Das Verfahren ist direkt vom (internen) Bottom Up Merge-Sort-Verfahren abgeleitet. Es beginnt mit einem einem einmaligen Vorlauf. Danach werden zwei Phasen ständig wiederholt bis die Liste ganz sortiert ist:

*Vorlauf:*
In einem Vorlauf werden zunächst vom Eingabeband $\boldsymbol{T}_1$ Teilfogen in den Arbeitsspeicher eingelesen und dort mit einem internen Verfahren sortiert. Die Länge dieser Teilfolgen ist durch die Anzahl der maximal gleichzeitig intern speicherbaren Listenelemente $N_{\text{int}}$ gegeben. Die sortierten Teilfolgen werden abwechselnd auf die Bänder $\boldsymbol{T}_3$ und $\boldsymbol{T}_4$ ausgegeben, wie die folgende Skizze zeigt:



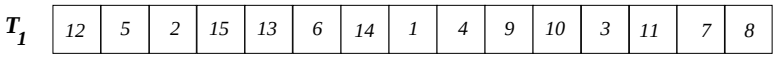

*Phase 1:*
Die beiden Bänder $\boldsymbol{T}_3$ und $\boldsymbol{T}_4$ werden nun zu Eingabebändern. Immer die beiden ersten Runs – einer von jedem der beiden Bänder – werden elementweise zu einem doppelt so großen Run verschmolzen und diese größeren Runs werden dabei abwechselnd auf die Bänder $\boldsymbol{T}_1$ und $\boldsymbol{T}_2$ ausgegeben. Dies deutet die folgende Skizze an:



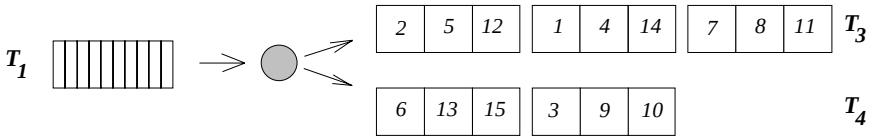

*Phase 2:*
Die Bänder $\boldsymbol{T}_1$ und $\boldsymbol{T}_2$ werden mit den beiden Bänder $\boldsymbol{T}_3$ und $\boldsymbol{T}_4$ vertauscht. Phase 1 wird mit vertauschten Rollen durchgeführt. Dies wird solange wiederholt, bis ein einziger Run entstanden ist.

### Beispiel

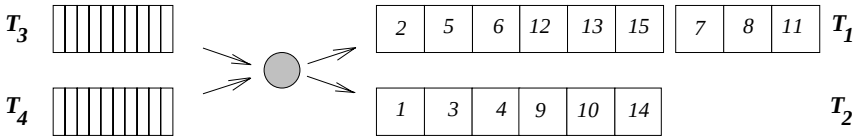
Wir gehen von der folgenden Liste $\boldsymbol{T}_1$ aus und nehmen $N_{\text{int}} = 3$ an:



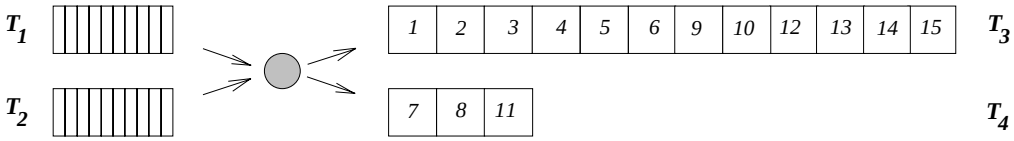

Die Vorbeitungsphase schreibt auf die beiden Ausgabebänder 3 bzw 2 Runs aus:



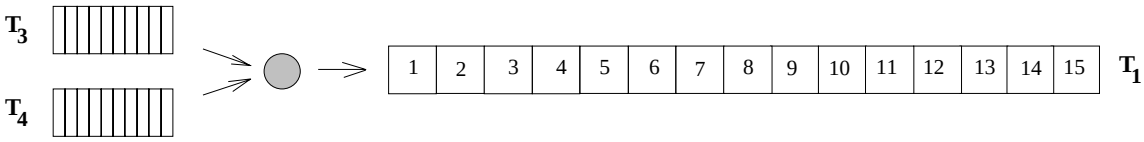

Nach dem ersten Mischvorgang gemäß Phase 1 ergibt sich:




Nach dem Bädertausch (Phase 2) und einem erneuten Mischvorgang in einer Phase 1 ergibt sich die folgende Situation:




Ein letzter Bändertausch und ein erneutes Mischen (Phase 1) stellt schließlich einen einzigen Run her und damit die vollständig sortierte Liste. Diese landet im Beispiel auf dem Band $\boldsymbol{T}_1$, was aber nicht immer der Fall sein muß:




### Analyse

Nach jeder Verschmelzungsphase hat sich bei dieser Methode die Länge der Runs verdoppelt und ihre Anzahl etwa halbiert. Die Vorbereitungsphase erzeugt zunächst $r = N_{ext} / N_{\text{int}}$ Runs mit der Länge $N$. Daher ist nach spätestens $\log_2(r)$ Verschmelzungsphasen ein einziger Run entstanden und damit die Folge ganz sortiert. Der Sortieraufwand läßt sich daher angeben mit

$$\boldsymbol{P}_{\min}(N) = \boldsymbol{P}_{av}(N) = \boldsymbol{P}_{\max}(N) = \log_2\left( N_{ext} / N_{\text{int}} \right)$$

## C - Routine

```
// Verfahrenskonstanten
#define Nintern    50          // maximale interne Listenlänge
#define Nextern  1000          // maximale externe Listenlänge

// Globale Variablen
int     A[Nintern+2];          // intern zu sortierende Liste
FILE *  T[5];                  // handles der Tapes 1, 2, 3, 4
char    tapes[5][10] = {"","TAPE1.DAT","TAPE2.DAT","TAPE3.DAT","TAPE4.DAT" };
                               // Namen der Tapes:
// Prototypen
int     EndOfFile              (FILE **T);
void    PresetTapes            (char *tape, char *tape1, char *tape2);
void    MergeRuns              (FILE *T1,   FILE *T2,    FILE *T);
void    TestItem               (FILE **T,   int *item);
void    nextTape               (int *out);
void    balanced_2way_MergeSort (int *i);
void    SelectionSort          (int *A, int n);

//---------------------------------------------------------------------------
//   Internes Sortieren durch Auswaehlen fuer  Nintern  Elemente
//---------------------------------------------------------------------------
void SelectionSort (int *A, int n)
{  int  i, j;                  // Laufvariablen
   int   min;                  // Minimum im Restfeld
   int  item;                  // Hilfsfeld zum Vertauschen

   for (i=1;i<n;i++)           // durchlaeuft die zu sortierende Liste
   { min = i;                  // Index des bisher kleinesten Elements
     for (j=i+1;j<=n;j++)      // Suche, ob noch ein kleineres Element kommt
        if (A[j]<A[min]) min = j;
     item   = A[min];          // jetzt werden das erste und das kleinste
     A[min] = A[i];            // Item der Restliste vertauscht
     A[i]   = item;
   }
}

//---------------------------------------------------------------------------
//   SetupTape: Band vorbereiten fuer das Verfahren
//---------------------------------------------------------------------------
void SetupTape(char *tape, int n)
{   int i;
    FILE *T;
    randomize();
    T = fopen (tape ,"w");
    for (i=1;i<=n;i++)  fprintf (T, "%d\n", random(1000));
    fclose (T);
}

//---------------------------------------------------------------------------
//  PresetTapes: Die ersten beiden Bänder mit Runs belegen
//---------------------------------------------------------------------------
void PresetTapes (char *tape, char *tape1, char *tape2)
{   FILE *T, *T1, *T2;
    int i, imax;

    T  = fopen (tape ,"r");
    T1 = fopen (tape1,"w");
    T2 = fopen (tape2,"w");
    while (EndOfFile(&T)!=0)
    { // Ersten Datenblock holen und als Run auf Tape 1 schreiben
      i=0;                          // Datenblock einlesen
       while ( (EndOfFile(&T)!=0) && (i<Nintern))
         fscanf  (T,"%d ", &(A[++i]));
      imax = i;                     // Anzahl gelesener Elemente merken
      SelectionSort (A,imax);       // Datenblock sortieren zu einem Run
      for (i=1; i<=imax; i++)       // Run auf Tape 1 ausgeben
         fprintf (T1, "%d\n", A[i]);
```

```
      // Zweiten Datenblock holen und als Run auf Tape 2 schreiben
      i=0;                              // Datenblock einlesen
      while ( (EndOfFile(&T)!=0) && (i<Nintern))
        fscanf  (T,"%d ", &(A[++i]));
      imax = i;                         // Anzahl gelesener Elemente merken
      SelectionSort (A,imax);           // Datenblock sortieren zu einem Run
      for (i=1; i<=imax; i++)           // Run auf Tape 1 ausgeben
        fprintf (T2, "%d\n", A[i]);
    }
    fclose (T); fclose (T1); fclose (T2);
}

//---------------------------------------------------------------------------
//  EndOfFile:  Feststellen, ob eine Datei noch Items enthält
//              auch true, wenn das nächste Lesen nichts mehr findet.
//---------------------------------------------------------------------------
int EndOfFile (FILE **T)
{   int i, item=0;
    fpos_t filepos;

    fgetpos (*T, &filepos);             // aktuelle Dateiposition merken
    i = fscanf (*T, "%d ", &item);      // Element probeweise lesen
    if ( i!=-1 )                        // wenn ein Item gefunden wurde
    {  fsetpos (*T, &filepos);          // alte Position wiederherstellen
       return(i);                       // und '1' zurückliefern
    }
    else                                // wenn kein Item mehr da war
      return (0);                       // '0' zurückliefern
}

//---------------------------------------------------------------------------
//  TestItem: Item lesen, ohne Dateiposition zu verändern      ****
//---------------------------------------------------------------------------
void TestItem (FILE **T, int *item)
{   fpos_t filepos;
    fgetpos (*T, &filepos);             // aktuelle Dateiposition merken
    fscanf  (*T, "%d ", item);          // Element probeweise lesen
    fsetpos (*T, &filepos);             // alte Position wiederherstellen
}

//---------------------------------------------------------------------------
//  MergeRuns:  Erzeugt aus zwei von externen Datenträgern T1, T2
//              gelesenen Runs einen neuen Run auf einem  dritten
//              externen Datenträger T
//---------------------------------------------------------------------------
void  MergeRuns (FILE *T1, FILE *T2, FILE *T)
{   int item, item1, item2;

    // Solange T1 und T2 noch Daten enthalten
    while ( (EndOfFile(&T1)!=0) && (EndOfFile(&T2)!=0))
    {
      TestItem (&T1, &item1);               // Prüfe die ersten Elemente
      TestItem (&T2, &item2);               // der beiden Runs:
      if (item1 <= item2)                   // Welches Element ist kleiner ?
      {  fscanf  (T1, "%d " , &item1);      // Item auf T1 verarbeiten
         fprintf (T , "%d\n",  item1);

         // Falls der Run zu Ende ist oder das Ende von T1 erreicht ist,
         // den aktuellen Run von T2 noch komplett übertragen
         if ((EndOfFile(&T1)!=0))   TestItem (&T1,&item);
         if ((EndOfFile(&T1)==0) || item1 > item )
         {  item = item2;                   // vorhergehendes Item merken
            do
            { fscanf  (T2,"%d "  , &item2); // nächstes Item holen
              fprintf (T ,"%d \n", item2);  // und ausgeben
              item = item2;                 // Item merken
              TestItem (&T2, &item2);       // ist nächstes Item noch im Run?
            } while ( (item <= item2) && (EndOfFile(&T2)!=0) );
            return;
         }
      }
```

```
      else
      {  fscanf  (T2, "%d ",  &item2);      // Item auf T2 verarbeiten
         fprintf (T, "%d\n",  item2);

         // Falls der Run zu Ende ist oder das Ende von T2 erreicht ist,
         // den aktuellen Run von T2 noch zu Ende übertragen
         if ((EndOfFile(&T2)!=0))  TestItem (&T2,&item);
         if ((EndOfFile(&T2)==0) || item2 > item )
         { item = item1;                    // vorangehendes Item merken
           do
           { fscanf  (T1, "%d ", &item1);   // nächstes Item holen
             fprintf (T, "%d \n", item1);   // und ausgeben
             item = item1;                  // Item merken
             TestItem (&T1, &item1);        // ist nächstes Item noch im Run?
           } while ( (item <= item1) && (EndOfFile(&T1)!=0) );
           return;
         }
      }
    }
}

//---------------------------------------------------------------------------
//  nextTape:  Ausgabeband umschalten
//---------------------------------------------------------------------------
void nextTape(int *out)
{   (*out)++;
    if (*out==3)  *out=1;
    if (*out==5)  *out=3;
}

//---------------------------------------------------------------------------
//  Balanced 2-Way Merge Sort:
//  Parameter:     i   Index des Bandes mit dem Ergebnis
//---------------------------------------------------------------------------
void balanced_2way_MergeSort (int *i)
{   int   out;                              // aktuelles Ausgabeband
    int   in1, in2, out1, out2;             // Indizes der Bänder
    int   j;                                // Hilfsgröße zum Vertauschen
    int item;                               // Element der Datei

    // Schreibe Runs der Länge N auf die Bänder T[3] und T[4]
    PresetTapes (tapes[1], tapes[3], tapes[4]);

    in1  = 3; in2  = 4;   // Nummern der Ein- und Ausgabebänder initialisieren
    out1 = 1; out2 = 2;

    // Arbeite solange bis T[in2] leer ist
    // T[in1] enthält dann die sortierte Datei
    T[in2] = fopen (tapes[in2],"r");
    while ((EndOfFile(&T[in2])!=0))
    { out = out2;      // auf out2 wurde gerade geschrieben
      fclose(T[in1]);  T[in1 ] = fopen(tapes[in1 ],"r");
      fclose(T[out1]); T[out1] = fopen(tapes[out1],"w");
      fclose(T[out2]); T[out2] = fopen(tapes[out2],"w");

      // Runs auf T[in1], T[in2] verschmelzen nach T[out]
      // Dabei enthält T[in2] höchstens so viele Runs wie T[in1]
      while ( (EndOfFile(&T[in2])!=0) )
      {  nextTape (&out);                   // Ausgabeband umschalten
         MergeRuns (T[in1], T[in2],T[out]); // nächste Runs ver-schmelzen
      }
      nextTape(&out);                       // Ausgabeband umschalten
      while ( (EndOfFile(&T[in1])!=0) )     // Rest von T[in1] kopie-ren
      {  fscanf  (T[in1], "%d ", &item);
         fprintf (T[out], "%d\n", item);
      }
      j=in1; in1=out1; out1=j;              // Bänder tauschen
      j=in2; in2=out2; out2=j;
      fclose(T[in2]); T[in2] = fopen (tapes[in2],"r"); // T[in2] neu öffnen
   }
   *i = in1;                                 // T[i] = sortierte Datei
}
```

Der oben betrachtete Ansatz läßt sich in verschiedenster Richtung ausbauen. Zwei mögliche, daraus abgeleitete Verfahren sind die folgenden:

### Ausgeglichenes Mehrwege-MergeSort-Verfahren

Im Gegensatz zum 2 Wege-MergeSort-Verfahren werden $k$ *( k=2, 4, 6, ...)* Bandgeräte benutzt. Für die Eingabe von Runs wird die eine Hälfte der Bänder, für die Ausgabe verschmolzener Runs die andere eingesetzt.

### Mehrphasen-Merge Sort-Verfahren

Es werden $k+1$ Bandgeräte vorausgesetzt. Davon werden zu jedem Zeitpunkt k Geräte für die Eingabe verwendet und eines für die Ausgabe.

Die entstehenden Runs werden auf das einzige Ausgabeband geschrieben. Sobald eines der Eingabebänder leer ist, wird das Ausgabeband zurückgespult und von da an als Eingabeband mitverwendet. Das leergelesene Eingabeband wird neues Ausgabeband.

## 4.5 Die Rolle der Vorsortierung

In vielen Fällen sind die zu sortierenden Datenbestände nicht völlig ungeordnet, z.B.

- wenn sie sie vorher schon einmal nach einem ähnlichen Kriterium geordnet wurden oder
- wenn Teile des Datenbestandes schon sortiert sind.

Die bisher betrachteten Sortierverfahren nehmen auf solche Situationen keine Rücksicht. In manchen Fällen, etwa bei Quicksort sind sie sogar besonders ineffizient, wenn die Datenbestände schon geordnet sind. Es ist daher notwendig bei der Auswahl eines geeigneten Sortierverfahrens den Aspekt der Vorsortierung von Datenbeständen zu beachten.

In diesem Abschnitt werden wir daher genauer präzisieren, wie Vorsortierung gemessen werden kann und Verfahren kennenlernen, die eine Vorsortierung optimal nutzen.

### Maße für die Vorsortierung

Die Tatsache, daß ein Datenbestand sortiert ist, wird durch den mathematischen Begriff der linearen Ordnung in allgemein akzeptierter Weise exakt beschrieben. Daß er nur teilweise sortiert ist, läßt sich nicht so leicht generell verbindlich präzisieren.

Um eine Vorstellung von den Möglichkeiten zur begrifflichen Festlegung zu bekommen, betrachten wir einige Beispielfolgen, die in unterschiedlicher Weise als vorsortiert gelten können:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Folge ***A :*** | *2* | *1* | *4* | *3* | *6* | *5* | *8* | *7* |
| Folge ***B :*** | *5* | *6* | *7* | *8* | *1* | *2* | *3* | *4* |
| Folge ***C :*** | *7* | *3* | *5* | *1* | *8* | *6* | *4* | *2* |

Die Folge ***A*** könnte man als global gut sortiert bezeichnen, denn kleine Items stehen eher am Anfang der Liste. Lokal stehen jedoch Paare benachbarter Items in der falschen Reihenfolge.

In der Folge ***B*** besitzen unmittelbare Nachbarn in den meisten Fällen schon die richtige Reihenfolge, dagegen steht die kleinere Hälfte der Items am Ende der Folge. Man wird eine solche Folge als lokal gut sortiert bezeichnen, global ist sie eher schlecht sortiert.

In der Folge ***C*** ist eine keine, auf den ersten Blick erkennbare Vorsortierung vorhanden.

Diese Beispiele zeigen, daß der Grad der Vorsortierung kaum mit einem einzigen Begriff gefaßt werden kann. Wir diskutieren daher im folgenden verschiedene, einander ergänzende Ansätze dafür.

### Das Maß *inv*

Wir betrachten zu einer Folge **F** die Paare von (nicht unbedingt benachbarten) Items, die in der falschen Reihenfolge stehen. Für die Folgen **A**, **B** und **C** ergeben sich so folgende Mengen falsch angeordneter Item-Paare:

***A***: *(2,1) (4,3) (6,5) (8,7)*

***B***: *(5,1) (5,2) (5,3) (5,4)*
*(6,1) (6,2) (6,3) (6,4)*
*(7,1) (7,2) (7,3) (7,4)*
*(8,1) (8,2) (8,3) (8,4)*

***C*** : *(7,3) (7,5) (7,1) (7,6) (7,4) (7,2)*
*(3,1) (3,2)*
*(5,1) (5,4) (5,2)*
*(6,4) (6,2)*
*(8,6) (8,4) (8,2)*

Für die global gut sortierte Folge ***A*** ergibt sich also eine geringe Zahl von Inversionen, während die anderen beiden Folgen wesentlich mehr Fehlstellungen aufweisen. Wir definieren daher als ein erstes Maß für die globale Unordnung einer Folge $\boldsymbol{F} = \langle i_1, i_2, \cdots, i_N \rangle$ von Items in mit den Schlüsseln $k_n$ ihre Inversionszahl

$$\boldsymbol{inv}(\boldsymbol{F}) = \#\left\{\left(k_i, k_j\right) \mid 1 \le i < j \le N \text{ und } k_i > k_j\right\}$$

Das Maß $\boldsymbol{inv}(\boldsymbol{F})$ ist also für eine global gut vorsortierte Folge klein und bei vollständiger Sortierung sogar *0*. Den größten Wert nimmt $\boldsymbol{inv}(\boldsymbol{F})$ für eine absteigend sortierte Folge an. In diesem Fall erhalten wir:

$$\boldsymbol{inv}(\boldsymbol{F}) = \left(N-1\right) + \left(N-2\right) + \cdots + 1 = \frac{N\left(N-1\right)}{2}$$

Folgen mit lokal guter Sortierung, wie z.B. die Folge ***B*** werden von der Inversionszahl eventuell doch als schlecht sortiert eingestuft. Wir wollen daher mit einem alternativen Maß versuchen, diesen Typ der Vorsortierung besser zu erfassen.

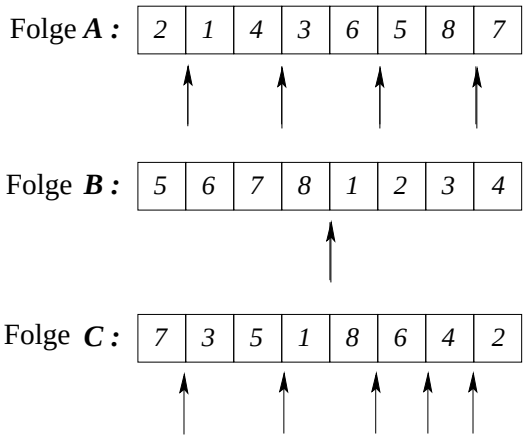
### Das Maß *runs*

Als Run-Zahl einer Folge bezeichnen wir die Anzahl von möglichst langen, aufsteigend sortierten Teilfolgen, die wir aus benachbarten Elementen bilden können. Diese Anzahl ist identisch mit der Anzahl direkter Nachbarn, die in falscher Reihenfolge stehen, weil an solchen Stellen aufsteigend sortierte Teilfolgen abbrechen müssen.

Wir definieren daher als ein weiteres Maß für die Vorsortierung

$$\boldsymbol{runs}(\boldsymbol{F}) = 1 + \#\left\{i \mid 1 \le i < N \text{ und } k_{i+1} > k_i\right\}$$

Für die oben angegebenen Folgen ***A***, ***B*** und ***C*** berechnen wir nun die Run-Zahlen. Die Positionen, an denen Inversionen vorkommen, werden zunächst durch Pfeile markiert:




Damit erhalten wir für die Folgen ***A***, ***B*** und ***C*** die folgenden Run-Zahlen:

$$\begin{aligned} \textbf{\textit{runs(A)}} &= \textbf{\textit{4 + 1 = 5}} \\ \textbf{\textit{runs(B)}} &= \textbf{\textit{1 + 1 = 2}} \\ \textbf{\textit{runs(C)}} &= \textbf{\textit{5 + 1 = 6}} \end{aligned}$$

Die lokal gut sortierte Liste ***B*** besitzt also eine kleine Run-Zahl, währen die beiden anderen Folgen mit diesem Maß als schlecht vorsortiert eingestuft werden.

Für eine aufsteigend sortierte Folge ***F*** hat also ***runs(F)*** den Wert *1*. Der schlechteste Grad der Vorsortierung liegt bei einer absteigend sortierten Folge vor, weil dort alle benachbarten Items in falscher Reihenfolge stehen: ihre Run-Zahl beträgt *N*.

### Das Maß *rem*

Ein ideales Maß für die Vorsortierung sollte, weder rein lokal noch rein global orientiert sein. Wir betrachten daher ein weiteres Maß, das dieser Forderung eher entspricht als die Maße $\textbf{\textit{inv}}(\textbf{\textit{F}})$ und $\textbf{\textit{runs}}(\textbf{\textit{F}})$: es orientiert sich an der längsten, aufsteigend sortierten Teilfolge, die in einer Folge ***F*** enthalten ist:

$$\textbf{\textit{rem}}(\textbf{\textit{F}}) = \begin{cases} \text{Anzahl derjenigen Elemente, diewenigstens aus} \\ \textbf{\textit{F}} \text{ entfernt werden müssen, um eine aufsteigend} \\ \text{sortierte Folge zu erhalten.} \end{cases}$$

Für die Folgen ***A***, ***B*** und ***C*** ergeben sich :

$$\begin{aligned} \textbf{\textit{rem(A)}} &= \textit{4} \\ \textbf{\textit{rem(B)}} &= \textit{4} \\ \textbf{\textit{rem(C)}} &= \textit{5} \end{aligned}$$

Die Berechnung von $\textbf{\textit{rem}}(\textbf{\textit{F}})$ ist im allgemeinen ein nichttriviales algorithmisches Problem und soll hier nicht weiter verfolgt werden.

# 5. Elementare Suchverfahren

Das Suchen von Items in einer vorgegebenen Menge spielt in zahlreichen DV-Verfahren eine wichtige Rolle, z.B.

- das Suchen einer Datei in einem Filesystem,
- das Suchen von Records in einer Datenbank,
- das Suchen einer Zeichenfolge in einem Text,
- das Suchen von synonymen Begriffen
- das Suchen von Postleitzahlen, Telefonnummern, Kontonummern, eMail-Adressen, ....
- das Suchen von Bildern mit bestimmten Inhalten
- das Suchen von URL zu einem Thema
- die Bestimmung des Medianwertes in einer Zahlenmenge

In diesem Kapitel betrachten wir nur *elementare* Suchverfahren, die durch die folgenden Merkmale charakterisiert sind:

- Die Items der Basismenge sind durch einen Schlüssel eindeutig charakterisiert. Es gibt bei einem Suchprozeß also keine Mehrfachtreffer.
- Die Schlüsselmenge besitzt eine lineare Ordnung.
- Als Operationen sind nur Schlüsselvergleiche erlaubt.
- Das Ergebnis der Suche ist entweder das gefundene Item oder eine Fehlanzeige.

In späteren Kapiteln werden wir noch weitere Verfahrenstypen kennenlernen:

- Hash-Verfahren: Diese berechnen die Position eines Items aus dem Wert des Schlüssels.
- Suchbäume: Dabei werden die Eigenschaften der Datenstruktur beim Suchen genutzt.

## 5.1 Bestimmung des Medianwertes

Eine einfache Problemstellung ist das Auffinden des kleinsten Schlüssels in einer ungeordneten Menge mit $N$ Items. Sie läßt sich mit $N-1$ Vergleichen, aber nicht mit weniger beantworten. Eine modifizierte Fragestellung zielt auf den zweitkleinsten, allgemein auf den i-kleinsten Schlüssel ab. Im Fall $i = N/2$ ist dies der *Medianwert*. Wir betrachten zunächst unterschiedliche Lösungsansätze für die Suche des i-kleinsten Schlüssels.

### Mehrfache Auswahl des kleinsten Items

Bei diesem Vorgehen wird zunächst das kleinste Item bestimmt, was $N-1$ Vergleiche erfordert. Danach bestimmt man unter den restlichen $N-1$ Items das kleinste usf. Der Gesamtaufwand zur Bestimmung des Medianwertes beträgt damit

$$(N-1)+(N-2)+\cdots+N-N/2 = \boldsymbol{O}\left(N^2\right) \quad \text{Vergleiche.}$$

### vorgeschaltetes Sortieren

Der quadratische Aufwand im ersten Ansatz kann reduziert werden, wenn wir ein geeignetes Sortierverfahren vorschalten: Dann setzt sich der Aufwand für die Bestimmung des Medianwertes zusammen aus dem Aufwand zum Sortieren und dem Zugriff auf das i-te Item der sortierten Liste. Wählen wir dabei z.B. Heap Sort, dann können wir den Medianwert in maximal $\boldsymbol{O}\left(N*\log(N)\right)$ Schritten bestimmen.

### Heap-Methode

Eine vollständige Sortierung der Basismenge ist zur Bestimmung des Medians gar nicht notwendig. Wir können aus dem Heap-Sort-Verfahren eine Methode ableiten, die den Medianwert effektiver ermittelt, aber immer noch mit der Größenordnung $\boldsymbol{O}(N*\log(N))$ für den Aufwand:

- Zunächst ordnen wir die Basismenge in einen Min-Heap um. In einem Min-Heap sind die Wurzeln aller Teilbäume kleiner als die Items in den Unterbäumen. Er läßt sich daher analog zum Heap-Sort-Verfahren in $\boldsymbol{O}(N)$ Schritten herstellen.
- Wir entfernen die Wurzel des Min-Heap, die das kleinste Item enthält und stellen das letzte Item des Heap an ihre Stelle.
- Wir lassen die neue Wurzel analog zu Heap-Sort versinken in maximal $\boldsymbol{O}(\log(N))$ Schritten.Die letzten beiden Schritte müssen wir zur Ermittlung des Medianwertes $N/2$ -mal wiederholen. Der Gesamtaufwand zur Bestimmung des Medianwertes ist daher

$$\boldsymbol{O}\left(N + N/2 * \log(N)\right) = \boldsymbol{O}(N*\log(N))$$

### Divide et Impera-Strategie

Diese Methode basiert auf den selben Ideen wie das Quicksort-Verfahren. Sie ermöglicht unter bestimmten Voraussetzungen eine Bestimmung des Medianwertes sogar in $\boldsymbol{O}(N)$ Schritten: Zunächst benötigen wir eine Zerlegungsmethode, die eine Liste wie bei Quicksort mit Hilfe eines Pivot-Elements in zwei Teil-Listen aufspaltet:

```
int  Partition  (int *A, int left, int right)
```

Die Funktion `Partition` zerlegt die Liste `A[left..right]` in die Teilliste `A[left..m-1]` mit Items, die ≤ pivot sind und in `A[m..right]` mit Items, die ≥ pivot sind. Als Funktionswert wird der Index m der Teilungsposition zurückgegeben.

Mit dem folgenden Verfahren können wir nun das k-kleinste Element einer Liste bestimmen: Es setzt in dieser Form voraus, daß alle Items verschieden sind.

```
void Select (int *A, int left, int right, int i, int *found)
{
  int  m;                                  // Teiler-PositionItem

  if (right>left)                          // Falls Listenlänge >=2
  {
    m = Partition (A,left,right);          // Liste aufteilen
    if (i<=m-left)                         // falls noch nicht gefunden
      Select (A,left,m-1,i,found);         // suche links weiter
    else
      Select (A,m,right,i-(m-left),found);// suche rechts weiter
  }
    else
    *found=A[left];                        // gefundenes Item abliefern
  return;
}
```

Wir stellen fest, daß bei ungünstiger Wahl des Pivot-Elements der Gesamtaufwand zur Bestimmung des Medianwertes analog zu Quicksort die Ordnung $\boldsymbol{O}(N^2)$ hat, z.B. dann wenn die Zerlegung stets eine Liste mit nur einem Element abspaltet.

Wenn es gelingt, die Teilung stets so vorzunehmen, daß höchstens ein konstanter Prozentsatz $q$ der Listengröße $N$ in jeder der beiden Teillisten vorkommt $0 < q < 1$, dann ist der Aufwand zur Bestimmung des Medians nur noch linear:

Dazu sei $T(N)$ der Aufwand zur Bestimmung des *i*-kleinsten Item aus $N$ Items. Dann setzt sich $T(N)$ zusammen aus dem linearen Aufwand für das Aufteilen und dem Aufwand zur Bestimmung des gesuchten Items in einer Teilliste:

$$T(N) = c*N + T(q*N) \le c*N*\sum_{i=0}^{\infty} q^i = c*N*\frac{1}{1-q} = \boldsymbol{O}(N)$$

Eine geeignete Methode, die diese Teilungsbedingung garantiert, ist die *Median-of-Median-Strategie:*

- Man teilt die Liste mit *N* Elementen in Gruppen zu je 5 Items.
- Man bestimmt in jeder 5er-Gruppe den Medianwert.
- Man verwendet die Funktion Select rekursiv zur Bestimmung des Medianwertes unter den $N/5$ Medianwerten. Das Ergebnis ist das Pivot-Element für die Aufteilung.

Man kann zeigen daß bei dieser Strategie jede der beiden Teillisten zwischen $1/4$ und $3/4$ der Items enthalten muß, wenn $N \ge 75$ ist. Die prinzipielle Situation stellt die folgende Skizze dar:

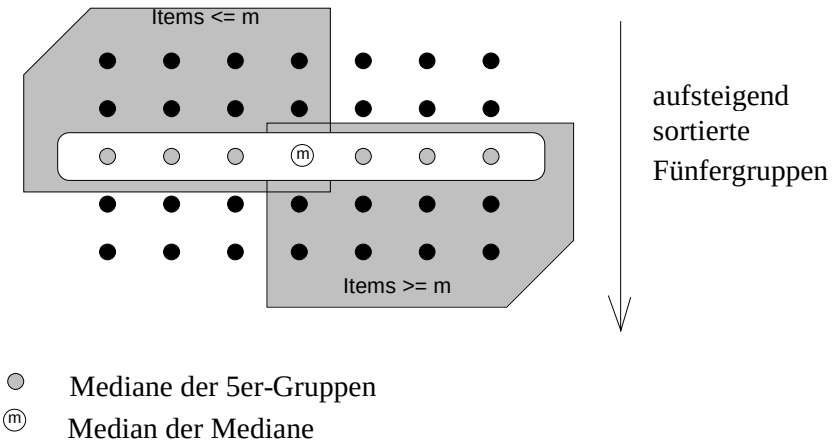



**Abbildung 5.1:** Zur Median-of-Median-Strategie

Abschließend betrachten wir noch eine iterative Variante des Algorithmus. Der Parameter i gibt wieder an, daß das i-kleinste Element zu suchen ist. Mit $i = N/2$ wird also der Medianwert gesucht. Der Parameter found liefert dann das gesuchte Element.

```
void SelectNR (int *A, int left, int right, int i, int &found)
{
  int  j, k;                        // Laufvariablen
  int  item;                        // Hilfsfeld zum Vertauschen
  int  pivot;                       // Pivot-Element

  while (left<right)                // Falls Listenlänge >=2
  {
    pivot = A[i];                   // i-tes Item wird Pivot
    k = left; j = right;            // Laufzeiger initialisieren
    do
    {
      while ((A[k] < pivot)) k++;  // von links  "grosses" El. suchen
      while ((A[j] > pivot)) j--;  // von rechts "kleines" El. suchen
      if (k <= j)
      {
        item = A[k];                // beide vertauschen
        A[k] = A[j];
        A[j] = item;
        k++; j--;
      }
    }  while (k<=j);

    // Falls das linke Teilfeld zu kurz ist, suche rechts weiter
    if (j<i)  left  = k;            // suche im rechten Teil weiter
    if (i<k)  right = j;            // suche im rechten Teil weiter
  }
  found = A[i];                     // Item gefunden, Wert abliefern
  return;
}
```

Das Verfahren verwendet als Kriterium zur Aufspaltung in Teillisten das Item A[i]. Es endet, wenn die Aufteilung genau an der Position i stattgefunden hat. Dann ist aber A[i] genau das k-kleinste Item.

# 5.2 Elementare Suchverfahren

### Sequentielles Suchen

Wir setzen in diesem Abschnitt voraus, daß die zu durchsuchende Liste in einem Array gespeichert ist, das die folgende Gestalt hat, z.B.

```
int A[N+1];
```

Das Feldelement `A[0]` wird wie üblich als Stopper-Element verwendet und darf daher kein Listen-Element enthalten.

Das *lineare* oder *sequentielle Suchen* ist das einfachste Suchverfahren. Es setzt insbesondere nicht voraus, daß die zu durchsuchende Liste sortiert ist. Die folgende Routine stellt eine Implementierung des Verfahrens dar:

```
int SeqSearch (int *A, int n, int item)
{ int i;                    // Laufvariable

  A[0] = item;              // Stopper-Element besetzen
  i    = n + 1;
  while (item != A[--i]);   // Suchschleife
  return (i);               // Index des Item oder 0 zurückliefern
}
```

Die Anzahl von notwendigen Vergleichen ist maximal $N+1$, dann nämlich, wenn das Item nicht gefunden und bis zur Stopper-Position gesucht wird. Im Mittel muß man die Liste zur Hälfte durchsuchen. Man benötigt für einen Treffer im Mittel:

$$\frac{1}{N}\sum_{i=1}^{N} i = \frac{N+1}{2}$$

Vergleiche. Das sequentielle Suchen kann im Gegensatz zu den weiter unten betrachteten Verfahren auch mit verketteten, linearen Listen durchgeführt werden.

### Binäres Suchen

Für das Binäre Suchen und die folgenden Verfahren setzen wir nun zusätzlich voraus, daß die Liste als ein *sortiertes Feld* vorgegeben ist.

Es wird nach einer Divide et Impera-Strategie gesucht: Wir zerlegen die Liste beim mittleren Item `A[m]` in zwei Teile. Da die Liste sortiert ist, können wir aufgrund des Vergleichs mit `A[m]` entweder einen Treffer melden, im linken oder im rechten Teil weitersuchen.

```
int  BinSearch  (int *A, int left, int right, int item)
{ int i,j;

  while (right >= left)           // solange noch mind. 1 Item exist.
  {
    i = (left + right) / 2;       // Feld teilen
    if (item < A[i])  right = i-1; // links weitersuchen
    else              left  = i+1; // rechts weitersuchen
    if (item == A[i]) return(i);   // bei Erfoilg Index zurückliefern
    else              j=0;
  }
  return (j);                     // 0 als Fehlanzeige zurückliefern
}
```

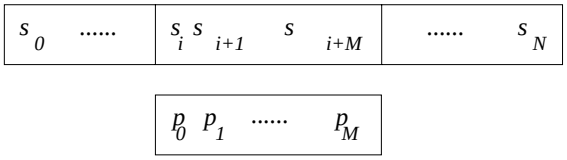
# 5.3 Suchen in Texten

Gegenüber allen bisher betrachteten Suchverfahren besitzt das Suchen von Zeichenfolgen in einem Text einige Besonderheiten:

- Der Datenmenge, in der gesucht wird, ist eine nicht weiter strukturierte Folge von Zeichen aus einem Alphabet, also ein *String* ***s***.
- Es werden nicht einzelne Items der Basismenge gesucht, also Alphabetzeichen, sondern ein bestimmtes *Muster* ***p*** (Pattern), das selbst wieder ein String ist.

### Problemstellung

Zu einer Zeichenfolge $\boldsymbol{s} = s_0 s_2 \cdots s_N$ und einem Muster $\boldsymbol{p} = p_0 p_2 \cdots p_M$ wird eine Position $i$ in $\boldsymbol{s}$ gesucht mit $s_i\, s_{i+1} \cdots s_{i+M} = p_0\, p_1 \cdots p_M$. Dabei muß $0 \le i \le N - M + 1$ sein und in der Regel ist auch $M << N$.

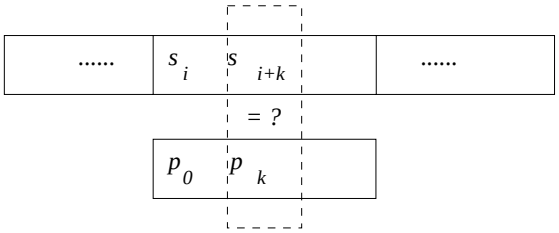



## 5.3.1 Direkte Mustersuche

### Methode

Der einfachste Ansatz zum Suchen eines Musters ***p*** in einem String ***s*** legt nacheinander das Muster ***p*** an aufeinanderfolgenden Positionen i im String ***s*** an und vergleicht die Zeichen des Musters mit den Zeichen $s_i\, s_{i+1} \cdots s_{i+M}$ des String ***s***:

Sobald erstmals ein Zeichenpaar $s_{i+k} \neq p_k$ gefunden ist, wird das Muster an der nächsten Position $i+1$ angelegt und erneut auf Übereinstimmung mit dem String ab dieser Stelle geprüft.

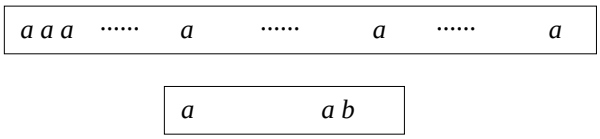



Das folgende C-Programm implementiert diese Methode:

```
int  TxtSearch  (char *s, char *p)
{
  int n, m;                 // String- Musterlänge
  int i, j;                 // Suchzeiger
  unsigned  int found;      // Flag: 1 == gefunden, 0 == Fehlanzeige

  n = strlen(s);            // Länge des String
  m = strlen(p);            // Länge des Pattern
  for (i=0; i<n-m+1; i++)   // über alle Positionen im String
  {
    found = 1;              // noch alles O.K.
    for (j=0; j<m; j++)     // über alle Positionen im Pattern
      if (s[i+j] != p[j]) found = 0;
    if (found == 1) return (i);
  }
  return(-1);
}
```

**Analyse**

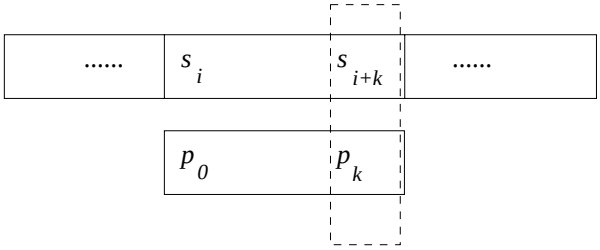
Das Verfahren benötigt im ungünstigsten Fall $O(N*M)$ Vergleiche, nämlich dann, wenn das Muster ab jeder Position $i$ vollständig verglichen werden muß. Dies ist z.B. in der folgenden Situation der Fall:



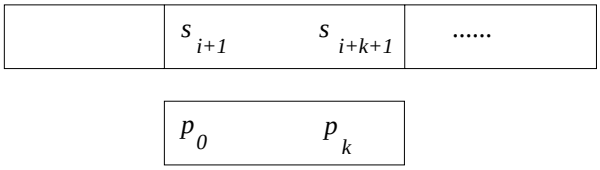

Falls das Alphabet viele Zeichen besitzt, ist zu erwarten, daß in den meisten Fällen eine Nicht-Übereinstimmung sehr viel früher entdeckt wird.

## 5.3.2 Das Verfahren von Knuth, Morris und Pratt

Die oben beschriebene direkte Methode nutzt im Fall einer Nicht-Übereinstimmung die bereits gewonnenen Informationen nicht optimal. Falls in der folgenden Situation eine Nicht-Übereinstimmung festgestellt wird:



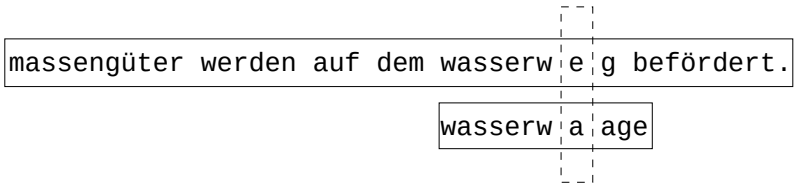

dann werden bei der direkten Mustersuche die Zeichen $s_{i+1}\, s_{i+2} \cdots s_{i+k+1}$ nach dem Verschieben des Musters auf Position $i+1$ noch einmal verglichen.



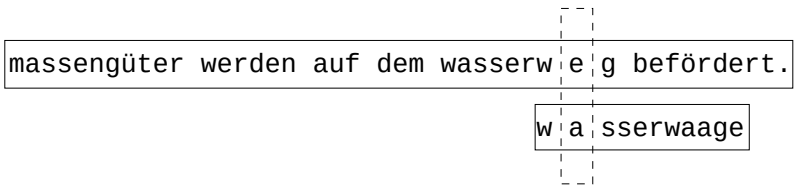

Das nun dargestellte Verfahren vermeidet dies und kommt deshalb sogar mit einem linearen Aufwand aus.

**Beispiel**

Wir suchen das Muster 'Wasserwaage' in einem Text und nehmen dabei folgende Situaion an:




Es wird hier erstmals ein Unterschied in der Position 8 des Musters entdeckt. Intuitiv ist klar, daß wir der vorliegenden Informationen das Muster sofort auf die folgende Position verschieben können:

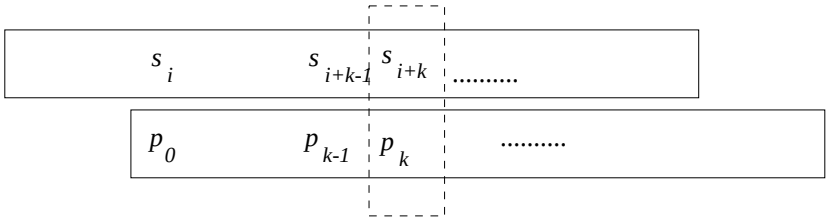


Als nächstes ist dann das zuletzt geprüfte Zeichen "e" des Teststring mit dem zweiten Zeichen "a" des Musters zu vergleichen. Der Anfang des Musters, der hier nur aus dem Zeichen "w" besteht muß nicht noch einmal überprüft werden.

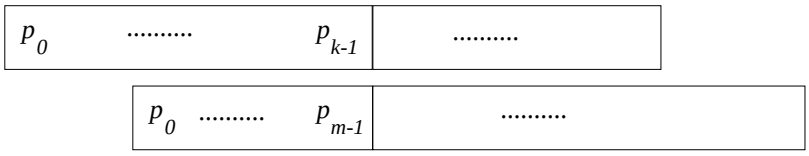
### Methode

Das oben demonstrierte Vorgehen können wir nun wie folgt präzisieren:

- Das Muster $\boldsymbol{p}$ wird an der Position *i* des Textes angelegt.
- Die ersten $k$ Zeichen von Textstring und Muster stimmen überein.
- An der Musterposition $k$ wird erstmals ein Unterschied festgestellt: $s_{i+k} \neq p_k$ .




- Es wird ein Endstück maximaler Länge im Muster-Anfang $p_0 \cdots p_{k-1}$ gesucht, das mit einem Muster-Anfang $p_0 \cdots p_{m-1}$ übereinstimmt.




- Die Länge *m* dieses maximalen Anfangs ordnen wir der Musterposition *k* zu. Für den Fall, daß an der Musterposition *k* erstmals ein Unterschied festgestellt wird, also $s_{i+k} \neq p_k$ ist, muß als nächstes $s_{i+k}$ mit $p_m$ verglichen werden. Diese Werte legen wir vor Beginn der Suche in einem Feld `next[m]` ab, so daß `next[k]=m` ist.
- Wenn bereits in der Position `j=0` keine Übereinstimmung zwischen $p_0$ und $s_i$ vorliegt, können wir j unverändert auf 0 stehen lassen und `i` um 1 fortschalten. Im folgenden Programm wird das dadurch erreicht, daß `next[0]=-1` gesetzt wird.

Das folgende C-Programm implementiert dieses Verfahren.

```
int  KMPSearch  (char *s, char *p)
{
  int i, j;                     // Suchzeiger
  int n = strlen(s);            // Stringlänge
  int m = strlen(p);            // Musterlänge

  InitNext(p);                  // Verweisfeld initialisieren
                                // Suchschleife
  for (i=0, j=0; j<m && i<n; i++, j++)
①   while ((j>=0) && (s[i] != p[j]))  j=next[j];
  if (j==m)   return (i-m);
  else        return (-1);
}
```

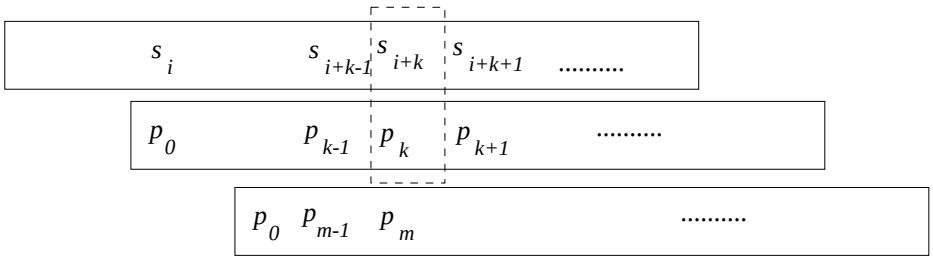
① Für s[i]=p[j] bewirkt die while-Schleife nichts und die for-Schleife schaltet nur die Indizes fort. Falls s[i]≠p[j] , dann wird die nächste Vergleichsposition für j gesetzt.

Wir müssen bei diesem Programm noch die vorab durchzuführende Initialisierung des Feldes next genauer betrachten:

### Initialisierung der next-Zeiger

Zunächst setzen wir aus dem oben geschilderten Grund `next[0]=-1.`

Falls next`[0]=-1, .....,` `next[k]=m` schon korrekte Werte besitzen, liegt die folgende Situation vor, wenn an der Stelle `k` des Musters eine Nicht-Übereinstimmung mit dem Textstring festgestellt wird:




`next[k]=m` gibt an, daß das Endstück maximaler Länge von $p_0 \cdots p_{k-1}$, das mit einem Anfang des Musters übereinstimmt, $p_0 \cdots p_{m-1}$ ist. Wenn `next[1], next[2], ... , next[k]` schon bekannt sind, haben wir dann zwei Möglichkeiten für den Wert `next[k+1]`:

1. Es ist next[k]=m und und $p_k = p_m$:
   Dann ist $p_0 \cdots p_m$ aber auch das Endstück maximaler Länge in $p_0 \cdots p_k$ und daher ist `next[k+1]=m+1`.
2. Es ist $p_k \neq p_m$:
   In diesem Fall müssen wir ein Endstück maximaler Länge in $p_1 \cdots p_{m-1}$ finden, das mit dem entsprechenden Anfang übereinstimmt. Dieses ist bereits bekannt und hat den Wert `next[m]=next[next[k]]`.

Das folgende C-Code realisiert diese Methode:

```
void InitNext (char *p)
{
  int i, j;                    // Laufzeiger
  int m = strlen(p);           // Musterlänge

  next[0] = -1;
  for (i=0, j=-1;i<m;i++, j++, next[i]=j)
    while ((j>=0) && (p[i] != p[j]))  j=next[j];
}
```

### Analyse

Der Gesamtaufwand des Verfahrens ergibt sich aus der vorbereitenden Initialisierung und der eigentlichen Suche. Die Argumentation ist für beide Routinen identisch:

- Die Zeiger `i` und `j` werden stets synchron erhöht, insgesamt genau `N` bzw. `M` mal.
- der Zeiger `i` wird nie zurückgestellt
- Für jeden Wert von i kann die while-Schleife den Wert von j mehrfach zurückstellen, insgesamt aber nur N-mal in KPMSearch und M-mal in InitNext.

Der Aufwand für KPMSearch beträgt deshalb $\boldsymbol{O}(N)$ und der Aufwand für `InitNext` beträgt $\boldsymbol{O}(M)$ Schritte. Daher ist der Gesamtaufwand für das Knuth-Morris-Pratt-Verfahren $\boldsymbol{O}(N+M)$.

## 5.3.3 Unscharfe Suche mit der Levenshtein-Distanz

### Motivation

Bei der Suche nach Mustern in einem Text ist das gesuchte Muster oft nicht exakt bekannt. Die Ursachen dafür können verschieden sein, z.B.

- Schreibfehler: Regemsburg, Regesnburg, Regenburg
- korrekte Varianten : Thunfisch – Tunfisch , Schiffahrt – Schifffahrt

Damit Treffer auch dann gefunden werden, wenn keine exakte Übereinstimmung zwischen dem Textstring und dem Suchmuster vorliegen, werden sehr unterschiedliche Methoden eingesetzt.

Ein auf V.I. Levenshtein zurückgehender Ansatz baut auf einem Distanzbegriff für Textstrings auf und meldet immer dann einen Treffer, wenn diese Distanz unterhalb einer Schwelle liegt.

### Die Levenshtein-Distanz

Wir nehmen elementare Operationen zum Abändern von Mustern und deren "Kosten" an:

- Ersetzen eines Zeichens durch ein anderes Kosten = p
- Einfügen eines Zeichens Kosten = q
- Löschen eines Zeichens Kosten = r

Als gewichtete Levenshtein-Distanz zweier Strings x und y bezeichnen wir dann:

$WLD(x, y) =$ minimale Kosten der Umwandlung von x in y

Wenn die Umwandlung $k_i$ Ersetzungen, $m_i$ Einfügungen und $n_i$ Löschungen erfordert, dann erhalten wir

$$WLD(x, y) = \min_i (p * k_i + q * m_i + r * n_i)$$

### Rekursive Bestimmung der Levenshtein-Distanz

Es seien $x_i$ und $y_j$ zwei Strings mit i bzw. j Zeichen. Wir teilen den Aufwand zur Herstellung der vollständigen Übereinstimmung auf in den Aufwand, der notwendig ist, um alle bis auf die letzte Elementar-Operation durchzuführen und den Aufwand für diese letzte Operation:

$$WLD(x, y) = \min \begin{bmatrix} WLD(x_{i-1}, y_{j-1}) + p_{ij} \\ + WLD(x_i, y_{j-1}) + q \\ + WLD(x_{i-1}, y_j) + r \end{bmatrix}$$

Dabei ist: $p_{ij} = \begin{cases} 1 & \text{falls das letzte Zeichen in x und y verschieden sind} \\ 0 & \text{sonst} \end{cases}$

### Beispiel

WLD("OCMPRT" , "COMPUTER") ist das Minimum der folgenden drei Varianten:

1. "OCMPR" in "COMPUTE" transformieren und dann "T" in "R" umwandeln:
   [OCMPR]T [COMPUTE]R
2. "OCMPRT" in "COMPUTE" transformieren und dann "R" anhängen:
   [OCMPRT] [COMPUTE]R
3. "OCMPR" in "COMPUTER" transformieren und dann "T" löschen:
   [OCMPR]T [COMPUTER]

Als Rekursionsbasis verwenden wir:

$WLD(x_0, y_j) = jq$ Aufwand, um aus dem Leerstring $x_0$ einen mit j Zeichen zu machen

$WLD(x_i, y_0) = ir$ Aufwand, um aus einem String mit i Zeichen den Leerstring zu machen

$WLD(x_0, y_0) = 0$ Aufwand, um aus dem Leerstring wieder einen zu machen

### Beispiel zur Mustertransformation nach Levenshtein

Gesucht ist eine Transformation von "OCMPRT" in "COMPUTER" . Dazu gehen wir im konkreten Fall wie folgt vor:

| | | | |
|---|---|---|---|
| 1. Schritt: | O Löschen | OCMPRT | → CMPRT |
| 2. Schritt: | O einfügen | | → COMPRT |
| 3. Schritt: | R in U umwandeln | | → COMPUT |
| 4. Schritt: | E einfügen | | → COMPUTE |
| 5. Schritt: | R einfügen | | → COMPUTER |

Damit ist noch nicht gesagt, daß diese Umwandlung die billigste ist, zumal das auch von p, q, r abhängt.

### C-Implementierung der Levenshtein-Methode

Die rekursive Definition der Distanz WLD eröffnet prinzipiell die Möglichkeit, sie rekursiv zu berechnen, alle möglichen Varianten der Umwandlung zu ermitteln und daraus das Minimum der Kosten zu bestimmen. Wesentlich weniger aufwendig ist eine iterative Lösung, die auf der Technik der dynamischen Programmierung beruht und die in dem folgenden Programmcode realisiert ist.

```
int LvDist(char* wort1, char* wort2)
{
  int i, j, d[wMax+1][wMax+1];

  d[0][0] = 0;
  for(j=1;j<=wMax;j++)  d[0][j]=d[0][j-1] + q;
  for(i=1;i<=wMax;i++)  d[i][0]=d[i - 1][0] + r;
  for(i=1;i<=strlen(wort1);i++)
    for(j=1;j<=strlen(wort2);j++)
      d[i][j]=min3(d[i-1][j-1] + pp(wort1,wort2,i - 1,j - 1),
        d[i][j-1] + q,
        d[i-1][j] + r);
  return(d[strlen(wort1)][strlen(wort2)]);
}
```

Als Hilfsfunktionen werden darin verwendet:

```
// Minimum von 3 Zahlen:
inline int min3(int x,int y,int z)
       { if(x<y) y=x; if(y<z) z=y; return(z); }

// Kosten der Erstezung eines Buchstabens an Position (x,y)
inline int pp(char* wort1, char* wort2, int x, int y)
       { if(wort1[x] == wort2[y]) return(0);  else return(p);
}
```

### Beispiel zur Berechnung der Levenstein-Dustanz

Wir nehmen der Einfachheit halber p = q = r = 1 an. Das Verfahren arbeitet mit einer Matrix, die zunächst wie folgt initialisert wird:

| | ε | C | O | M | P | U | T | E | R |
|---|---|---|---|---|---|---|---|---|---|
| ε | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| O | 1 | | | | | | | | |
| C | 2 | | | | | | | | |
| M | 3 | | | | | | | | |
| P | 4 | | | | | | | | |
| R | 5 | | | | | | | | |
| T | 6 | | | | | | | | |

Die Zeile ε enthält in der Spalte j den Aufwand, um aus dem Leerstring ε die ersten j Zeichen von "COMPUTER" zu erzeugen durch Einfügen.

Die Spalte ε enthält in der Zeile i den Aufwand, um aus einem i Zeichen langen Anfang von "OCMPRT" den Leerstring zu erzeugen durch Löschen.

Danach wird der Rest der Matrix zeilenweise gefüllt. Die gesuchte Distanz steht dann auf der Position ganz unten rechts: WLD("OCMPRT" , "COMPUTER") = 4

| | ε | C | O | M | P | U | T | E | R |
|---|---|---|---|---|---|---|---|---|---|
| ε | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| O | 1 | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| C | 2 | 1 | 2 | 2 | 3 | 4 | 5 | 6 | 7 |
| M | 3 | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 7 |
| P | 4 | 3 | 3 | 3 | 2 | 3 | 4 | 5 | 6 |
| R | 5 | 4 | 4 | 4 | 3 | 3 | 4 | 5 | 5 |
| T | 6 | 5 | 5 | 5 | 4 | 4 | 3 | 4 | **5** |

# 6 Bäume

Bäume sind eine der wichtigsten Datenstrukturen der Informatik und Datenverarbeitung. Sie sind daher sehr intensiv untersucht worden und finden Anwendung in unterschiedlichsten Bereichen:

- als Suchbäume zum Finden von Elementen in geordneten Mengen
- als Entscheidungsbäume zur Organisation sukzessiver Entscheidungen,
- als Strukturbäume zur Repräsentation der syntaktischen Struktur von Programmen,
- als Datenstruktur zur Organisation eines Sortierprozesses beim Heap Sort-Verfahren.

Grundlegende Begriffsbildungen und Verfahren zum Umgang mit Baumstrukturen sind Gegenstand der folgenden Abschnitte.

## 6.1 Begriffe im Zusammenhang mit Bäumen

Im folgenden gehen wir aus von einer endlichen Menge $\boldsymbol{N}$ von *Knoten* (nodes) und einer binären Relation $\boldsymbol{V}$ über $\boldsymbol{N}$. Die Elemente $(p,q) \in \boldsymbol{V}$ heißen *Kanten* (vertices). Ist $(p,q) \in \boldsymbol{V}$ eine Kante, so nennen wir

- $p$ den Vorgänger- oder Vaterknoten von $q$ und
- $q$ den Nachfolger- oder Sohn-Knoten von $p$.

### Baum

Eine Datenstruktur $\boldsymbol{B} = (\boldsymbol{N},\boldsymbol{V})$ heißt dann ein *Baum*, wenn $\boldsymbol{B}$ die folgenden Eigenschaften besitzt:

- Es gibt genau einen Knoten $q \in \boldsymbol{N}$, der keinen Vorgänger besitzt. Diesen Knoten bezeichnen wir als die *Wurzel* von $\boldsymbol{B}$.
- Alle Knoten von $\boldsymbol{B}$, außer dem Wurzelknoten besitzen genau einen Vorgängerknoten.

### Blätter und innere Knoten

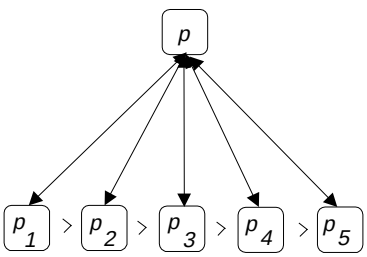
Als *Blätter* bezeichnen wir diejenigen Knoten, die keinen Nachfolger besitzen. Alle anderen Knoten heißen *innere Knoten* von $\boldsymbol{B}$.

### Pfad

Ist $\mathbf{B}=(\mathbf{N},\mathbf{V})$ ein Baum, dann heißt die Folge $p_0, p_1 \cdots p_k$ ein *Pfad* in $\boldsymbol{B}$, wenn $(p_i, p_{i+1}) \in \boldsymbol{V}$ ist für alle Indizes $0 \le i < k$. Der Index $k$ gibt dann die Länge des Pfades an.

### Geordneter Baum

Ein Baum $\boldsymbol{B} = (\boldsymbol{N},\boldsymbol{V})$ heißt geordnet, wenn die Söhne eines jeden Knotens eine vollständig geordnete Menge bilden. Sie lassen sich dann z.B. gemäß dieser Ordnung linear aufsteigend verketten.



### Höhe eines Baumes

Als *Höhe* eines Baumes bezeichnen wir die größtmögliche Pfadlänge in ihm, d.h. die Länge des längsten Weges zwischen der Wurzel und einem Blatt.

### Tiefe eines Knotens

Die *Tiefe* eines Knotens ist sein Abstand von der Wurzel, d.h. die Länge des Pfades von der Wurzel zu ihm.

### Rang eines Knotens

Unter dem *Rang* eines Knotens verstehen wir die Anzahl seiner Nachfolger.

### Ordnung eines Baumes

Der maximale Rang eines Knotens im Baum ***B*** heißt die *Ordnung* des Baumes ***B***.

### Binärbaum - Mehrwegebaum

Ein *Binärbaum* ist ein geordneter Baum der Ordnung 2. Wir können daher sinnvoll vom linken und rechten Nachfolger eines Knotens sprechen. Bäume mit einer höheren Ordnung als 2 heißen *Mehrwegebäume*.

Eine wichtige Klasse von Mehrwegebäumen sind die *B-Bäume*: die Anzahl der Nachfolger eines Knotens muß für sie stets zwischen einem minimalen und maximalen Wert liegen.

### Vollständiger Baum

Ein Baum heißt vollständig, wenn

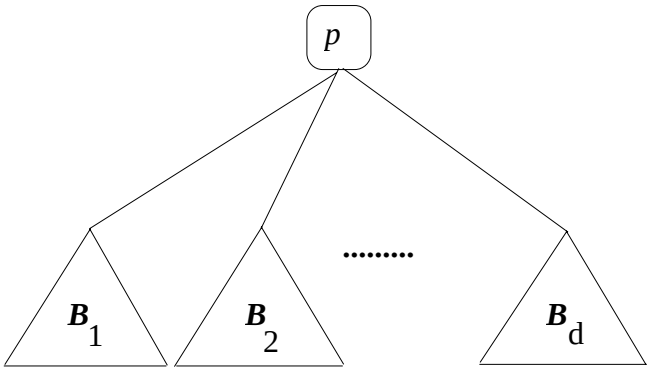
- auf jedem Niveau die maximal mögliche Zahl von Knoten existiert und
- alle Blätter die gleiche Tiefe besitzen.

### Rekursive Definition von Bäumen

Bäume der Ordnung $d$ $(d \geq 1)$ kann man auch rekursiv definieren wie folgt:

- ein isolierter Knoten stellt einen Baum der Ordnung $d$ dar.


- wenn $\boldsymbol{B}_1, \boldsymbol{B}_2 \cdots \boldsymbol{B}_d$ bereits Bäume der Ordnung $d$ sind, dann erhält man einen weiteren Baum der Ordnung $d$, wenn man sie an einen neuen Wurzelknoten als Söhne anhängt.




Mit dieser Festlegung erhält man nur vollständige Bäume der Ordnung *d*,. Beliebige Bäume der Ordnung d erhält man, wenn man den "leeren Baum" als einen Baum der Ordnung $d$ hinzunimmt.

### Operationen auf Bäumen

Bäume stellen eine Datenstruktur zum Speichern von Schlüsseln dar. Dementsprechend sind die wichtigsten Operationen auf Bäumen

- das Suchen nach einem vorgegebenen Schlüssel,
- das Einfügen eines neuen Knotens mit einem vorgegebenen Schlüssel und
- das Entfernen eines Knotens mit einem vorgegebenen Schlüssel.

Bei manchen Anwendungen kommen Such-Operationen wesentlich häufiger vor als das Einfügen und Löschen von Knoten. In diesen Fällen wird man die Knoten zweckmäßig so anordnen, daß häufig gesuchte Elemente schneller gefunden werden können.

Wenn Einfügen und Löschen häufig auftretende Operationen sind, dann muß man darauf achten, daß dabei die Struktur der Bäume nicht degeneriert. Im Extremfall könnten lineare Listen entstehen. Durch Balancierungstechniken kann man dafür sorgen ,daß ausgeglichene Bäume entstehen, in denen die Operationen Suchen, Einfügen und Löschen mit logarithmischem Aufwand möglich sind.

## 6.2 Typen binärer Bäume

Unter der Annahme, daß jeder Schlüssel nur einmal im Baum auftreten darf, wollen wir nun Suchbäume so organisieren, daß

- vorhandene Schlüssel schnell auffindbar sind und
- im Fall, daß der gesuchte Schlüssel nicht vorkommt, schnell eine Fehlanzeige möglich ist.

Wir unterscheiden zwei gleichwertige Ansätze, nämlich

- *binäre Suchbäume*: Sie speichern die Schlüssel in den inneren Knoten
- *Blattsuchbäume*: Sie speichern die Schlüssel nur in den Blättern.

### Binäre Suchbäume

Binärere Suchbäume sind dadurch charakterisiert, daß für innere Knoten $p$ mit Schlüssel $s_p$ gilt:

- Der linke Unterbaum von $p$ enthält nur Knoten mit Schlüsseln $s < s_p$ .
- Der rechte Unterbaum von $p$ enthält nur Schlüssel $s > s_p$ .
- Die Blattknoten speichern keine Information.

Die folgende Abbildung zeigt einen solchen binären Suchbaum mit ganzen Zahlen als Schlüssel:

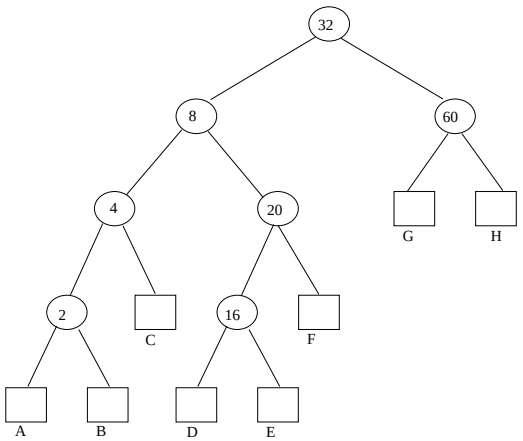



**Abbildung 6.1:** Binärer Suchbaum

Bei binären Suchbäumen repräsentieren die Blätter Intervalle in der Menge der möglichen Schlüssel. Im Beispiel der Abbildung 6.1 sind dies für die Blätter *A, B, ... , G* die Intervalle

(-∞,2) (2,4) (4,8) (8,16) (16,20) (20,32) (32,60) (60,+∞)

### Suchen in einem binären Suchbaum

Die beschriebene Organisation der Schlüssel in einem binären Suchbaum legt die folgende systematische Suche nach einem Schlüssel nahe, die unten als Pseudocode angegeben wird: Eine Prozedur SEARCH1 sucht nach einem Knoten mit den Schlüssel $s$ in dem Baum unterhalb des Knotens $p$. Dabei bezeichnen $p_l$ und $p_r$ den linken bzw. rechten Nachfolger von $p$ und $s_p$ den Schlüssel im Knoten $p$.

```
SEARCH1(s, p)
{ Suche nach einem Schlüssel s unterhalb eines Knotens p in e. binären Suchbaum}
IF { p innerer Knoten} THEN IF { s < s_p } THEN SEARCH1( s, p_l )
                                          ELSE IF ( s > s_p ) THEN SEARCH1( s, p_r )
                                                             ELSE { gefunden ! }
                         ELSE { Schlüssel s kommt nicht vor ! }
```

### Blatt-Suchbäume

Blatt-Suchbäume enthalten nur in den Blättern Schlüssel. Innere Knoten enthalten ausschließlich Wegweiser, die die Suche nach einem Schlüssel lenken. Für die Speicherung der Schlüssel gilt wie bei den binären Suchbäumen:

- Der linke Unterbaum von $p$ enthält nur Knoten mit Schlüsseln $s \le s_p$
- Der rechte Unterbaum von $p$ enthält nur Schlüssel $s \ge s_p$ .

Damit eignen sich als Wegweiser-Information z.B. die größten, im linken Unterbaum vorkommenden Schlüssel. Abbildung 6.2 zeigt einen solchen Blatt-Suchbaum.

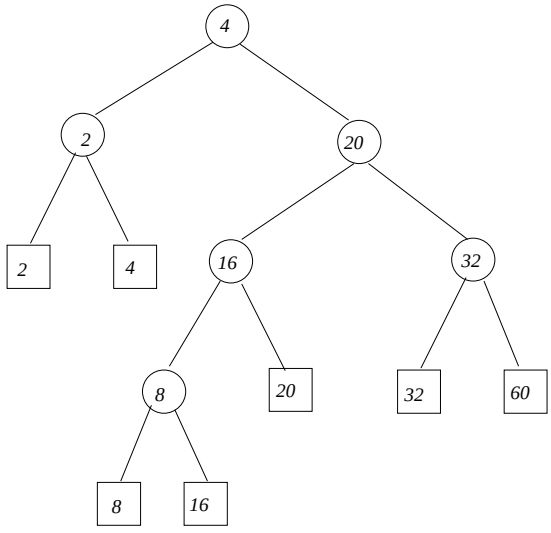



**Abbildung 6.2**: Blatt-Suchbaum

### Suchen in einem Blatt-Suchbaum

Die beschriebene Organisation der Schlüssel in einem Blatt-Suchbaum legt die folgende systematische Suche nach einem Schlüssel nahe, die als Pseudocode angegeben wird. Die Bezeichnungen $s_p$ , $p_l$ , $p_r$ sind wie bei den binären Suchbäumen gewählt.

```
SEARCH2(s, p)
{ Suche nach einem Schlüssel s unterhalb eines Knotens p im Blatt-Suchbaum}
IF { p innerer Knoten} THEN IF { s ≤ s_p } THEN SEARCH2 (s, p_l)
                                           ELSE SEARCH2 (s, p_r)
                       ELSE IF { s = s_p } THEN { Schlüssel s gefunden ! }
                                           ELSE { Schlüssel s kommt nicht vor ! }
```

### Entscheidungsbäume

Zur systematischen Organisation von aufeinanderfolgenden ja/nein-Entscheidungen verwendet man Entscheidungsbäume. Jedem Knoten ist dabei ein Test zugeordnet, der positiv oder negativ ausfallen kann. Das Testergebnis wird als Anweisung interpretiert, im linken oder rechten Unterbaum weiterzulaufen, z.B. bei "ja" nach links.

Diese Technik kann verwendet werden, um ein Sortierverfahren, das auf Schlüsselvergleichen beruht darzustellen. Die den inneren Baumknoten zugeordneten Entscheidungen sind Vergleiche von Listenelementen. Die Blattknoten geben die Permutation der Schlüssel an, welche sie sortiert. Ein Beispiel für einen solchen Entscheidungsbaum zeigt die Abbildung 6.3.

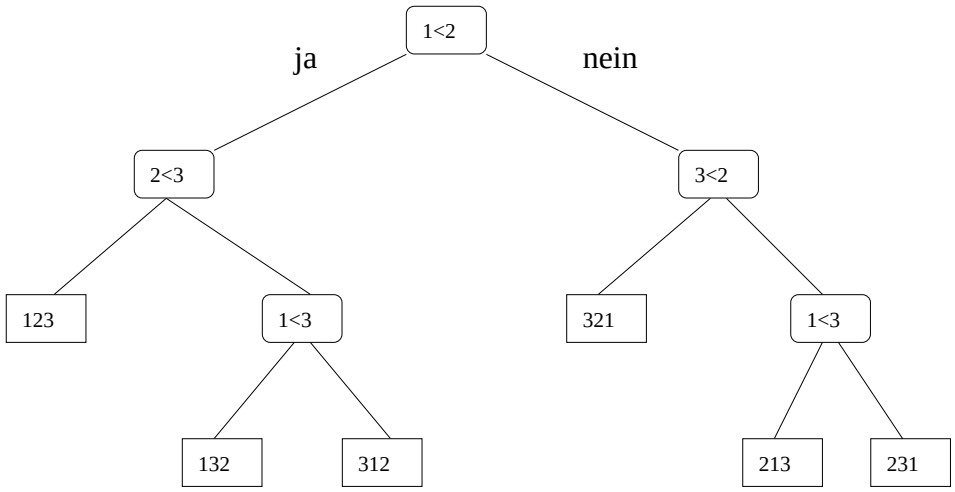



**Abbildung 6.3 :** Entscheidungbaum für einen Sortierprozeß

## 6.3 Implementierungstechniken für Bäume

Bäume werden in der Regel als verkettete Listen implementiert, wobei die Knoten des Baumes als Records realisiert sind. Diese Realisierung von Bäumen heißt auch *natürliche Implementierung* binärer Bäume. Eine C++-Klassendefinition dafür könnte wie folgt lauten:

```
class Node
{  public:
    Node (int Key, char *Info="")
    {  key   = Key;
       strncpy (info, Info, INFLEN); info[INFLEN]=0;
       left  = NULL;
       right = NULL;
    }
    virtual ~Node () {}

    void  getKey   (int &Key)       { Key = key;   }
    void  setKey   (int Key)        { key = Key;   }
    void  getInfo  (char* Info)     { strncpy (Info,info,INFLEN);
                                      Info[INFLEN]=0;       }
    void  setInfo  (char* Info)     { strncpy (info,Info,INFLEN);
                                      info[INFLEN]=0;       }
    void  setLeft  (Node *leftPtr)  { left = leftPtr;       }
    Node  *getLeft (void)           { return (left);        }
    void  setRight (Node *rightPtr) { right = rightPtr;     }
    Node  *getRight(void)           { return (right);       }

    int  key;                       // Knoten-Schlüssel
    char info[INFLEN];              // Knoten-Information
    Node *left;                     // Zeiger zum linken  Nachfolger
    Node *right;                    // Zeiger zum rechten Nachfolger
};

class Tree
{
  public:
    Tree    ()   { root=NULL; }
    virtual ~Tree         ()    { DestroyTree(root); }
    void    insertNode    (int  newKey, char *newInfo);
    void    deleteNode    (int  delKey);
    Node    *findKey      (int  sKey);
    void    inOrder       ();
    void    preOrder      ();
    void    postOrder     ();
    void    printTree     ();
    void    listTree      ();
    int     treeHeight    ();

     Node    *root;

   private:
      virtual void DestroyTree(Node *rootPtr);
      void printSubTree    (Node *subRoot, int width);
      void listSubTree     (Node *subRoot);
      void listInOrder     (Node *root);
      void listPreOrder    (Node *root);
      void listPostOrder   (Node *root);
      void delNode         (Node **root, int delKey);
      Node *PredsymSucc    (Node *node);
      int  treeHeight1     (Node *root);
};
```

Abbildung 6.4 zeigt die Realisierung des Suchbaumes aus Abbildung 6.1 durch Records mit diesem Aufbau. Dabei wird auf die Blattknoten ganz verzichtet, da diese keine Information enthalten. Die Zeiger auf sie haben dann den Wert NULL.

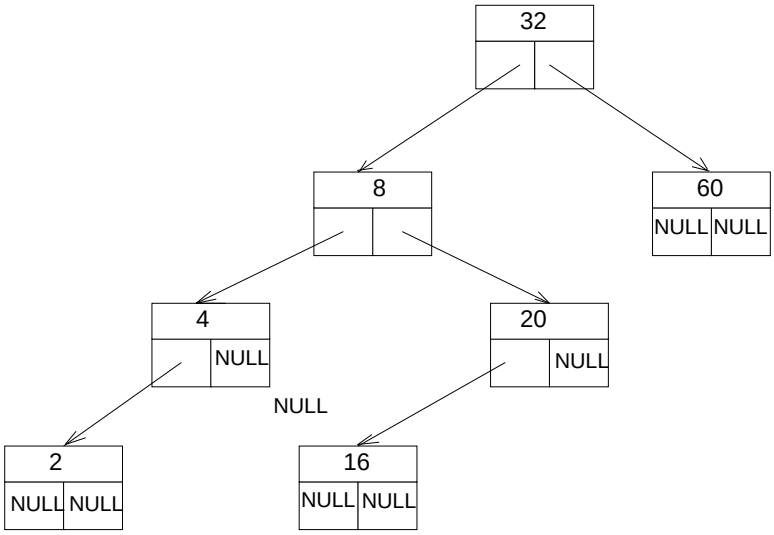
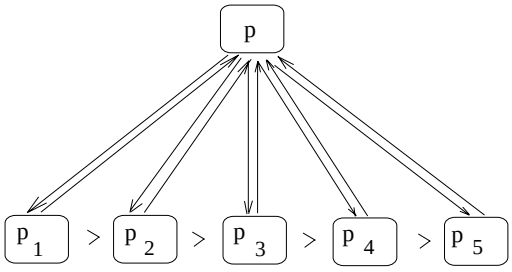



**Abbildung 6.4:** Natürliche Implementierung des binären Suchbaums aus Abbildung 6.1

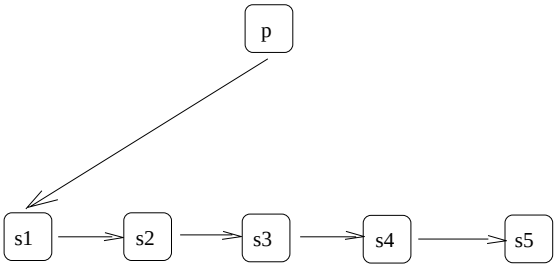
### Alternative Implementierungen für Bäume

Andere Verkettungstechniken für Bäume sind uU. günstiger als die oben gezeigte:

– *Doppelte Verkettung*:
Sie ermöglicht einfaches Durchlaufen der Baumknoten in beliebiger Reihenfolge.




– *Lineare Verkettung der Nachfolgerknoten*
Insbesondere bei Bäumen höherer Ordnung ist es zweckmäßig,statt mit einem Zeigervektor auf alle Nach- folger-Knoten zu verweisen, nur auf den ersten Nachfolger zu zeigen und an diesen die weiteren Nachfolger linear anzuketten.



– *Speicherung in einem Array*
Eine weitere Möglichkeit besteht darin, die Knoten eines Baumes in einem Array zu speichern. Man erreicht dann die Knoten durch eine Adreßrechnung. Diese Technik haben wir beim Heapsort-Verfahren benutzt, um einen "Heap" zu speichern. Sie setzt jedoch einen vollständigen Baum voraus.

Im weiteren Verlauf legen wir in der Regel die natürliche Implementierung binärer Suchbäume zugrunde, wie sie in Abbildung 6.4 gezeigt ist.

## 6.4 Operationen mit binären Bäumen

Dieser Abschnitt stellt implementierungstechnische Aspekte typischer Baum-Operationen vor: Aufbau eines binären Suchbaumes sowie Suchen, Einfügen und Entfernen von Knoten.

### Aufbau von binären Suchbäumen durch Einfügen von Knoten

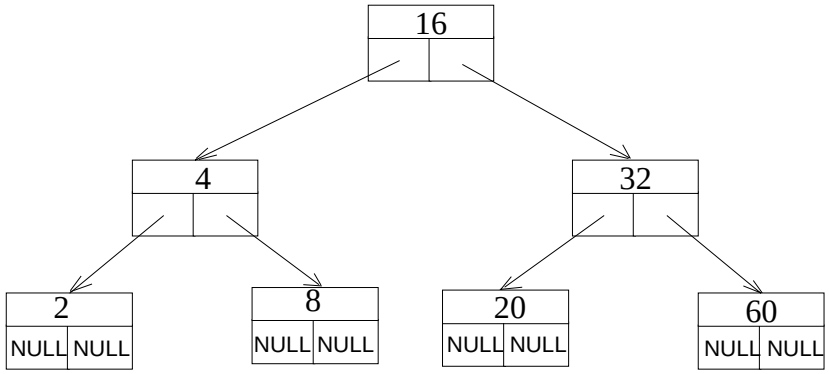
Aus einer vorgegebenen Menge paarweise verschiedener Schlüssel soll ein Suchbaum erzeugt werden. Dazu fügen wir die Elemente nacheinander so in einen zunächst leeren Baum ein, daß stets eine Suchbaum-Struktur erhalten bleibt. Der folgende Pseudocode beschreibt den wesentlichen Elementarschritt, nämlich das Einfügen eines einzelnen Knotens mit Schlüssel s in einen Suchbaum mit der Wurzel p:

```
InsertNode ( s,p)
{ fügt einen neuen Knoten mit Schlüssel s in einen Baum mit der Wurzel p ein }
IF { Baum mit Wurzel p leer }
   THEN { hänge einen neuen Knoten mit Schlüssel s an p an. }
   ELSE IF ( s ≤ s_p ) THEN InsertNode ( s, p_l )
                       ELSE IF ( s > s_p ) THEN InsertNode ( s, p_r )
                                           ELSE { Schlüssel kommt im Baum schon vor ! }
```

### Beispiel

Abbildung 6.5 zeigt zwei Suchbäume, die nach der obigen Methode mit der selben Schlüsselmenge erzeugt worden sind. Die Reihenfolge, mit der die Schlüssel in den Baum eingetragen wurden war jedoch unterschiedlich. Dies hat Konsequenzen für die Struktur des Suchbaumes: Im Extremfall erhält man sogar eine lineare Liste. Wann ist dies der Fall ?

a)




b)

b)

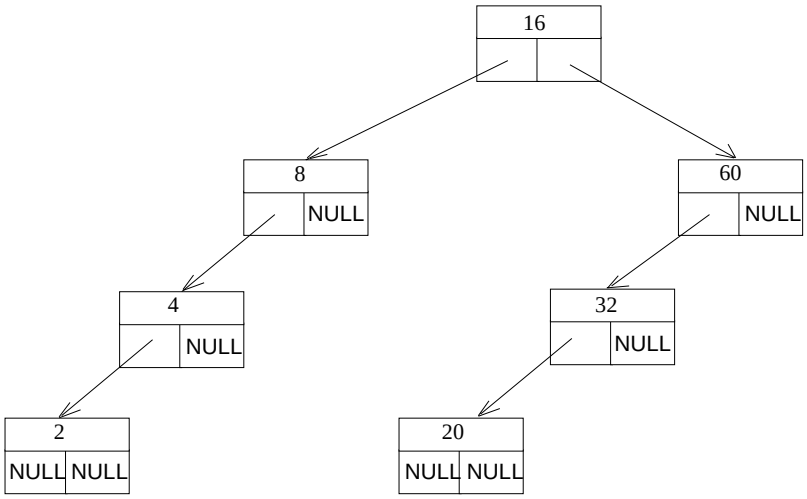



**Abbildung 6.5:** Suchbäume zur gleichen Schlüsselmenge, verschiedene Aufbau-Reihenfolge
a) Suchbaum, aufgebaut mit der Schlüsselfolge 16, 32, 4, 60, 20, 8, 2
b) Suchbaum, aufgebaut mit der Schlüsselfolge 16, 8, 4, 2, 60, 32, 20

### C++ - Methode

```
void Tree::insertNodeR (int newKey, char *newInfo)
{  insertNodeR1 (&root, newKey, newInfo); }

void Tree::insertNodeR1 (Node **root, int newKey, char *newInfo)
{  if (!(*root)) { *root = new Node(newKey, newInfo);  }
   else if (newKey < (*root)->key)        // links einfuegen
           insertNodeR1(&((*root)->left), newKey, newInfo);
   else if (newKey > (*root)->key)        // rechts einfuegen
           insertNodeR1(&((*root)->right), newKey, newInfo);
   else    cout << "\nSchluessel ist schon vorhanden." << endl;
}
```

### Suchen in einem binären Suchbaum

Das folgende Verfahren sucht zu einem vorgegebenen Schlüssel in einem binären Suchbaum den Knoten mit diesem Schlüssel. Von der Wurzel her wird der Baum rekursiv durchlaufen. Es wird ein Zeiger auf den Knoten zurückgeliefert, dessen Schlüssel mit dem gesuchten übereinstimmt. Als Fehlanzeige wird ein NULL-Zeiger zurückgeliefert.

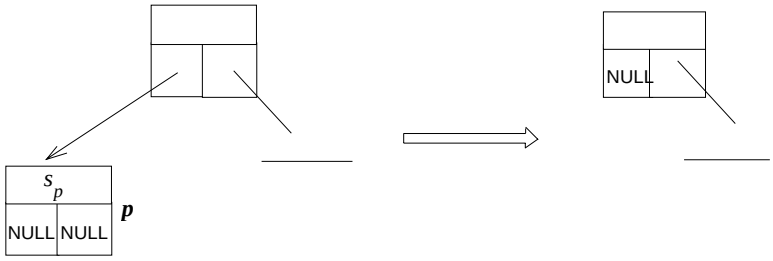
### C++ - Methode

```
Node *Tree::findKeyR (int sKey)
{  return(findKeyR1(root, sKey)); }

Node *Tree::findKeyR1 (Node *root, int sKey)
{  if (!root)   return (NULL);
   else if (sKey < root->key)
        root = findKeyR1 (root->left,sKey);
   else if (sKey > root->key)
        root = findKeyR1 (root->right,sKey);
   return(root);
}
```

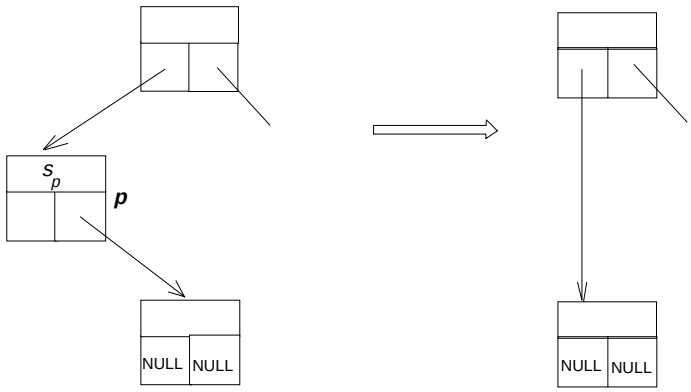
### Löschen von Knoten in einem binären Suchbaum

Um einen Knoten $p$ in einem binären Suchbaum zu löschen, müssen wir ihn zunächst aufgrund seines Schlüssels $s_p$ finden. Das Entfernen muß dann so durchgeführt werden, daß die Suchbaumstruktur erhalten bleibt. Es sind dabei drei Fälle zu unterscheiden:

**1. Fall:** Der zu löschende Knoten $p$ hat keine inneren Knoten als Nachfolger.
In dieser Situation setzen wir den Zeiger auf ihn im Vaterknoten auf NULL und können dann den Knoten selbst löschen.




**2. Fall:** Der zu löschende Knoten $p$ hat genau einen inneren Knoten als Nachfolger.
In dieser Situation ersetzen wir den Zeiger auf ihn im Vaterknoten durch einen Zeiger auf den einzigen Nachfolger von $p$ und können dann den Knoten p selbst löschen.

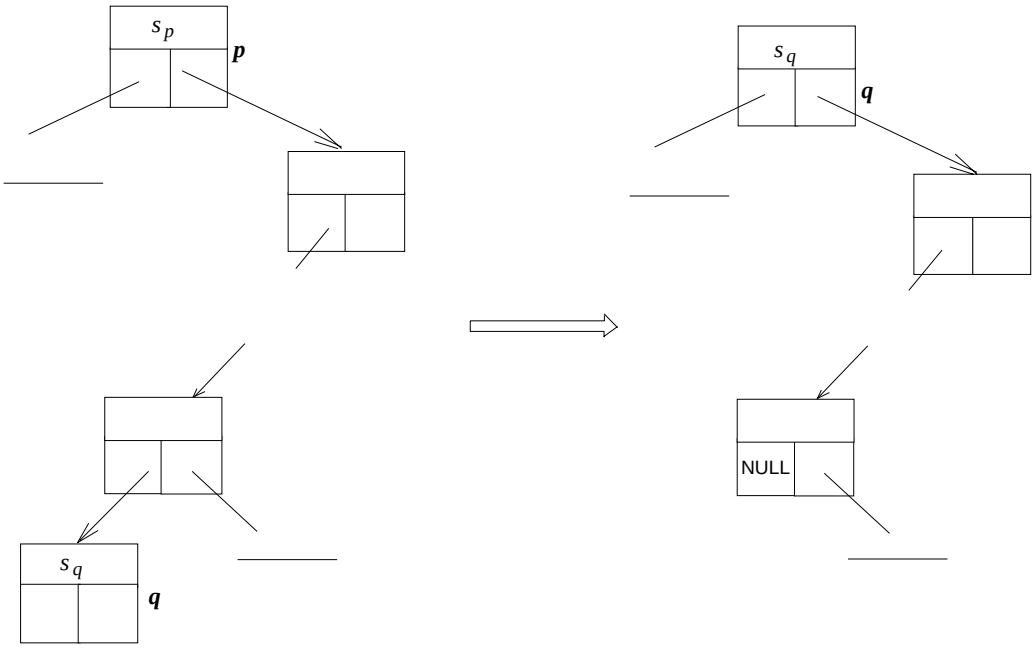



**3. Fall:** Der zu löschende Knoten $p$ hat zwei innere Knoten als Nachfolger.

In diesem Fall ersetzen wir den Schlüssel im Knoten $p$ durch denSchlüssel desjenigen Knotens $q$ aus dem rechten Unterbaum von p, der dort den kleinsten Schlüssel besitzt. Danach entfernen wir $q$. Die Methode heißt daher auch *Schlüsseltransfer*.

Dabei bleibt der Baum als ein Suchbaum nach der Definition in Abschnitt 6.2 erhalten. Der Knoten $q$ kann höchstens einen *rechten* Nachfolger haben, da sonst ein innerer Knoten mit noch kleinerem Schlüssel existieren würde. Mit $q$ haben wir einen Knoten zu entfernen, auf den einer der Fälle 1 oder 2 zutrifft. Die Abbildung auf der nächsten Seite zeigt dies schematisch.

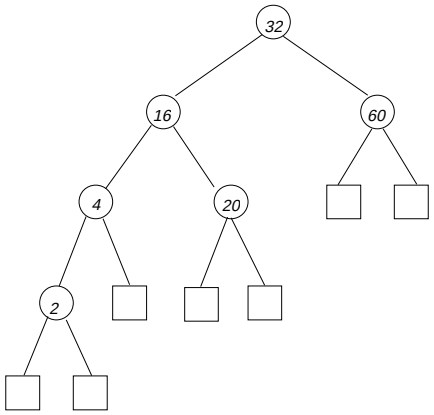
Der Knoten $q$ heißt der *symmetrische Nachfolger* von $p$ .




Statt des symmetrischen Nachfolgers könnten wir analog auch den symmetrischen Vorgänger zum Austausch mit dem zu entfernenden Knoten verwenden, oder abwechselnd eine dieser beiden Alternativen.

### Beispiel

Wenn wir aus dem Suchbaum der Abbildung 6.1 den Knoten mit dem Schlüssel 8 entfernen, dann wird dieser ersetzt durch den symmetrischen Nachfolger mit dem Sclüssel 16 und wir erhalten den folgenden Suchbaum



**C-Methode**

```
void Tree::deleteNode (int delKey)
{  delNode (&root, delKey); }

void Tree::delNode (Node **root, int delKey)
{  Node  *pred;          //Vater des symetrischen Nachfolgers
   Node  *toDel;         //zu löschender symm. Nachf.

   if(*root == NULL);    // Bei leerem Baum: nichts zu tun
   else
   { if (delKey < (*root)->key)    // im li. Unterbaum löschen
       delNode (&((*root)->left), delKey);
     else
     {  if (delKey > (*root)->key )// im re. Unterbaum löschen
          delNode (&((*root)->right), delKey);
        else  // jetzt ist der Knoten gefunden
        { if ((*root)->left == NULL )   // kein li. Nachfolger
          {  toDel = *root;             // Knoten merken
             *root = (*root)->right;    // Knoten aushängen
          }
          else
          {  if ((*root)->right == NULL)// kein re. Nachfolger
             { toDel  = *root;          // Knoten merken
               *root = (*root)->left;   // Knoten aushängen
             }
             else  // jetzt gibt es zwei Nachfolger
             { pred = PredsymSucc(*root);// symm.Nachfolger
               if ( pred == *root )
               { // Sonderfall: symm. Nachf. = rechter Sohn
                 toDel = pred->right;
                 (*root)->key   = pred->right->key;
                 (*root)->right = (*root)->right->right;
               }
               else // Normalfall
               {
                 toDel = pred->left;
                 (*root)->key = pred->left->key;
                 pred->left   = pred->left->right;
               }
             }
          }
          cout << "\nKnoten " << delKey << " wird geloescht !";
          delete toDel;  // Speicherplatz des Knotens freigeben
        }
     }
  }
}
```

Die folgende Abbildung verdeutlicht die verschiedenen Situationen, die beim Entfernen eines Knotens zu beachten sind. Die Kennbuchstaben verweisen auf die im Programm-Listing angegebenen Situationen:

- In den Fällen **A** gibt es keine Nachfolgerknoten
- In den Fällen **B** gibt es zwei Nachfolgerknoten
- In den Fällen **C** gibt es nur linke Nachfolgerknoten
- In den Fällen **D** gibt es nur rechte Nachfolgerknoten

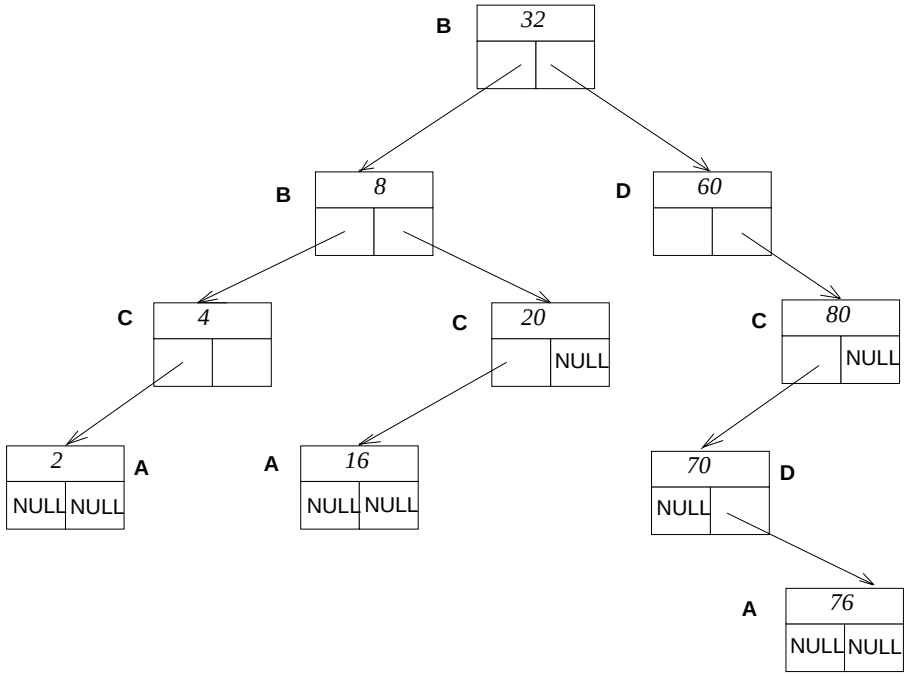



**Abbildung 6.6:** Verschiedene Fälle beim Entfernen eines Knotens

## 6.5 Durchlaufprinzipien für Bäume

Zahlreiche Baum-Anwendungen erfordern das Durchlaufen der Knoten eines Baumes, z.B.
- das Kopieren eines Baumes,
- das Ausdrucken oder Zeichnen eines Baumes,
- das Suchen eines Schlüssels im Baum,
- die Bestimmung charakteristischer Baumeigenschaften (Höhe, Pfadlänge, Vollständigkeit, ...)
- Die Ausgabe aller Schlüssel in einer bestimmten Reihenfolge ....

Die gemeinsame Basis all dieser Anwendungen sind Algorithmen zum Durchlaufen der Knoten in bestimmter Reihenfolge. Wir betrachten hier die drei wesentlichen Durchlaufprinzipien:

#### Inorder-Reihenfolge (LWR)

Diese auch als *symmetrische* oder *LWR-Reihenfolge* bekannte Methode durchläuft den Baum nach der folgenden Vorschrift:

- Durchlaufe zunächst den linken Teilbaum
- Besuche dann die Wurzel
- Durchlaufe zuletzt den rechten Teilbaum

#### Preorder-Reihenfolge (WLR)

Diese auch als *Haupt-* oder *WLR-Reihenfolge* bekannte Methode durchläuft den Baum nach der folgenden Vorschrift:

- Besuche zunächst die Wurzel
- Durchlaufe dann den linken Teilbaum
- Durchlaufe zuletzt den rechten Teilbaum

#### Postorder-Reihenfolge (LRW)

Diese auch als *Neben-* oder *LRW-Reihenfolge* bekannte Methode durchläuft den Baum nach der folgenden Vorschrift:

- Durchlaufe zuerst den linken Teilbaum
- Durchlaufe dann den rechten Teilbaum
- Besuche zuletzt die Wurzel

#### Beispiel

Wenn wir den Baum der Abbildung 6.1 nach den drei angegebenen Reihenfolge-Prinzipien durchlaufen und dabei den Schlüssel des jeweils aktuellen Knotens ausgeben erhalten wir drei verschiedene Schlüsselfolgen:

- Schlüsselfolge bei Inorder-Reihenfolge: 2, 4, 8, 16, 20, 32, 60
- Schlüsselfolge bei Preorder-Reihenfolge: 32, 8, 4, 2, 20, 16, 60
- Schlüsselfolge bei Postorder-Reihenfolge: 2, 4, 16, 20, 8, 60, 32

### C-Programme

Die folgenden C-Routinen stellen eine Implementierung der obigen drei Durchlauf-Methoden dar:

```
void InOrder (PNode Proot)
{
   if ( Proot != NULL )
   {
      InOrder (Proot->left);                 // gib den linken Unterbaum aus
      printf ("   %d  ", Proot->key);        // gib die Wurzel aus
      InOrder (Proot->right);                // gib den rechten Unterbaum aus
   }
}


void PreOrder (PNode Proot)
{
   if ( Proot != NULL )
   {
      printf ("   %d  ", Proot->key);        // gib die Wurzel aus
      PreOrder (Proot->left);                // gib den linken Unterbaum aus
      PreOrder (Proot->right);               // gib den rechten Unterbaum aus
   }
}


void PostOrder (PNode Proot)
{
   if ( Proot != NULL )
   {
      PostOrder (Proot->left);               // gib den linken Unterbaum aus
      PostOrder (Proot->right);              // gib den rechten Unterbaum aus
      printf ("   %d  ", Proot->key);        // gib die Wurzel aus
   }
}
```

### Alternative Reihenfolgen

Außer den angegebenen drei Durchlaufprinzipien lassen sich weitere daraus ableiten, wenn man statt der linken Teilbäume zuerst die rechten durchläuft. Man erhält dann außer den schon bekannten Reihenfolgen LWR, WLR und LRW noch die Reihenfolgen RLW, WRL und RLW. Die angegebenen rekursiven Verfahren lassen sich alternativ auch iterativ realisieren.

## 6.6 Fädelung

Die Durchlauf-Algorithmen im Abschnitt 6.5 erfordern einen Stack für die Verwaltung der wiederholt aufzusuchenden Knoten. Die rekursiven Varianten benutzen dazu die vom Laufzeitsystem implizit zur Verfügung gestellte Verwaltung der Prozeduraufrufe. Iterative Implementierungen verwalten selbst einen Stack mit Zeigern auf die Knoten, zu denen man noch einmal zurückkehren muß.

Die Fädelung löst dieses Verwaltungsproblem elegant dadurch, daß zusätzliche Zeiger in den Knoten gespeichert werden, so daß eine symmetrische Durchlaufreihenfolge ohne Verwaltungsaufwand ermöglicht wird. Die Zeiger werden an der Stelle der NULL-Zeiger untergebracht, die bei der natürlichen Implementierung von binären Bäumen die Blattknoten repräsentieren. Zur Unterscheidung zwischen 'normalen' Zeigern und Fädelungszeigern wird jedem Zeiger ein Flag zugeordnet. Die Fädelungszeiger werden wie folgt gesetzt:

– Falls ein Knoten ein Blatt als rechten Nachfolger hat, dann speichert man auf der Position des normalerweise notwendigen NULL-Zeigers einen Zeiger auf den symmetrischen Nachfolger des Knotens ab. Dieser steht weiter oben im Baum. Der letzte Knotenpunkt in Bezug auf den symmetrischen Durchlauf zeigt wieder auf die Wurzel. Diese Verkettung heißt *Vorwärtsverkettung für den symmetrischen Durchlauf.*
– Falls ein Knoten ein Blatt als linken Nachfolger hat, dann speichert man auf der Position des normalerweise notwendigen NULL-Zeigers einen Zeiger auf den symmetrischen Vorgänger des Knotens ab. Der erste Knotenpunkt in Bezug auf den symmetrischen Durchlauf zeigt wieder auf die Wurzel. Diese Verkettung heißt *Rückwärtsverkettung für den symmetrischen Durchlauf*.

**Beispiel**

Die folgende Abbildung zeigt einen mit Fädelungszeigern ausgestatteten Binärbaum aus den Abbildungen 6.1 bzw. 6.6.

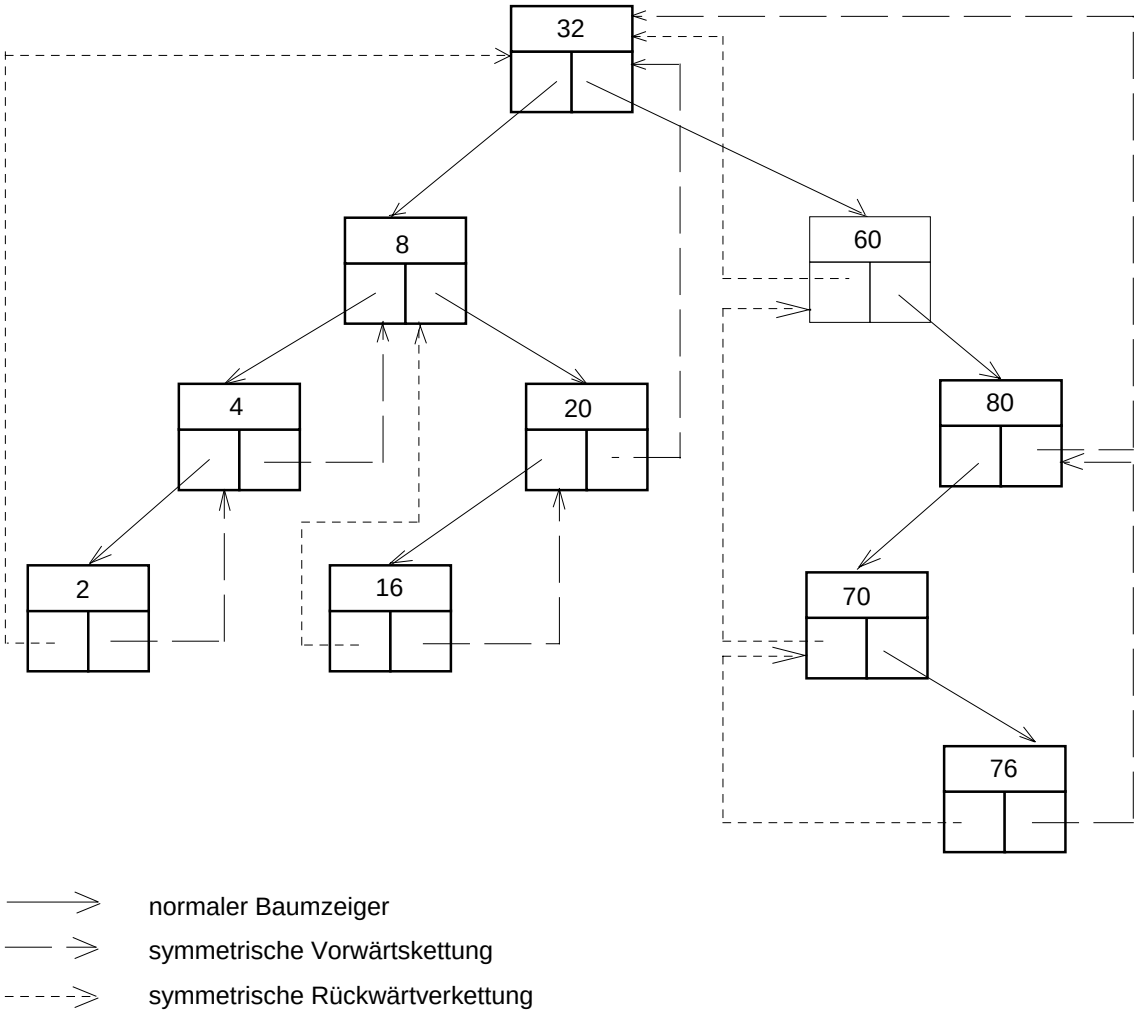



**Abbildung 6.7:** Natürlicher binärer Baum mit Fädelungszeigern

### Record-Struktur für gefädelte Binärbäume

Da bei der Fädelung die NULL-Zeiger, welche die Blattknoten repräsentieren, durch die Fädelungszeiger ersetzt werden, muß es für jeden Zeiger ein Kennzeichen geben, aufgrund dessen man seinen Typ erkennen kann. In der Praxis genügt dazu ein Bit, mit dem zwischen "normalen" und Fädelungszeigern unterschieden wird, im folgenden wird aber ein eigenes Feld im Knotenrecord benutzt.

Die C-Klassendefinition für die Knoten gefädelter Bäume kann wie folgt aussehen:

```
enum ptrType { treeLink,      // normaler Baumzeiger
               symmPred,      // Zeiger zum symmetrischen Vorgänger
               symmSucc       // Zeiger zum symmetrischen Nachfolger
             };

class Fnode
{
  public:
   Fnode ( int Key, char *Info="", Fnode *Left=NULL, Fnode*Right=NULL,
           ptrType lType=symmPred, ptrType rType=symmSucc)
         {  strncpy (info, Info, INFLEN); info[INFLEN]=0;
            key       = Key;
            left      = Left;
            right     = Right;
            leftType  = lType;
            rightType = rType;
         }
   virtual ~Fnode () {}

   void    getKey   (int &Key)   { Key = key;                    }
   void    setKey   (int Key)    { key = Key;                    }
   void    getInfo  (char* Info) { strncpy (Info,info,INFLEN);
                                   Info[INFLEN]=0;               }
   void    setInfo  (char* Info) { strncpy (info,Info,INFLEN);
                                   info[INFLEN]=0;               }
   int      key;            // Knoten-Schlüssel
   char     info[INFLEN];   // Knoten-Information
   Fnode    *left;          // Zeiger zum linken  Nachfolger
   Fnode    *right;         // Zeiger zum rechten Nachfolger
   ptrType  leftType;       // Typ des Zeigers zum linken  Nachfolger
   ptrType  rightType;      // Typ des Zeigers zum rechten  Nachfolger
};
```

### Symmetrischer Durchlauf durch einen gefädelten Baum

Um einen mit Fädelungszeigern ergänzten binären Suchbaum in symmetrischer Reihenfolge zu durchlaufen kann man nun wie folgt verfahren:

- Suche als Startpunkt den am weitesten links im Baum stehenden Knoten auf.
- Solange bis wir von dem am weitesten rechts im Baum stehenden Knoten wieder zurück zur Wurzel kommen, gehen wir vom aktuellen Knoten zu seinem symmetrischen Nachfolger.

Dabei sind zwei Fälle zu unterscheiden:

- Falls der Rechts-Zeiger ein Fädelungszeiger ist , d.h. vom Typ symSucc, verweist er uns direkt auf den symmetrischen Nachfolger.
- Falls der Rechts-Zeiger ein "normaler" Baumzeiger ist, d.h. vom Typ TreeLink, ist der symmetrische Nachfolger der am weitesten links stehenden Knoten im rechten Teilbaum.

Wie man in Abbildung 6.7 feststellt, erhält man beim symmetrischen Durchlaufen des Suchbaumes die Schlüssel in aufsteigender Reihenfolge.

### C++ - Methode für den symmetrischen Durchlauf durch einen gefädelten Baum

```
void Ftree::inOrder() {  listInOrder(root); }

void Ftree::listInOrder(Fnode *node)
{
  int  rCount = 0;            // zählt, wie oft wir die Wurzel treffen
  Fnode *Root = node;         // für Abbruchbedingung Wurzel merken

  if ( node == NULL )return;  // leeren Baum abfangen
  while ( rCount < 2 )
  { // gehe zum am weitesten links stehenden Knoten
    while ( node->leftType == treeLink )
       node = node->left;

    // solange es einen symmetrische Nachfolger gibt, bearbeite diesen
    for (;;)
    {
      if ( node == Root )   // Abbruch, wenn zum 2. Mal bei der Wurzel
      if ( ++rCount >= 2 )  return;
         cout << node->key << "  " ;     // aktuellen Knoten ausgeben
      if ( node->rightType != symmSucc ) break;
         node = node->right;             // weiter zum symm. Nachf.
    }
    // der aktuelle Knoten hat keinen direkten symm. Nachfolger,
    // dann hat er einen echten rechten Sohn:  gehe zum rechten
    // Unterbaum und beginne von vorn.
    node = node->right;
  }
}
```

### Aufbau eines gefädelten Binärbaumes

Beim Aufbau eines gefädelten Binärbaumes ist die wesentliche Operation das Einfügen eines neuen Knotens in den bereits bestehenden Baum. Dabei ist wie folgt zu verfahren:

- zunächst ist die Einfügeposition für den neuen Knoten aufzusuchen. Dabei gehen wir genau wie bei normalen Binärbäumen vor. Die Einfügeposition wird im Gegensatz zu normalen Suchbäumen nicht an den NULL-Zeigern, sondern aufgrund der Fädelungszeiger erkannt, die an ihrer Stelle stehen. Die Prozedur `FInsertNodeI` führt diesen Suchprozeß iterativ durch.
- Wenn die Einfügeposition erreicht ist, muß der Knoten eingehängt werden. Dabei müssen außer dem Einhängen in den Baum die Fädelungszeiger korrigiert werden. Dies wird mit den Prozeduren `InsertLeft` und `InsertRight` erreicht.

### C++ - Methode für das Einfügen von Knoten in einen gefädelten Baum

```
void Ftree::insertNode (int newKey, char *newInfo)
{  Fnode *thisNode;             // Zeiger zum aktuellen Knoten
   Fnode *newNode;              // Zeiger zu seinem Vorgaenger
   if (root == NULL)
   {  root = new Fnode(newKey, newInfo, root, root, symmPred, symmSucc);
      root->left  = root;
      root->right = root;
   }
```

```
   else                                  // Falls Baum nicht leer:
   {  thisNode = root;                   // auf die Wurzel zeigen
      do                                 // Einfuegeposition suchen:
      {  if (newKey == thisNode->key)    // nur neue Schluessel erlauben
         { cout << "Schluessel  " << thisNode->key
                << "  ist bereits vorhanden\n";
           return;
         }
         if (newKey < thisNode->key)
           if (thisNode->leftType == treeLink)
               thisNode = thisNode->left; // kleine Schluessel links
            else
            {  newNode = new Fnode( newKey, newInfo, thisNode->left,
                                    thisNode, symmPred, symmSucc );
               thisNode->left     = newNode;
               thisNode->leftType = treeLink;
               return;
            }
         else
           if (thisNode-> rightType == treeLink)
             thisNode = thisNode->right;   // grosse Schluessel rechts
           else
           {  newNode = new Fnode( newKey, newInfo, thisNode,
                                   thisNode->right, symmPred, symmSucc);
              thisNode->right     = newNode;
              thisNode->rightType = treeLink;
              return;
           }
      } while (true);
   }
}
```

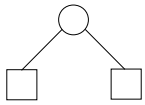
## 6.7 Analytische Betrachtungen für Bäume

In diesem Abschnitt untersuchen wir Eigenschaften von Binärbäumen mit dem Ziel, Aussagen über den Aufwand für die typischen Baumoperationen, also das Suchen, Einfügen und Ersetzen von Knoten zu finden.

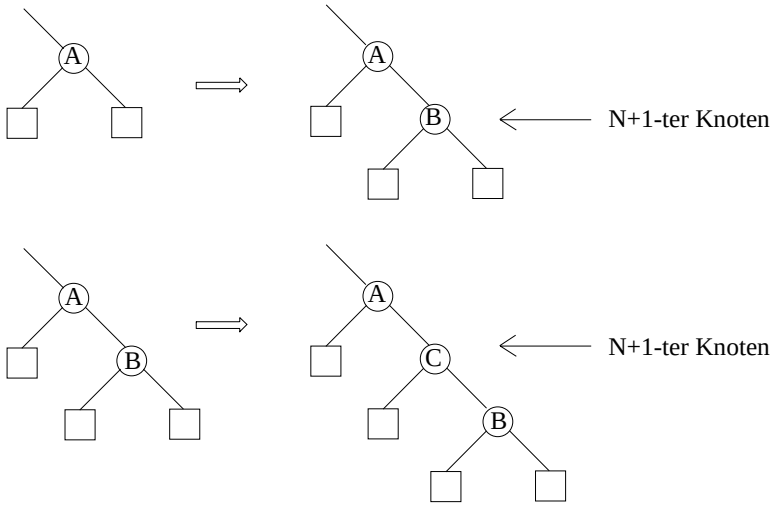
**Eigenschaft 1:** Ein Binärbaum mit $N$ inneren Knoten besitzt $b(N) = N + 1$ Blätter.

Dies begründen wir durch einen Induktionsschluß:

– Für $N = 1$ hat der Baum die unten abgebildete Gestalt. Es ist also $b(1) = 2$.



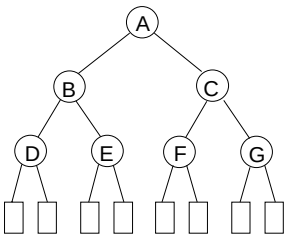
– Wenn für alle Bäume mit $N$ Knoten angenommen wird, daß die Anzahl Ihrer Blätter $b(N) = N + 1$ ist, dann erhält man daraus Bäume mit $N + 1$ Knoten durch Einfügen eines weiteren Knotens, entweder im Innern oder anstelle eines Blatts. In beiden Fällen erhöht sich die Zahl der Blätter um 1, wie die folgende Abbildung zeigt. Es gilt also auch für einen Baum mit $N + 1$ Knoten $b(N + 1 = (N + 1) + 1$.




**Abbidung 6.8:** Zur Anzahl innerer Knoten eines Baumes

**Eigenschaft 2:** Die Höhe eines Binärbaumes mit $N$ inneren Knoten liegt zwischen $N$ und $\lceil \log_2(N+1) \rceil$.

**Beispiel:**




$N = 7$

$h = \lceil {}_2\log(7+1) \rceil = 3$

Eigenschaft 2 läßt sich folgendermaßen begründen:

- Die maximale Höhe $N$ wird bei Entartung des Baumes zu einer linearen Liste erreicht.
- Für einen vollständigen Binärbaum besteht zwischen seiner Höhe $h$ und der Anzahl $N$ seiner *inneren* Knoten die Beziehung

$$N = 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1}$$

$$N = \sum_{i=0}^{h-1} 2^i = \frac{2^h - 1}{2 - 1} = 2^h - 1$$

  also $$h = \log_2(N+1)$$

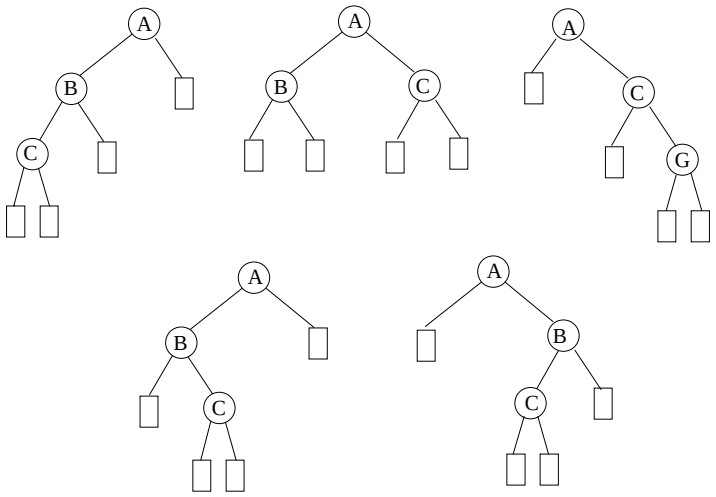
- Falls sich mit $N$ Knoten kein vollständiger Baum aufbauen läßt, wird das Minimum der Höhe erreicht für einen "fast" vollständigen Baum, der nur auf der untersten Ebene nicht voll besetzt ist. Die Höhe ist in diesem Fall die nächst größere ganze Zahl nach $\log_2(N+1)$ also $\lceil \log_2(N+1) \rceil$

Da der Aufwand typischer Operationen in binären Bäumen direkt mit den Pfadlängen von der Wurzel zu dem bearbeiteten Knoten zusammenhängt, liegt der Aufwand aufgrund der Eigenschaft 2 häufig zwischen der maximalen Baumhöhe $N$ und der minimalen Baumhöhe $\lceil \log_2(N+1) \rceil$

**Überlegungen zum mittleren Aufwand für Operationen**

Ein weiteres wichtiges Kriterium neben den oberen und unteren Schranken für die Komplexität ist der *mittlere Aufwand* für typische Operationen in binären Bäumen. Dazu ist festzulegen, über welche Klasse von Bäumen gemittelt wird. Im wesentlichen sind die folgenden alternativen Ansätze denkbar:

- der mittlere Aufwand für alle Bäume, die sich aus $N$ zufällig gewählten Schlüsseln aufbauen lassen durch sukzessives Einfügen (Random Tree-Analyse).
- der mittlere Aufwand für alle Baumtypen, die strukturell verschieden sind (Gestaltsanalyse). Für den Fall $N = 3$ erhält man z.B. fünf strukturell verschiedene Baumtypen:




- der mittlere Aufwand für alle vollständigen Binärbäume mit $N$ Knoten. Dieser letzte Fall ist besonders einfach zu analysieren. Wir werden dies im folgenden genauer betrachten:

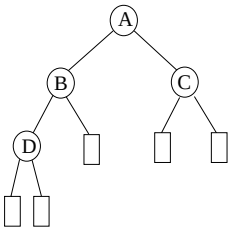
### Die interne Pfadsumme I

Als ein Maß für den gesamten Zugriffsaufwand auf alle Knoten eines Binärbaumes mit $N$ inneren Knoten definieren wir die *interne Pfadsumme* wie folgt:

- für den leeren Baum setzen wir $\boldsymbol{I} = 0$
- für einen Baum mit der Wurzel als einzigem inneren Knoten setzen wir $\boldsymbol{I} = 1$
- für einem Baum $\boldsymbol{B}$ mit einer Wurzel $p$, einem linken Teilbaum $\boldsymbol{B_l}$ und einem rechten Teilbaum $\boldsymbol{B_r}$ setzen wir als innere Pfadsumme fest:

$$\boldsymbol{I}(\boldsymbol{B}) = \boldsymbol{I}(\boldsymbol{B_l}) + \boldsymbol{I}(\boldsymbol{B_r}) + \text{Anzahl innerer Knoten von } \boldsymbol{B}$$

Von der Wurzel *w* aus gesehen haben alle inneren Knoten einen um 1 größeren Abstand zur Wurzel *w* als zur Wurzel des linken oder rechten Unterbaumes, in dem sie hängen. Die interne Pfadsumme gibt also an, wieviele Zugriffe notwendig sind, wenn wir alle inneren Knoten besuchen wollen:




Anzahl der inneren Knoten: 4

interne Pfadsumme: **I(B)** = 4 + 1 + 2 + 1 =

Allgemein gilt für den Zusammenhang zwischen der internen Pfadsumme $\boldsymbol{I}$ und der Höhe $h$ eines vollständigen Binärbaumes:

$$\boldsymbol{I}(h) = \sum_{\substack{\text{innere} \\ \text{Knoten } p}} \left[\text{Tiefe}(p) + 1\right]$$

### Die interne Pfadsumme vollständiger Binärbäume

Ein Baum der Höhe $h$ hat auf den Levels $0, 1, \cdots, h-1$ $2^h$ innere Knoten. Für diese Level gibt die folgende Übersicht die Knotenanzahl und Pfade an:

| Level | Anzahl innerer Knoten | Zugriffe zum Findender Knoten |
|---|---|---|
| 0 | 1 | 1=0+1 |
| 1 | $2 = 2^1$ | 2=1+1 |
| 2 | $4 = 2^2$ | 3=2+1 |
| …… | …… | …… |
| h-1 | $2^{h-1}$ | h=(h-1)+1 |

Damit ist die interne Pfadsumme $\boldsymbol{I}(h)$ eines vollständigen Baumes der Höhe $h$:

$$\begin{aligned}\boldsymbol{I}(h) &= \sum_{i=0}^{h-1} (i+1) * 2^i \\ &= (h-1) * 2^h + 1\end{aligned}$$

Der letzte Schritt läßt sich folgendermaßen durch einen Induktionsschluß begründen:

Für $h=1$ ist : $\boldsymbol{I}(1) = (1-1)*2^1 + 1 = 1$

Wenn wir die Gültigkeit von $\boldsymbol{I}(h) = (h-1)*2^h + 1$ für h voraussetzen, dann gilt für $h+1$ :

$$= \sum_{i=0}^{h} (i+1)*2^i == \sum_{i=0}^{h-1} (i+1)*2^i + (h+1)*2^h$$
$$= (h-1)*2^h + 1 + (h+1)*2^h$$
$$= (2h)*2^h - 2^h + 1 + 2^h$$
$$= h*2^{h+1} + 1 = \boldsymbol{I}(h+1)$$

### Die mittlere interne Pfadsumme $\boldsymbol{I_m}$

Die mittlere interne Pfadsumme eines vollständigen Binärbaumnes können wir nun als ein Maß für den *durchschnittlichen* Aufwand betrachten, der für eine Operation im Baum notwendig ist. Für einen Baum mit $N$ inneren Knoten gilt:

$$\boldsymbol{I}_m(h) = \frac{\boldsymbol{I}(h)}{\boldsymbol{N}} = \frac{(h-1)*2^h + 1}{N}$$

Wegen $h = \log_2(N+1)$ bzw. $N = 2^h - 1$ erhalten wir durch Elimination von $N$:

$$\boldsymbol{I}_m(h) = \frac{(h-1)*2^h + 1}{2^h - 1}$$
$$= \frac{(2^h - 1)(h-1) + 1 + (h-1)}{2^h - 1}$$
$$= (h-1) + \frac{h}{2^h - 1}$$

Analog erhalten wird bei Elimination von $h$ den mittleren Aufwand als eine Funktion von *N*.

$$\boldsymbol{I}_m(N) = \log_2(N+1) - 1 + \frac{\log_2(N+1)}{N}$$

Als Ergebnis können wir somit festhalten:

> Der mittlere Aufwand für die Operationen Suchen, Ersetzen, Einfügen in vollständigen binären Bäumen mit $N$ inneren Knoten beträgt:
>
> $$\boldsymbol{I}_m(N) = \left(1 + \frac{1}{N}\right) * \log_2(N+1) - 1 = \mathbf{O}(\log(N))$$

Für die obigen alternativen Ansätze kann man ebenfalls einen mittleren Aufwand ableiten:

– bei der Random Tree-Analyse erhält man den mittleren Aufwand

$$\boldsymbol{I}_m(N) \approx 1.386 * \log_2(N) - 1.845 + \frac{2 * \ln(N)}{N} + \cdots\cdots = \mathbf{O}(\log(N))$$

– bei der Gesaltsanalyse ergibt sich ein mittlerer Aufwand: $\boldsymbol{I}_m(N) = \sqrt{\pi\, N} + const$

# 6.8 Balancierte Binärbäume

Binärbäume können durch häufiges Einfügen und Löschen von Schlüsseln zu linearen Listen entarten. Der Aufwand für Zugriffe steigt dann von $\boldsymbol{O}(\log(N+1))$ auf $\boldsymbol{O}(N)$ an. Durch Balancierung versucht man eine Entartung der Baumstruktur zu verhindern und damit den Aufwand für Baumoperationen in der Nähe des Optimums zu halten.

### Balancierungskriterien

Die Ausgeglichenheit eines Binärbaumes kann in unterschiedlicher Weise präzisiert werden:

- Vollständigkeit stellt das strengste Kriterium für die Ausgeglichenheit dar: Es wird verlangt, daß alle Ebenen des Baumes voll besetzt sind. Da dies nur für bestimmte Anzahlen von Knoten zu erreichen ist, nämlich für $N = 2^h - 1$ , läßt man auf der untersten Baumebene Ausnahmen zu.
- Bei höhenbalancierten Bäumen fordert man, daß bei jedem inneren Knoten die Höhen seiner beiden Unterbäume sich nicht *wesentlich* unterscheiden.
- Bei gewichtsbalancierten Bäumen darf für jeden inneren Knoten die Anzahl der Knoten in seinem linken und in seinem rechten Unterbaum nicht *wesentlich* verschieden sein.

### Balancierungsaufwand

In Abschnitt 7.7 haben wir als Maß für den Zugriffsaufwand die mittlere interne Pfadlänge betrachtet. Für zufällig aufgebaute Baumstrukturen war diese ungefähr $1{,}386 * \log_2(N)$. Dies ist vom erreichbaren Optimum, nämlich $\lceil(\log_2(N+1))\rceil$, nicht allzu weit entfernt. Eine exakte Balancierung lohnt sich daher nur, wenn der zusätzliche Aufwand dafür gering ist oder wenn Such-Operationen wesentlich häufiger als Einfüge- oder Lösch-Operationen vorkommen.

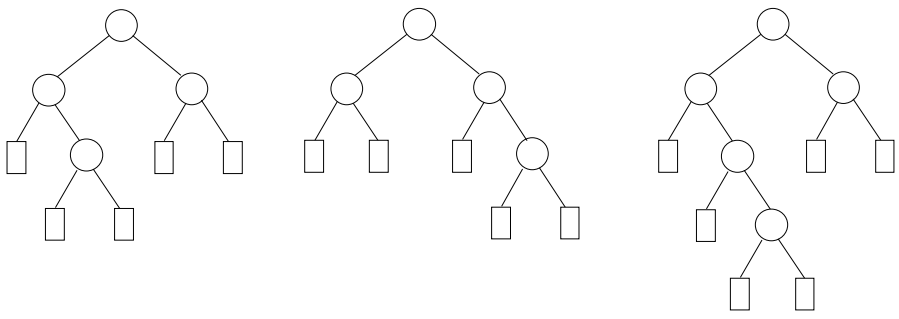
Um den Balancierungsaufwand zu minimieren, benutzt man abgeschwächte Balancierungskriterien, die zwar keine vollständig ausgeglichenen Bäume ergeben, dafür aber effizient erreichbar sind und dem Optimum trotzdem sehr nahe kommen.

### AVL-Bäume

Von den beiden russischen Mathematikern Adelson-Velski und Landis wurde 1962 ein Typ von höhenbalancierten Bäumen untersucht, der nach ihnen als AVL-Baum benannt ist. AVL-Bäume sind einfach implementierbar, ermöglichen effiziente Ausgleichsprozeduren und führen zu Pfadlängen, die dem Optimum sehr nahe kommen.

> Ein Binärbaum heißt *AVL-Baum*, wenn für jeden seiner inneren Knoten $p$ gilt:
> Die Höhe des linken und des rechten Unterbaumes von $p$ differieren höchstens um 1.

Die ersten beiden unten dargestellten Bäume genügen dem AVL-Kriterium, der dritte jedoch nicht, da die Bedingung in der Wurzel verletzt ist:



### Balancegrad

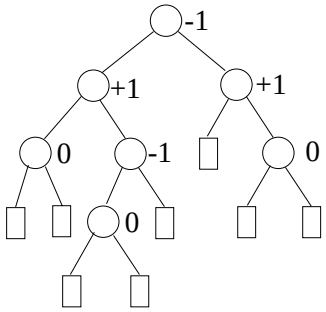
Für die Implementierung der Ausgleichsalgorithmen genügt es, in jedem Knoten $p$ die Höhendifferenz zwischen dem linken Unterbaum $\boldsymbol{B}_l$ und dem rechten Unterbaum $\boldsymbol{B}_r$ zu kennen. Dieser Balancegrad kann in einem AVL-Baum nur die folgenden Werte annehmen:

$bal(p) = +1$ falls der rechte Teilbaum $\boldsymbol{B}_r$ eine um 1 größere Höhe hat als der linke.

$bal(p) = -1$ falls der rechte Teilbaum $\boldsymbol{B}_r$ eine um 1 kleinere Höhe hat als der linke.

$bal(p) = 0$ falls die Teilbäume $\boldsymbol{B}_l$ und $\boldsymbol{B}_r$ die gleiche Höhe haben. In diesem Fall heißt $p$ ein balancierter Knoten.

Das folgende Beispiel zeigt einen AVL-Baum mit den Balancegraden der einzelnen Knoten:




### Implementierung von AVL-Bäumen

Bei der Realisierung von AVL-Bäumen als Listen wird gegenüber dem bisherigen Vorgehen in den Knoten eine zusätzliche Angabe über den Balancierungsgrad erforderlich. In der Praxis genügen dafür 2 Bit. Für die C-Beispiele verwenden wir die Klassen AVLnode und AVLtree:

```
class AVLnode
{
  public:
    AVLnode (int Key, int Bal=0, char *Info="")
            {
              strncpy (info, Info, INFLEN); info[INFLEN]=0;
              key       = Key;
              balance   = Bal;
              left      = NULL;
              right     = NULL;
             }
     virtual ~AVLnode () {}

     void  getKey  (int &Key)      { Key = key;                }
     void  setKey  (int Key)       { key = Key;                }
     void  getInfo (char* Info)    { strncpy (Info,info,INFLEN);
                                     Info[INFLEN]=0;           }
     void  setInfo (char* Info)    { strncpy (info,Info,INFLEN);
                                     info[INFLEN]=0; }
     int     key;                  // Knoten-Schlüssel
     int     balance;              // Balancegrad des Konotens
     char    info[INFLEN+1];       // Knoten-Information
     AVLnode *left;                // Zeiger zum linken  Nachfolger
     AVLnode *right;               // Zeiger zum rechten Nachfolger
};
```

```
class AVLtree
{
  public:
    AVLtree             () { root=NULL; delPtr=NULL; }
    virtual ~AVLtree    () { destroyTree(root);      }
    bool  insertNode    (int newKey, char *newInfo);
    bool  insertNode1   (AVLnode **root, int newKey, char *newInfo);
    void   deleteNode   (int delKey);
    void   AVLprintTree ();
    void   listTree     ();
    int    treeHeight   ();

     AVLnode    *root;

  private:
    virtual void destroyTree (AVLnode *rootPtr);
    bool  balanced           (AVLnode *root);
    int   hoehe              (AVLnode *root);
    void  AVLprintSubTree    (AVLnode *subRoot, int width);
    void  listSubTree        (AVLnode *subRoot);
    bool  delNode1           (AVLnode **root, int delKey);
    void  balanceLeft        (AVLnode **p, bool *bal_changed);
    void  balanceRight       (AVLnode **p, bool *bal_changed);
    void  keyTransfer        (AVLnode **root, bool * bal_changed);

    AVLnode  *delPtr;         // Zeiger auf den zu löschenden Knoten
};
```

### Einfüge-Operationen in AVL-Bäumen

Das Einfügen neuer Knoten in einen AVL-Baum wird zunächst genauso durchgeführt wie bei gewöhnlichen binären Suchbäumen. Weil dabei aber das AVL-Kriterium verletzt werden kann, müssen zusätzliche Maßnahmen ergriffen werden, die den Balancegrad von Knoten korrigieren und bei Bedarf die Baumstruktur modifizieren.

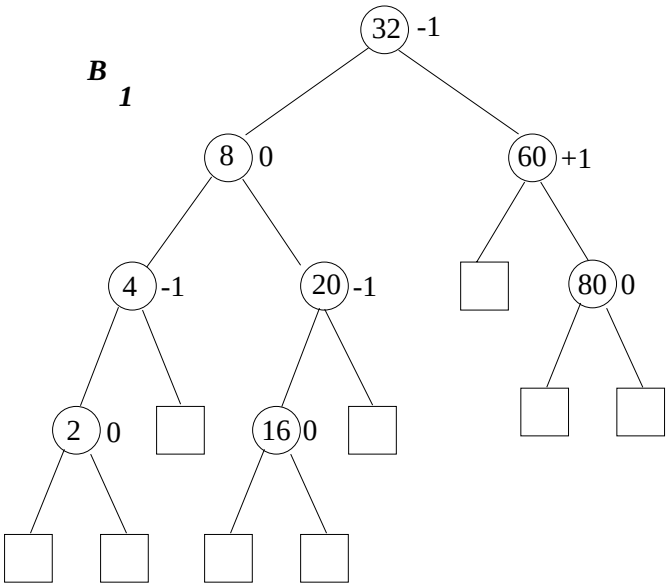
Prinzipiell sind die folgenden Schritte dabei erforderlich:

1. Suche die Einfügeposition im Baum und hänge den neuen Knoten ein.
2. Korrigiere, soweit erforderlich den Balancegrad von Knoten.
3. Modifiziere, soweit erforderlich die Baumstruktur.

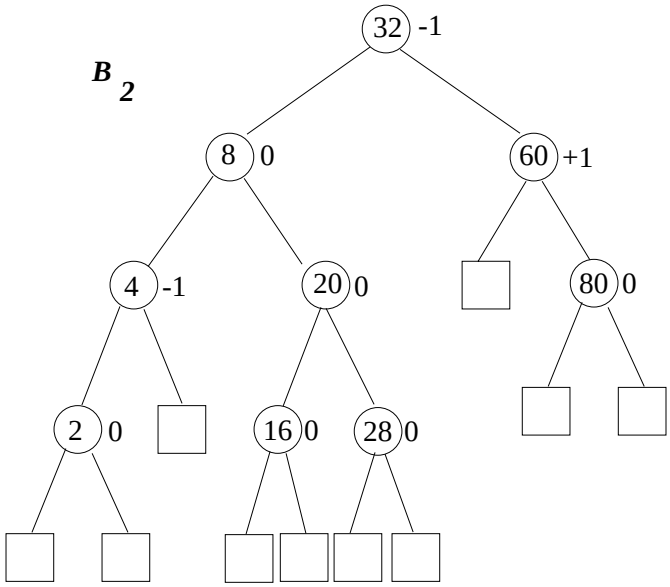
Die Schritte 2. und 3. müssen nur entlang des Suchpfades erfolgen, der bei Schritt 1 benutzt wurde. Beim 3. Schritt werden Rotations- und Doppel-Rotations-Operationen verwendet, die wir zunächst an Beispielen betrachten:

**Beispiel 1:**

Wir gehen von einem AVL-Baum $B_1$ aus von der folgenden Gestalt:



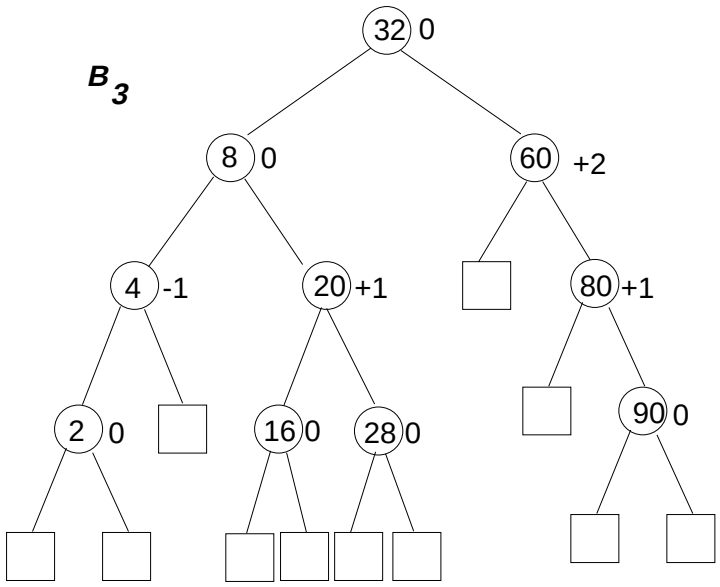

Wir fügen zunächst den Schlüssel 28 neu in den Baum $B_1$ ein und erhalten den Baum $B_2$




Bei dieser Operation hat der Baum seine AVL-Eigenschaft nicht verloren, weil wir den Unterbaum des Knotens 20 verlängert haben, der die geringere Höhe hat. Es mußte daher nur der Balancegrad im Knoten mit dem Schlüssel 20 modifiziert werden. Weiter oben im Baum stehende Knoten sind von der Operation nicht betroffen, da der Teilbaum mit der Wurzel 20 seine Gesamthöhe nicht verändert hat.

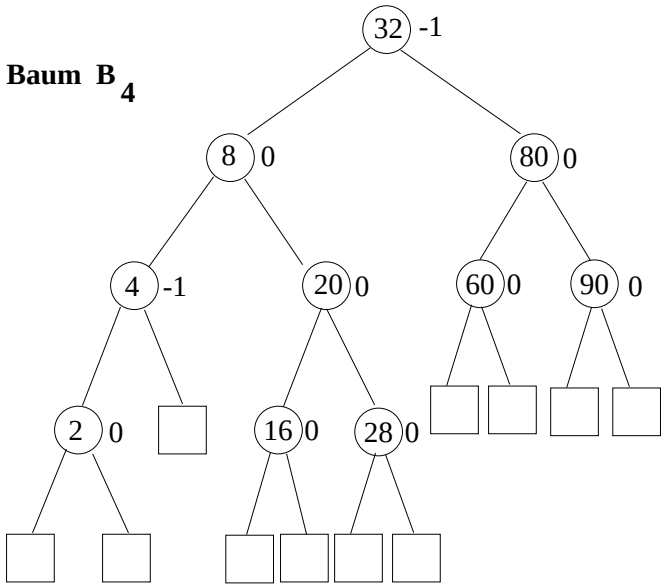
**Beispiel 2:**

Wenn wir im Baum $B_2$ einen weiteren Knoten mit dem Schlüssel 90 einfügen, dann geht zunächst die AVL-Struktur verloren und wir erhalten den Baum $B_3$:



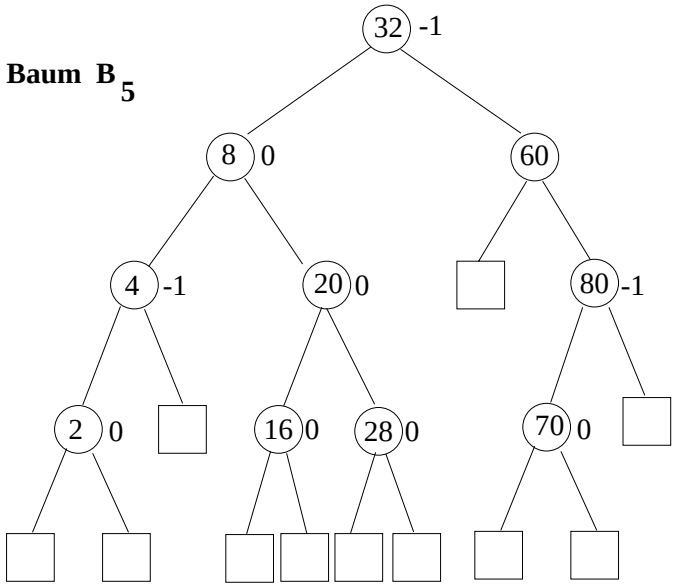

Der Knoten 80 erhält nach dem Einfügen den Balancegrad +1, der Teilbaum mit der Wurzel 80 hat aber weiterhin die AVL-Eigenschaft.

Der rechte Unterbaum des Knotens 60 hat aber nun eine um 2 größere Höhe als der linke. Diesen Mangel beseitigen wir durch eine *Rotation* und erhalten damit den Baum $B_4$

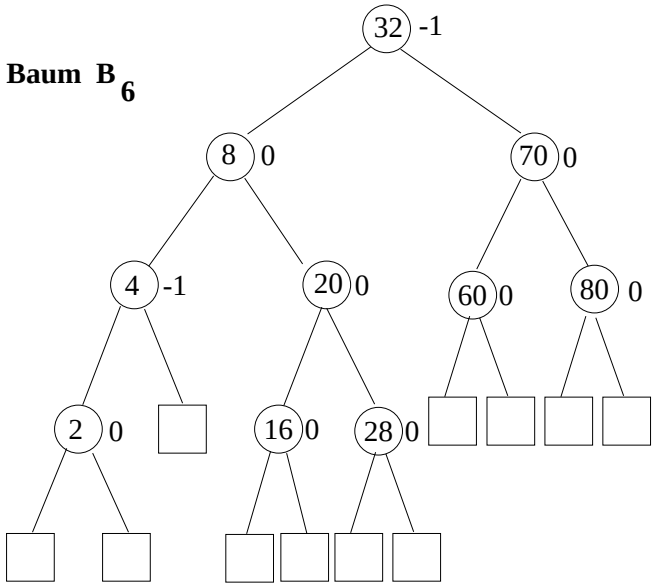


**Beispiel 3:**

Wenn wir im Gegensatz zum Beispiel 2 im Baum $B_2$ einen weiteren Knoten mit dem Schlüssel 70 einfügen, dann geht ebenfalls die AVL-Struktur verloren und wir erhalten den Baum $B_5$ :




Der rechte Unterbaum des Knotens 60 hat nun wie im Beispiel 2 eine um 2 größere Höhe als der linke. Im Unterschied zu Beispiel 2 ist die Verlängerung aber nach links erfolgt. In dieser Situation beheben wir den Mangel wir durch eine *Doppel-Rotation*:



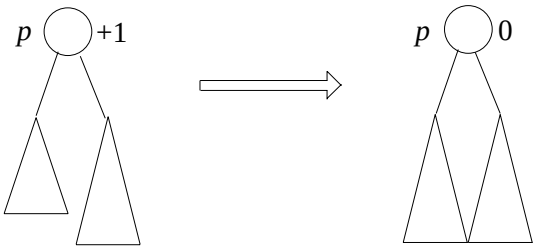

Eine einfache Rotation wie im Beispiel 2 hätte die Unausgeglichenheit nicht gelöst !

### Einfügen in AVL-Bäume

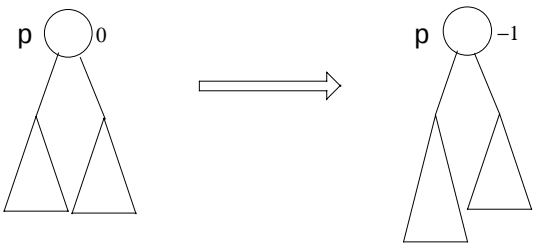
Das Einfügen eines neuen Schlüssels beginnt zunächst genau wie bei normalen Binärbäumen. Anschließend muß der Suchpfad jedoch zurückverfolgt werden. In allen dabei angetroffenen Knoten ist aufs neue der Balancegrad zu überprüfen. Im einfachsten Fall genügt es, ihn zu aktualisieren. In anderen Fällen muß auch die Baumstruktur modifiziert werden. Sobald man einen Knoten antrifft, dessen Balancegrad sich nicht mehr ändert, kann das Verfahren abgebrochen werden.

Bei der Überprüfung des Balancegrades eines Knotens $p$ unterscheiden wir danach, ob der neue Knoten im linken oder im rechten Unterbaum von $p$ eingefügt worden ist. Da beide Fälle ganz symmetrisch behandelt werden beschränken wir uns auf den Fall, daß er *links eingefügt* wurde. Dabei können drei verschiedene Fälle auftreten:

**Fall 1:** Der linke Unterbaum von $p$ war vor dem Einfügen kürzer als der rechte. Der alte Balancegrad war daher +1. Er muß auf 0 umgestellt werden, da jetzt der linke Unterbaum von $p$ genauso hoch ist wie der rechte. Die Höhe des Teilbaumes mit der Wurzel $p$ hat sich aber beim Einfügen nicht verändert. Daher wird der Balancegrad des Vaterknotens von $p$ sich nicht ändern und wir können das Verfahren beenden.

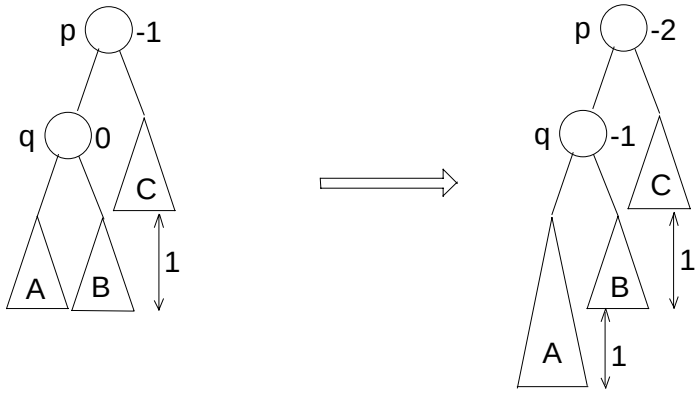



**Fall 2:** Beide Unterbäume von $p$ ware
n vor dem Einfügen gleich hoch.
Der alte Balancegrad war daher 0. Er muß auf -1 umgestellt werden, da jetzt der linke Unterbaum von $p$ um eine Ebene höher ist als der rechte. Danach müssen wir den über $p$ liegenden Knoten untersuchen, weil sich die Gesamthöhe des Teilbaumes unter $p$ verändert hat.



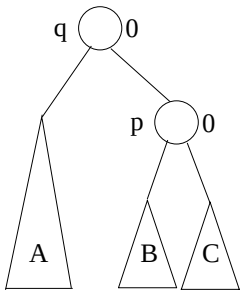

**Fall 3:** Der linke Unterbaum von $p$ war vor dem Einfügen schon um 1 Ebene höher als der rechte. Es gab also schon vor dem Einfügen einen linken Sohn $q$ von $p$ und die beiden Unterbäume von $q$ waren gleich hoch (andernfalls hätte schon früher eine Balancierung stattfinden müssen). Wir unterscheiden nun danach, ob der neue Knoten im linken oder im rechten Unterbaum von $q$ eingefügt wurde.

**Fall 3.1:** Der linke Unterbaum von *p* wurde nach links verlängert:

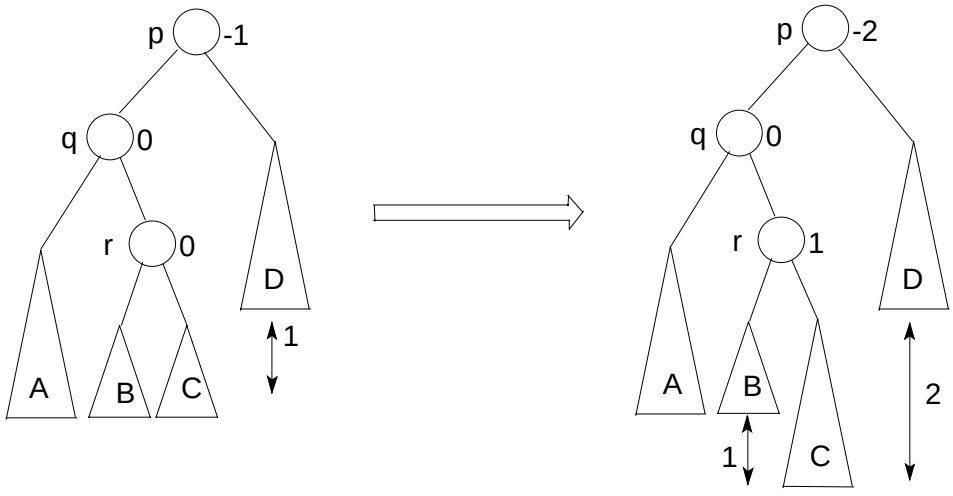



In dieser Situation modifizieren wir die Baumstruktur durch eine Rotation:
Der Knoten *q* wird zur neuen Wurzel, der Knoten *p* sinkt eine Stufe nach unten in in seinen rechten Unterbaum. Der Knoten *q* behält seinen linken Unterbaum. Sein rechter Unterbaum wird aber linker Unterbaum von *p*. Dabei geht die Suchbaum-Eigenschaft nicht verloren, die Gesamthöhe des Teilbaumes wird wieder auf den alten Wert reduziert und der Balancegrad der beiden Knoten *p*, und *q* wird 0.

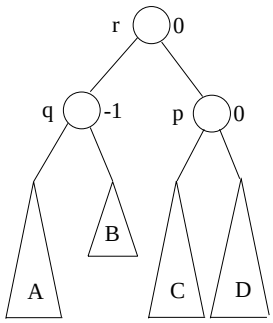
Damit ist die Balancierung beendet. Das Ergebnis dieser Operation ist das folgende:




**Fall 3.2:** Der linke Unterbaum von *p* wurde nach rechts verlängert. Die Wurzel des rechten Unterbaums sei *r* . Wir betrachten hier nur den Fall, daß der rechte Unterbaum von *r* verlängert wird. Dabei geht die AVL-Bedingung aber verloren:



In dieser Situation modifizieren wir die Baumstruktur durch eine Doppelrotation: Der Knoten *r* wird zur neuen Wurzel, der Knoten *p* sinkt eine Stufe nach unten in seinen rechten Unterbaum. Der Knoten *q* behält seinen Platz. Der linke Unterbaum von *r* wird nun neuer rechter Unterbaum von *q* und der rechte Unterbaum von *r* wird linker Unterbaum von *p*. Bei dieser Umordnung bleibt einerseits die Suchbaum struktur erhalten, ande rerseits wird die Höhenbilanz wieder ausgegleichen. Das Ergeb-nis zeigt die folgende Abbildung:




Damit haben wir den Ausgleich für den Fall, daß der neue Knoten in den linken Unterbaum von *p* eingefügt wurde, vollständig untersucht. Das Einfügen in den rechten Unterbaum wird analog behandelt.

### C++ - Methode für das Einfügen in einen AVL-Baum

```
bool AVLtree::insertNode(int newKey, char *newInfo )
{ return(insertNode1 (&root, newKey, newInfo)); }

bool AVLtree::insertNode1 ( AVLnode **root, int newKey,
                                            char *newInfo )
{
  AVLnode  *WurzelVerlaengTeilbaum; // für Ausgleichsmaßnahmen
  AVLnode  *NeueWurzel;
  bool     bal_changed;

  bal_changed = false;
  if (!(*root))                     // neuen Baum erzeugen
  {  *root = new AVLnode (newKey, 0, newInfo);
     bal_changed = true;
  }
  else
  { // neuen Knoten im linken Teilbaum einhängen
    if ( newKey < (*root)->key )
    { bal_changed = insertNode1(&(*root)->left,newKey,newInfo);
      // Einfügen ist damit erledigt; nun wird nun balanciert
      if ( bal_changed )
      { // je nach Balancierungsgrad des aktuellen Knotens:
        switch ( (*root)->balance )
        { case  0: // Teilbäume unter aktuellem Knoten waren
                   // bisher ausgeglichen, da im linken Teil-
                   // baum eingefügt wurde, ist jetzt der
                   // rechte Teilbaum niedriger.
                   // bal_changed bleibt true. Der Balancie-
                   // rungsgrad muß aber modifiziert werden
                   (*root)->balance = -1;
                   break;
```

```
        case +1: // rechter Teilbaum war bisher höher da im
                 // linken Teilbaum angehängt wurde, ist die
                 // Balance nun ausgeglichen. // Die Gesamt-
                 // höhe des TeilBaumes mit aktuellem Knoten
                 // als Wurzel hat sich nicht verändert
                 (*root)->balance = 0;
                 bal_changed  = false;
                 break;

        case -1: // linker Teilbaum war vorher schon höher;
                 // wurde jetzt noch höher und muß daher
                 // ausgeglichen werden !
                 // Dazu muß unterschieden werden, ob der
                 // linke Teilbaum nach rechts oder links
                 // verlängert wurde
                 WurzelVerlaengTeilbaum = (*root)->left;
                 switch(WurzelVerlaengTeilbaum->balance)
                 { case -1:
                      // Linker Teilbaum wurde nach
                      // links verlängert. Es wird
                      // einfach rotiert
                      (*root)->left
                         = WurzelVerlaengTeilbaum->right;
                      WurzelVerlaengTeilbaum->right = *root;
                      (*root)->balance = 0;
                      *root = WurzelVerlaengTeilbaum;
                      (*root)->balance = 0;
                      break;

                   case +1:
                      // Linker Teilbaum wurde nach rechts
                      // verlängert. Es wird doppelt rotiert
                      NeueWurzel
                         = WurzelVerlaengTeilbaum->right;
                      WurzelVerlaengTeilbaum->right
                         = NeueWurzel->left;
                      (*root)->left     = NeueWurzel->right;
                      NeueWurzel->left
                         = WurzelVerlaengTeilbaum;
                      NeueWurzel->right = *root;
                      if ( NeueWurzel->balance == +1 )
                         WurzelVerlaengTeilbaum->balance=-1;
                      else
                         WurzelVerlaengTeilbaum->balance=0;
                      if ( NeueWurzel->balance == -1 )
                         (*root)->balance = +1;
                      else
                         (*root)->balance = 0;
                      *root = NeueWurzel;
                      (*root)->balance=0;
                      break;
                 }
                 bal_changed = false;
                 break;
      }
    }
  }
  else if ( newKey > (*root)->key )
```

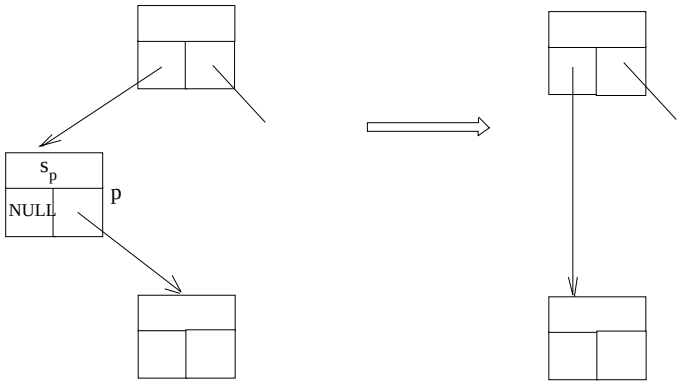
```
{ //neuen Knoten im rechten Teilbaum einhängen
  bal_changed=insertNode1(&(*root)->right,newKey,newInfo);
  // Einfügen ist damit erledigt; nun wird nun balanciert
  if ( bal_changed )
  { // Abhängig vom bisherigen Balancierungsgrad des
    // aktuellen Knotens sind verschiedene Aktionen nötig
    switch ( (*root)->balance )
    {
      case  0: // Teilbäume unter aktuellem Knoten waren
               // bisher ausgeglichen. Da im rechten Teil-
               // baum eingefügt wurde,ist jetzt der linke
               // Teilbaum niedriger. bal_changed bleibt
               // true, da der Teilbaum um 1 verlängert
               // worden ist. Der Balnacierungsgrad muß
               // aber modifiziert werden
               (*root)->balance = +1;
               break;

      case -1: // rechter Teilbaum war bisher niedriger;
               // da im rechten Teilbaum angehängt wurde,
               // ist die Balance nun ausgeglichen. Die
               // Gesamthöhe des Teilbaumes mit der ak-
               // aktuellen Wurzel bleibt aber gleich.
               (*root)->balance = 0;
               bal_changed  = false;
               break;

      case +1: // rechter Teilbaum war vorher schon höher;
               // wurde jetzt noch höher und muß daher
               // ausgeglichen werden ! Nun ist zu unter-
               // scheiden, ob der rechte Teilbaum nach
               // links oder nach rechts verlängert wurde
               WurzelVerlaengTeilbaum = (*root)->right;
               switch(WurzelVerlaengTeilbaum->balance)
               { case +1:
                    // rechter Teilbaum wurde nach rechts
                    // verlängert --> einfach rotieren
                    (*root)->right
                        = WurzelVerlaengTeilbaum->left;
                     WurzelVerlaengTeilbaum->left = *root;
                    (*root)->balance = 0;
                          // Unterbäume der bisherigen
                          // Wurzel sind nun ausgeglichen
                    *root = WurzelVerlaengTeilbaum;
                     (*root)->balance = 0;
                     break;

                 case -1:
                     // rechter Teilbaum wurde nach links
                     // verlängert  --> doppelt rotieren
                     NeueWurzel
                        = WurzelVerlaengTeilbaum->left;
                     WurzelVerlaengTeilbaum->left
                        = NeueWurzel->right;
                     (*root)->right
                        = NeueWurzel->left;
                      NeueWurzel->right
                        = WurzelVerlaengTeilbaum;
```

```
                              NeueWurzel->left  = *root;
                              if ( NeueWurzel->balance == -1 )
                                WurzelVerlaengTeilbaum->balance=+1;
                              else
                                WurzelVerlaengTeilbaum->balance=0;
                                if ( NeueWurzel->balance == +1 )
                                   (*root)->balance = -1;
                                else
                                   (*root)->balance = 0;
                                *root = NeueWurzel;
                                (*root)->balance=0;
                                break;
                        }
                        bal_changed = false
                        break;
                  }
               }
            }
        else
        { cout << "\nSchlüssel " <<newKey<< " ist existiert schon";
          bal_changed = false;
        }
    }
    return(bal_changed);
}

//-------------------------------------------------------------
// balanced    : Hilfsfunktion: prüft AVL-Baum-Bedingung
//-------------------------------------------------------------
bool AVLtree::balanced (AVLnode *root)
{
  int  hl, hr;               // Höhe des li./re. Unterbaumes

  if ( !root) return(true);   // leerer Baum ist ausgeglichen
  hl = hoehe( root->left );   // Höhen der Unterbäume bestimmen
  hr = hoehe( root->right);
  // wenn Höhendiffrenz <= 1, untersuche die Unterbäume
  if (abs(hl-hr)<=1)
    return ((balanced(root->left) && balanced(root->right)));
  return (false);             // nicht balanciert !!!
}

//-------------------------------------------------------------
// hoehe       : Hilfsfunktion: bestimmt die Höhe eines Baumes
//-------------------------------------------------------------
int AVLtree::hoehe (AVLnode *node)
{
  int hl, hr;                // Höhe des li./re. Unterbaumes

  if ( !node ) return(0);      // Höhe des leeren Baumes ist 0
  hl = hoehe(node->left)+1;   // Höhen der Unterbäume bestimmen
  hr = hoehe(node->right)+1;
  if (hl<hr)  return(hr);      // liefere größeren Wert zurück
  else        return(hl);
}
```

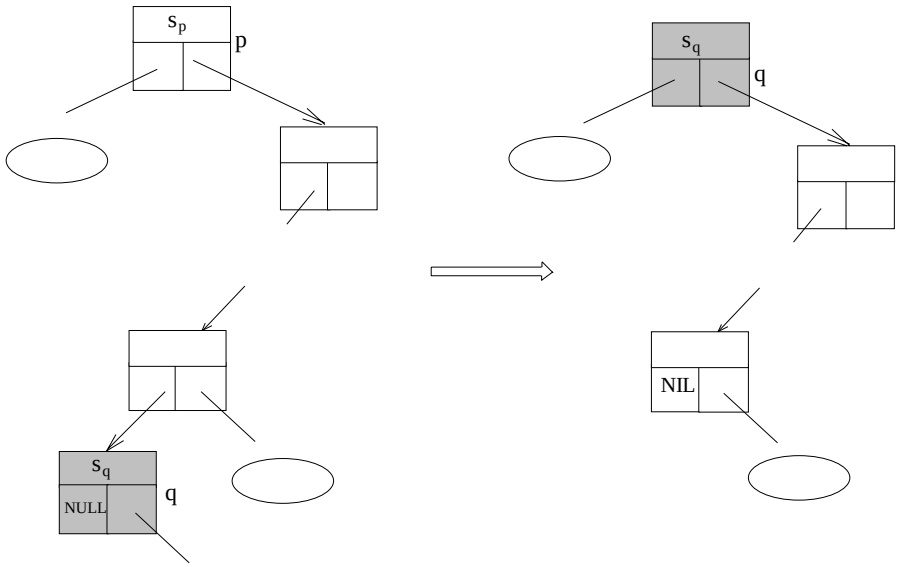
### Löschen von Knoten in AVL-Bäumen

Das Löschen von Knoten in einem AVL-Baum wird zunächst genauso durchgeführt wie bei gewöhnlichen binären Suchbäumen:

– Falls der zu löschende Knoten höchstens einen inneren Knoten als Nachfolger hat, dann kann er einfach aus dem Baum ausgehängt werden, z.B. in der folgenden Situation:



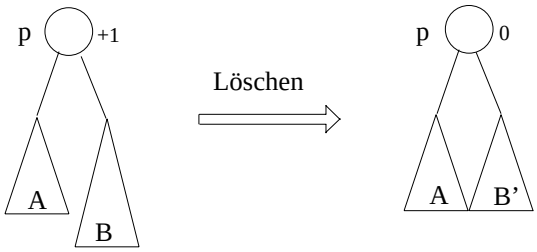

– Falls der zu löschende Knoten zwei innere Knoten als Nachfolger hat, dann wird er zunächst durch seinen symmetrischen Nachfolger ersetzt (Schlüsseltransfer). Dieser hat dann aber maximal einen inneren Knoten als rechten Nachfolger und kann daher wieder einfach entfernt



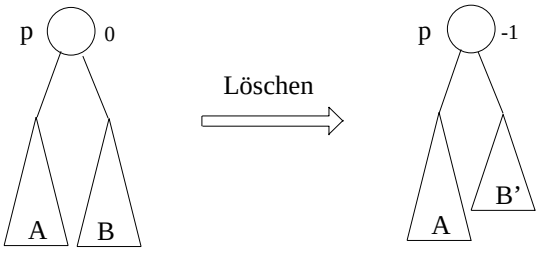

Das Löschen kann die Balance im Baum sowohl lokal als auch global zerstören. Wir müssen daher, ausgehend von der Löschposition, eventuell den Suchpfad zur Wurzel hin zurückverfolgen und an allen Stationen einen Ausgleich durchführen.

Da die Maßnahmen symmetrisch verlaufen für das Löschen und Ausgleichen im linken wie im rechten Unterbaum, betrachten wir nur Situationen, in denen der rechte Unterbaum eines Knotens *p* durch die Löschoperation verkürzt worden ist.

**Fall 1:** Der rechte Unterbaum von *p* war zuvor eine Stufe höher als der linke, d.h. $bal(p) = +1$. Der neue Balancegrad von *p* ist somit = 0. Weil sich die Höhe des Unterbaumes mit der Wurzel *p* aber verändert hat, müssen wir danach evtl. auf übergeordneten Knoten noch einen Ausgleich durchführen. Die folgende Abbildung zeigt dies schematisch:

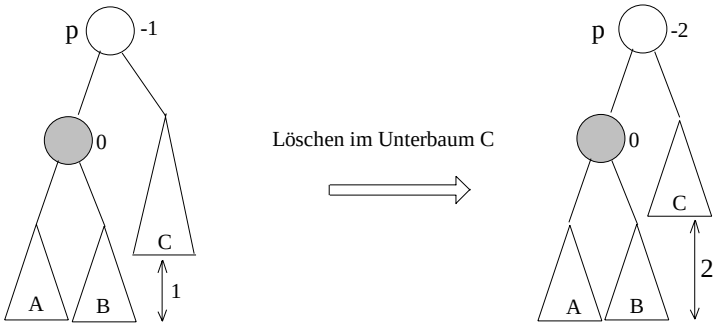



**Fall 2:** Beide Unterbäume von *p* waren zuvor gleich hoch, also $bal(p) = 0$.
Der neue Balancegrad von *p* ist somit $-1$. Weil sich die Höhe des Unterbaumes mit der Wurzel *p* aber nicht verändert hat, müssen wir keine weiteren Ausgleichsmaßnahmen durchführen. Die folgende Abbildung zeigt dies schematisch:

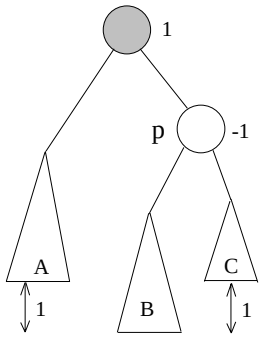



**Fall 3:** Der rechte Unterbaum von *p* war schon vor dem Löschen um eine Stufe niedriger als der linke, d.h. $bal(p) = -1$. Er ist durch das Löschen um eine weitere Stufe verkürzt worden. Wir unterscheiden dann folgende Fälle, die beim Ausgleichen unterschiedlich zu behandeln sind:

**Fall 3.1:** Der linke Nachfolger von *p* hat zwei gleich hohe Unterbäume. Die Höhen der Unterbäume A, B, C müssen in diesem Fall gleich sein. Das Zwischenergebnis des Löschens im Unterbaum C ist nachfolgend gezeigt:

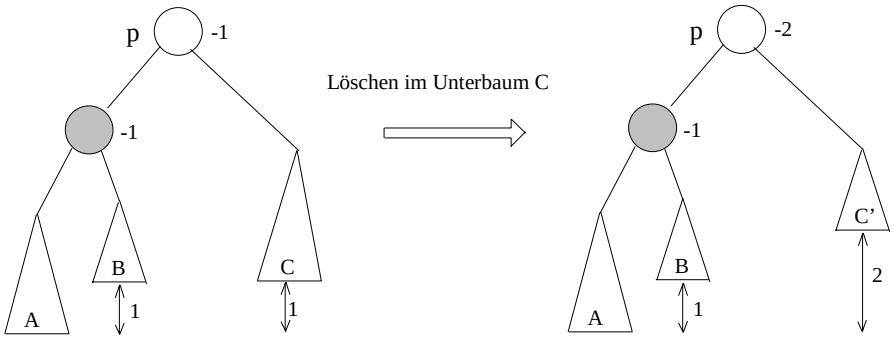


Der Ausgleich erfolgt in diesem Fall durch eine Rotation nach rechts, wobei der linke Nachfolger von *p* zur neuen Wurzel wird und die Balancegrade entsprechend angepaßt werden. Als Ergebnis erhält man:

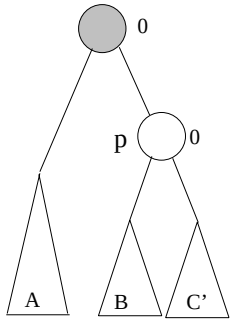



Da die Gesamthöhe des Teilbaumes unverändert geblieben ist, sind keine Ausgleichsmaßnahmen auf höheren Ebenen des Baumes erforderlich.

**Fall 3.2:** Der linke Nachfolger von *p* hat zwei verschieden hohe Unterbäume und den Balancierungsgrad –1. Die folgende Abbildung stellt das Löschen in dieser Situation dar.

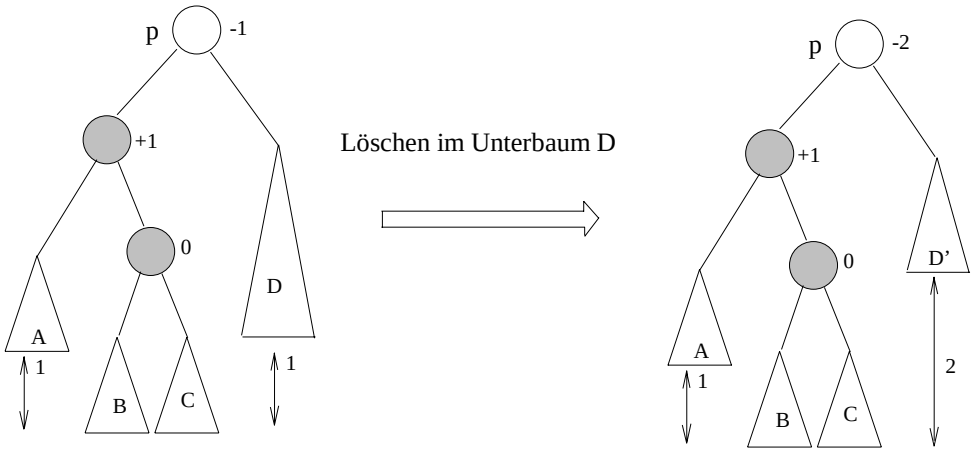



In diesem Fall erfolgt der Ausgleich ebenfalls durch eine Rotation nach rechts, wobei der linke Nachfolger von *p* zur neuen Wurzel wird und die Balancegrade entsprechend angepaßt werden. Als Ergebnis erhält man:

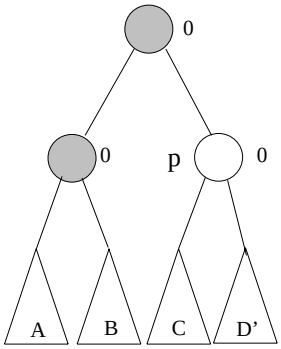



Die Gesamthöhe hat sich um 1 vermindert, d.h. wir müssen anschließend auch auf höheren Levels ausgleichen.

**Fall 3.3:** Der linke Nachfolger von *p* hat zwei verschieden hohe Unterbäume und den Balacierungsgrad +1. Die folgende Abbildung zeigt das Löschen in dieser Situation:




Der Ausgleich erfolgt in dieser Situation durch eine Doppelrotation und wir erhalten:




Die dargestellte Situation ist bezüglich der Balancegrade ein Spezialfall: Wir haben angenommen, daß die Unterbäume B und C gleich hoch sind. Ist dies nicht der Fall so müssen nach dem Ausgleich die Balancegrade der an der Rotation beteiligten Knoten entsprechend anders eingestellt werden. Da sich die Gesamtöhe des Unterbaums verändert hat, müssen wir in diesem Fall auch auf höheren Levels bei Bedarf ausgleichen.

### C++ - Methode für das Löschen in einem AVL-Baum

```
void AVLtree::deleteNode (int delKey)
{ delNode1 ( &root, delKey); }

bool AVLtree::delNode1 (AVLnode **node, int delKey)
{ AVLnode  *toDel;             // symm. Nachf. des zu löschenden Knoten
  bool      bal_changed=false; // Ausgleich in den U-Bäumen verletzt

  if(!(*node)) ;               // Sonderfall leerer Baum
  else
  { if (delKey < (*node)->key) // Knoten liegt im linken Unterbaum
    { bal_changed = delNode1 (&((*node)->left), delKey);
      if (bal_changed) balanceLeft(&(*node), &bal_changed);
    }
    else
    { if (delKey > (*node)->key )    // Knoten liegt im re. Teilbaum
      { bal_changed = delNode1 (&((*node)->right), delKey);
        if (bal_changed) balanceRight(&(* node), &bal_changed);
      }
      else                           // jetzt ist der Knoten gefunden
      { if ((*node)->left == NULL )  // es gibt keinen linken Nachfolger
        { toDel       = *node;
          *node       = (*node)->right;
          bal_changed = true;
          cout << "Knoten  " << toDel->key << "  wird gelöscht !\n\n";
          delete(toDel);
        }
        else
        { if ((*node)->right == NULL)// es gibt keinen re. Nachfolger
          { toDel  = *node;
            *node = (*node)-> left;
            bal_changed = true;
            cout << "Knoten  " << toDel->key << "  wird gelöscht !\n\n";
            delete(toDel);
          }
          else                       // jetzt gibt es zwei Nachfolger
          { delPtr = *node;
            keyTransfer (&(*node)->right, &bal_changed);
            if (bal_changed)  balanceRight(&(*node), &bal_changed);
          }
        }
      }
    }
  }
  return (bal_changed);
}
```

```
//-------------------------------------------------------------------------
// balanceLeft: stellt die Balance wieder her, wenn ein Knoten im linken
//              Unterbaum gelöscht wurde und dabei ein Ungleichgewicht entstand.
//-------------------------------------------------------------------------
void AVLtree::balanceLeft (AVLnode **p, bool *bal_changed)
{
  AVLnode  *p1, *p2;

  switch ((*p)->balance)
  {   case -1:  // der linke Unterbaum war vorher höher als der rechte
                // nach dem Löschen sind beide gleich hoch
                (*p)->balance = 0;
                *bal_changed = true;
                break;

      case  0:  // beide Unterbäume waren vorher gleich hoch
                // Nach außen ändert sich die Balance nicht weiter
                (*p)->balance = 1;
                *bal_changed   = false;
                break;

      case +1:  // Der linke Unterbaum war schon vorher kürzer; jetzt
                // ist eine Ausgleichsoperation fällig
                p1 = (*p)->right;
                switch (p1->balance)
                {
                  case 0:
                  case 1:  // Rotation durchführen
                           (*p)->right = p1->left;
                           p1->left    = *p;
                           if (p1->balance == 0)
                           {  p1->balance   = -1;
                              (*p)->balance = 1;
                              *bal_changed  = false;;
                           }
                           else
                           {  p1->balance   = 0;
                             (*p)->balance = 0;
                           }
                           *p = p1;
                           break;

                  case -1: // Doppelrotation
                           p2           = p1->left;
                           (*p)->right   = p2->left;
                           p1->left      = p2->right;
                           p2->left      = *p;
                           p2->right     = p1;
                           if (p2->balance == -1) p1->balance   = 1;
                           else                   p1->balance   = 0;
                           if (p2->balance == +1) (*p)->balance = -1;
                           else                   (*p)->balance =  0;
                           *p          = p2;
                           p2->balance = 0;
                           break;
                }
                break;
  }
}
```

```
//-------------------------------------------------------------------------
// balanceRight: stellt die Balance wieder her, wenn ein Knoten im rechten
//               Unterbaum gelöscht wurde und dabei ein Ungleichgewicht entstand.
//-------------------------------------------------------------------------
void AVLtree::balanceRight (AVLnode **p, bool *bal_changed)
{
   AVLnode *p1, *p2;

   switch ((*p)->balance)
   {   case +1: // der linke Unterbaum war vorher höher als der rechte
                // nach dem Löschen sind beide gleich hoch
                (*p)->balance = 0;
                *bal_changed  = true;
                break;

       case  0: // beide Unterbäume waren vorher gleich hoch
                // Nach außen ändert sich die Balance nicht weiter
                (*p)->balance  = -1;
                *bal_changed   = false;
                break;

       case -1: // Der linke Unterbaum war schon vorher kürzer; jetzt
                // ist eine Ausgleichsoperation fällig
                p1 = (*p)->left;
                switch (p1->balance)
                { case  0:
                  case -1: // Rotation durchführen
                           (*p)->left = p1->right;
                           p1->right  = *p;
                           if (p1->balance == -1)
                           { p1->balance   =  0;
                             (*p)->balance = 0;
                           }
                           else
                           { p1->balance   =  1;
                             (*p)->balance = -1;
                             *bal_changed  = false;
                           }
                           *p = p1;
                           break;

                 case +1:  // Doppelrotation
                           p2           = p1->right;
                           (*p)->left   = p2->right;
                           p1->right    = p2->left;
                           p2->left     = p1;
                           p2->right    = *p;
                           if (p2->balance == -1) (*p)->balance = 1;
                           else                   (*p)->balance = 0;
                           if (p2->balance == 1)  p1->balance   = -1;
                           else                   p1->balance   =  0;
                           *p         = p2;
                           p2->balance = 0;
                           break;
                }
                break;
  }
}
```

```
//-------------------------------------------------------------------------
// KeyTransfer: sucht den symmetrischen Nachfolger, führt den
//              Schlüsseltransfers aus und löscht den freigewordenen
//              Knoten des symmetrischen Nachfolgers
//-------------------------------------------------------------------------
void AVLtree::keyTransfer (AVLnode **node, bool * bal_changed)
{  if ((*node)->left != NULL)
   { keyTransfer (&(*node)->left, &(*bal_changed));
     if (*bal_changed) balanceLeft (&(*node), &(*bal_changed));
   }
   else
    { delPtr->key  = (*node)->key;
      strncpy (delPtr->info, (*node)->info, INFLEN);
      delPtr->info[INFLEN]=0;
      delPtr       = *node;
      *node        = (*node)->right;
      *bal_changed = true;
      cout << "Knoten  " << delPtr->key << "  wird gelöscht !\n\n";
      delete (delPtr);
   }
}
```

## 6.9 Analyse von AVL-Bäumen

Wir wollen nun den Aufwand bestimmen, der in AVL-Bäumen für das Suchen, Einfügen und Löschen von Schlüsseln erforderlich ist. Dabei sind wir an einer worst case-Abschätzung interessiert, also an der Frage, wie hoch der Aufwand maximal werden kann.

Zuerst untersuchen wird eine verwandte Fragestellung, nämlich, wieviele Schlüssel ein AVL-Baum der Höhe $h$ mindestens enthält und kommen zu einer untereren Schranke $N_{\min}$ für deren Anzahl :

$$N_{\min} \geq f(h)$$

Daraus ergibt sich umgekehrt aber eine obere Schranke für die maximale Höhe $h_{\max}$ eines AVL-Baumes mit einer vorgegebenen Anzahl von $N$ Knoten:

$$h_{\max} \leq f^{-1}(N)$$

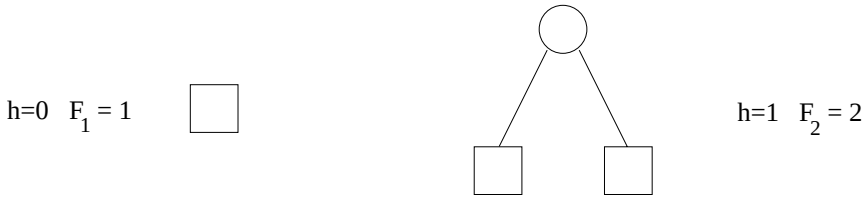
**Bemerkung 1:**

> Ein AVL-Baum der Höhe $h$ hat mindestens $F_{h+1}$ Blätter, wobei $F_h$ die Folge der Fibonacci-Zahlen ist.

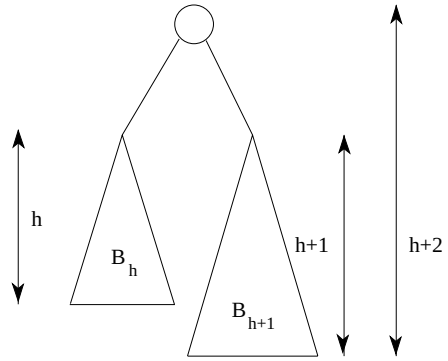
Begründung: Die Fibonacci-Zahlen sind definiert durch die Rekursionsformel

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_{h+2} &= F_{h+1} + F_h \end{aligned}$$

Wir begründen die Behauptung durch Induktion über die Höhe $h$ des Baumes. Für $h = 0$ und $h = 1$ haben die AVL-Bäume die folgende Struktur




Die Behauptung ist daher für $h = 0$ und $h = 1$ richtig. Ein AVL-Baum der Höhe $h + 2$ mit minimaler Anzahl von Blättern hat den folgenden Aufbau, wobei jeder der beiden Unterbäume von der Höhe $h$ bzw. $h$+1 selbst eine minimale Anzahl von Blättern enthalten muß.



Die Induktionsannahme gilt für die beiden Unterbäume. Die Anzahl der Blätter im Gesamtbaum ist aber offenbar die Summe der Blätter der beiden Unterbäume, also $F_{h+2} = F_{h+1} + F_h$ .

**Bemerkung 2:**

Die Fibonacci-Zahlen $F_h$ haben den Wert $F_h = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{h+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{h+1}\right)$

Diese geschlossene Darstellung der Fibonacci-Zahlen begründen wir wieder durch Induktion. Für die Indices 0 und 1 kann man die Gültigkeit durch Einsetzen nachrechnen:

$$F_0 = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1\right) = \frac{1}{\sqrt{5}}\frac{2\sqrt{5}}{2} = 1$$

$$F_1 = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^2 - \left(\frac{1-\sqrt{5}}{2}\right)^2\right) = \frac{1}{\sqrt{5}}\left(\frac{1+2\sqrt{5}+5}{4} - \frac{1-2\sqrt{5}+5}{4}\right) = \frac{1}{\sqrt{5}}\frac{4\sqrt{5}}{4} = 1$$

Für den Induktionsschluß nehmen wir an, daß die behauptete Beziehung für $h$ = 0, 1, ..., h+1 bereits besteht und weisen sie für $h$+2 nach:

$$F_{h+1} + F_h = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - \left(\frac{1-\sqrt{5}}{2}\right)^{h+2}\right) + \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{h+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{h+1}\right)$$

$$= \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{h+1}\frac{3+\sqrt{5}}{2} - \left(\frac{1-\sqrt{5}}{2}\right)^{h+1}\frac{3-\sqrt{5}}{2}\right]$$

$$= \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{h+1}\left(\frac{1+\sqrt{5}}{2}\right)^2 - \left(\frac{1-\sqrt{5}}{2}\right)^{h+1}\left(\frac{1-\sqrt{5}}{2}\right)^2\right]$$

$$= \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} - \left(\frac{1-\sqrt{5}}{2}\right)^{h+3}\right]$$

$$= F_{h+2}$$

**Bemerkung 3:**

Ein AVL-Baum der Höhe $h$ hat mindestens $N_{\min} = 1.1708 * 1.618^h$ Blätter.

Dies begründen wir wie folgt:

Wegen $\left|\frac{1-\sqrt{5}}{2}\right| < 1$ können wir für $F_h$ eine Abschätzung erhalten:

$$F_h \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+1} = \frac{1+\sqrt{5}}{2\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h} = 0.7236*1.618^h$$

Nach Bemerkung 1 ist aber $N_{\min} = F_{h+1}$ , oder $N_{\min} = 0.7236*1.618^{h+1} = 1.1708*1.618^h$

## Bemerkung 4:

Ein AVL-Baum mit $N$ Blättern und deshalb $N-1$ inneren Knoten hat maximal die Höhe

$$h \le 1.44*\log_2(N)$$

Begründung: Wie obene bereits dargestellt erhalten wir die gewünschte Abschätzung der maximalen Höhe aus der Abschätzung für die Zahl von Knoten. Es gilt nach Bemerkung 3:

$$N \ge 1.1708*1.618^h$$
$$\log_2(N) \ge \log_2(1.1708) + h*\log_2(1.618)$$

also

$$h \le \frac{\log_2(N)}{\log_2(1.618)} - \frac{\log_2(1.1708)}{\log_2(1.618)}$$
$$h \le 1.44*\log(N)$$

**Ergebnis:**

AVL-Bäume besitzen eine obere Schranke für den maximalen Aufwand einer Operation, die vergleichbar ist mit dem mittleren Aufwand einer Operation in zufällig aufgebauten binären Suchbäumen.

## 6.10 B-Bäume

Bei balancierten Binärbäumen haben wir bereits eine wesentliche Reduktion des Aufwandes für das Suchen, Löschen und Einfügen erreicht:

- für vollständige Binärbäume beträgt der Aufwand maximal $_2\log(N)$
- für zufällige Binärbäume beträgt der Aufwand im Mittel $1.39\ _2\log(N)$
- für AVL-Bäume beträgt der Aufwand maximal $1.44\ _2\log(N)$

Für die Verwendung von Binärbäumen bei der Organisation von Dateizugriffen ist dies jedoch immer noch nicht schnell genug, wie das unten stehende Beispiel zeigt. Wir werden im folgenden B-Bäume (Bayer-Bäume) als einen Baumtyp kennenlernen, der die Zahl von notwendigen Plattenzugriffen pro Datensatz wesentlich reduziert, nämlich auf eine für weite Bereiche konstante Anzahl. B-Bäume wurden erstmals 1972 von R.Bayer und E.McCreight genauer untersucht.

### Beispiel 1

Wir betrachten Zugriffe zu einer Datei mit 1 Million Datensätzen:

- Die Zugriffszeit der Platte soll 10 msec betragen.
- Jeder Datensatz soll einen Knoten in einem binären Suchbaum darstellen. Der Suchbaum soll vollständig, also optimal organisiert sein. Er hat also daher Höhe

$$h = {}_2\log(10^6) = \frac{6}{{}_{10}\log(2)} \approx 20$$

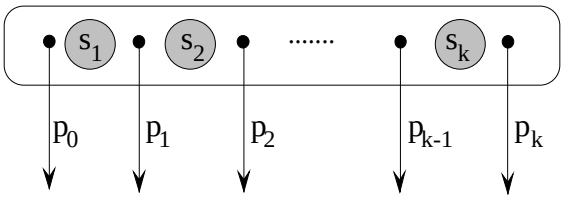
Unter diesen Voraussetzungen dauert jeder Zugriff bis zu 0.2 Sekunden für jeden Datensatz.

### B-Bäume der Ordnung m

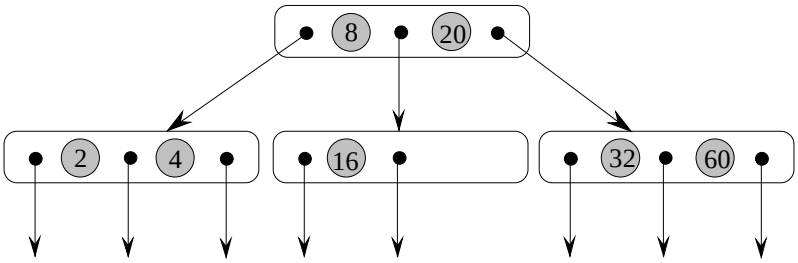
Ein B-Baum der Ordnung m ist ein Mehrwegebaum, der durch die folgenden Eigenschaften charakterisiert wird:

- Alle Blätter haben die selbe Tiefe.
- Die Wurzel hat mindestens zwei Söhne.
- Jeder innere Knoten hat zwischen $\left\lceil \frac{m}{2} \right\rceil$ und $m$ Söhne $p_0, p_1, \cdots\cdots, p_k$ .
- Jeder innere Knoten mit $(k+1)$ Söhnen hat $k$ Schlüssel $s_1 < \cdots\cdots < s_k$ .
- Die Werte aller Schlüssel im Unterbaum mit der Wurzel $p_i$ liegen zwischen $s_i$ und $s_{i+1}$,
- alle Schlüssel in $p_0$ sind kleiner als $s_1$ und alle Schlüssel in $p_k$ sind größer als $s_k$

Wir können einen Knoten daher so darstellen:



**Beispiel 2:** Das folgende Bild zeigt einen B-Baum der Ordnung *m*=3.




### Suchen in einem B-Baum

Aufgrund ihrer Konstruktion erlauben B-Bäume eine Suchstrategie, die sich als eine direkte Verallgemeinerung des Suchens in einem Binärbaum darstellt. Um einen Schlüssel $s$ in einem B-Baum zu suchen, verfahren wir wie folgt:

1. Zunächst suchen wir im Wurzelknoten direkt nach dem Schlüssel $s$.
2. Falls *s* in Schritt 1 nicht gefunden wurde, suchen wir den ersten Schlüssel $s_i$ mit $s < s_i$ :
   - Falls ein solcher Schlüssel existiert, steigen wir über den $s_i$ vorausgehenden Zeiger $p_{i-1}$ in den Unterbaum ab und suchen in dessen Wurzelknoten weiter gemäß Schritt 1.
   - Falls kein solcher Schlüssel existiert, steigen wir über den letzten Zeiger $p_k$ in den Unterbaum ab und suchen in dessen Wurzelknoten weiter gemäß Schritt 1.

Da ein Baumknoten stets ganz im Arbeitsspeicher liegt, können wir für die Suche nach dem Schlüssel $s$ bzw. nach dem ersten Schlüssel $s_i$ mit $s < s_i$ durch ein einfaches lineares oder binäres Suchverfahren realisieren. Der Aufwand für die Suche hängt hauptsächlich davon ab, wie viele Knoten zu untersuchen sind, weil diese vom Externspeicher gelesen werden müssen.

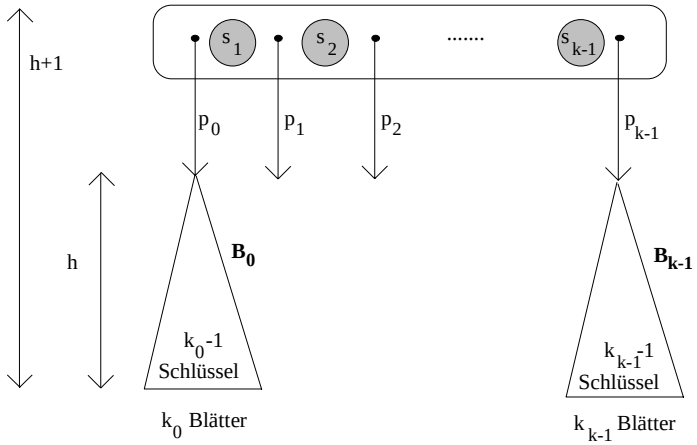
### Komplexitäts-Betrachtung für B-Bäume

Wir wollen nun untersuchen, welcher Aufwand für das Suchen in B-Bäumen erforderlich. Für andere Operationen, z.B. das Einfügen oder Löschen erhöht sich dieser Aufwand noch um eventuell notwendige Ausgleichs-Maßnahmen. Zunächst betrachten wir einige Eigenschaften von B-Bäumen:

**Bemerkung 1:** Ein B-Baum mit $k$ Blättern hat $k-1$ Schlüssel.

Dies begründen wir durch Induktion über die Höhe des Baumes. Für $h = 1$ ist die Aussage nach der Definition von B-Bäumen unmittelbar klar. Wenn wir die Aussage für Bäume bis zur Höhe $h$ voraussetzen, dann gilt sie auch für B-Bäume der Höhe $h+1$, denn

- Wenn der Baum der Höhe $h+1$ aus einer Wurzel $p$ und $k$ Unterbäumen $B_i$ der Höhe $h$ besteht, dann hat jeder Unterbaum mit $k_i$ Blättern auch $k_{i-1}$ Schlüssel nach Induktionsannahme.




Damit hat der Gesamtbaum $\sum_{i=1}^{k} k_i$ Blätter und die Zahl aller Schlüssel beträgt:

$$\sum_{i=1}^{k}(k_i - 1) + (k-1) = \sum_{i=1}^{k} k_i - 1$$

**Bemerkung 2:** Für die minimale bzw. maximale Anzahl von Blättern in einem B-Baum gilt:

$$N_{\min} = 2 * \left\lceil \frac{m}{2} \right\rceil^{h-1} \quad \text{und} \quad N_{\max} = m^h$$

Diese Aussagen sind klar, wenn man berücksichtigt, daß für einen maximal gefüllten B-Baum jeder Knoten $m$ Nachfolger besitzt und für einen minimal ausgebauten die Wurzel $2$ und alle anderen Knoten $\lceil \frac{m}{2} \rceil$ Nachfolger.

**Bemerkung 3:** Für die Höhe eines B-Baumes mit $N$ Schlüsseln gelten folgende Schranken:

$$_m\log(N+1) \le h \le _{\frac{m}{2}}\log\left(\frac{N+1}{2}\right) + 1$$

Ein B-Baum mit $N$ Schlüsseln hat gemäß Bemerkung 1 $(N+1)$ Blätter. Da diese Anzahl immer zwischen der in Bemerkung 2 angegebenen minimalen und maximalen Anzahl liegt, erhalten wir:

$$N_{\min} = 2 * \left\lceil \frac{m}{2} \right\rceil^{h-1} \le N+1 \le m^h = N_{\max}$$

Daraus folgt: $h \le _{\left\lceil \frac{m}{2} \right\rceil}\log\left(\frac{N+1}{2}\right) + 1$ h $\le$ $_{\lceil \frac{m}{2} \rceil}$log $(\frac{N+1}{2})$ + 1

und $h \ge _m \log(N+1)$

**Beispiel 3:**

Wir betrachten wieder eine Datei mit 1 Million Datensätzen. Die Schlüssel sollen in einem B-Baum der Ordnung $m$=100 organisiert sein. Dann ergibt sich aus Bemerkung 3:

$$_{100}\log(1.000.000) \le h \le {}_{50}\log(500.000) + 1$$

Wegen $${}_m\log(z) = {}_{10}\log(z) / {}_{10}\log(m)$$

erhalten wir daraus: $$\frac{1}{{}_{10}\log(100)} * {}_{10}\log(1.000.000) \le h \le \frac{1}{{}_{10}\log(50)} * {}_{10}\log(50.000) + 1$$

Dies ergibt: $$3 \le h \le 0.59 * 5.70 + 1 < 5$$

Wie man leicht nachprüft, genügen auch bei einer Vervierfachung des Datenvolumens maximal 5 Zugriffe pro Satz, so daß der Aufwand für Operationen in B-Bäumen praktisch konstant ist.

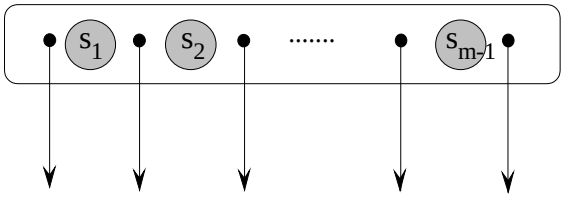
Bei einer Plattenzugriffszeit von 10 msec ergibt sich daher eine konstante Zugriffszeit von 0,04 sec pro Datensatz, statt der mittleren Zugriffszeit von 0,2 sec wie in Beispiel 1. Der Preis dafür ist der zusätzliche Speicherbedarf für die nicht vollständig gefüllten Knoten des B-Baumes.

**Einfügen in B-Bäume**

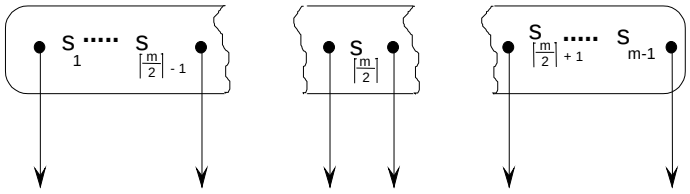
Das Einfügen eines neuen Schlüssels s in einen B-Baum der Ordnung m erfordert zunächst das Suchen der Einfügestelle, das wir schon am Anfang dieses Abschnitts betrachtet haben. Falls der einzufügende Schlüssel noch nicht im Baum vorkommt, endet die Suche immer in einem Blattknoten, in der untersten Ebene des Baumes. Wir betrachten dann den Vaterknoten $p$ dieses Blattes und müssen zwei Fälle unterscheiden:

**Fall 1:** Der Knoten $p$ hat noch Platz für einen weiteren Schlüssel.
Der neue Schlüssel kann daher eingefügt werden und die Operation ist beendet.

**Fall 2:** Der Knoten $p$ ist bereits mit der Maximalzahl von ($m$-1) Schlüsseln gefüllt. In diesem Fall fügen wir den neuen Schlüssel $s$ zunächst trotzdem ein und erhalten einen Knoten mit $m$ Schlüsseln:




Danach teilen wir den Knoten in der Mitte wie folgt auf:

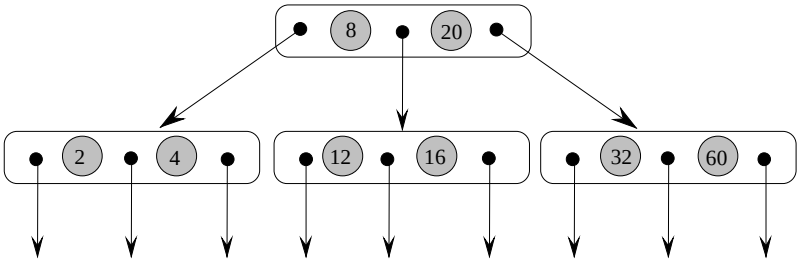


Die Anfangs- und Endstücke werden nun neue B-Baum-Knoten auf der untersten Baumebene. Da der Knoten vor dem Einfügen bereits maximal gefüllt war, haben die beiden neuen Knoten den minimalen Füllgrad, also $\left\lceil \frac{m}{2} \right\rceil$.

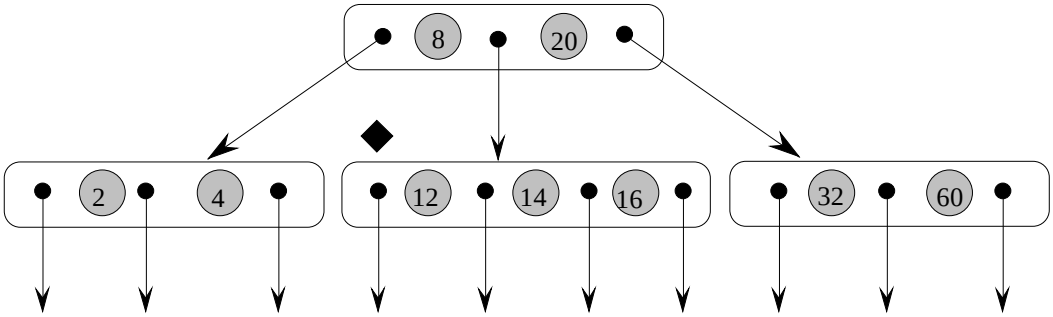
Der mittlere Knoten wird nun auf die selbe Weise in den darüberliegenden Knoten aufgenommen. Da dieser u.U. schon maximal gefüllt ist, muß man evtl. auch ihn aufteilen und im schlimmsten Fall den Suchpfad ganz zurückverfolgen.

**Beispiel 4:**

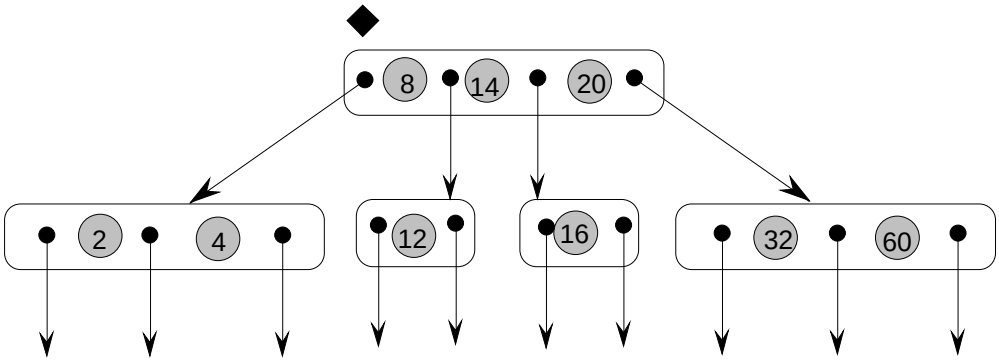
Wenn wir den Schlüssel $s$=12 neu in den B-Baum aus Beispiel 2 einfügen, dann liefert dies nach Fall 1 den folgenden B-Baum:



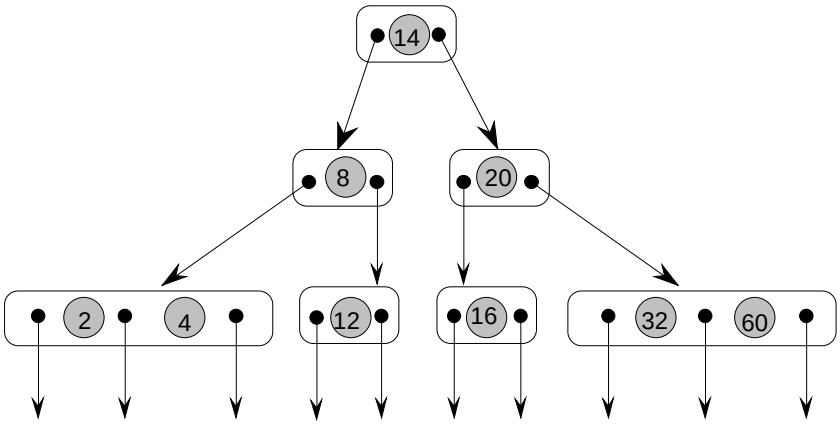

Wenn wir nun einen weiteren Schlüssel $s$=14 einfügen, dann erhalten wir zunächst einen Überlauf in dem mit ♦ markierten Knoten:




Aufsplitten dieses Knotens liefert nun einen Überlauf im Wurzelknoten des B-.Baumes:



Durch Aufsplitten erhalten wir einen um eine Stufe höheren B-Baum mit einer neuen Wurzel:

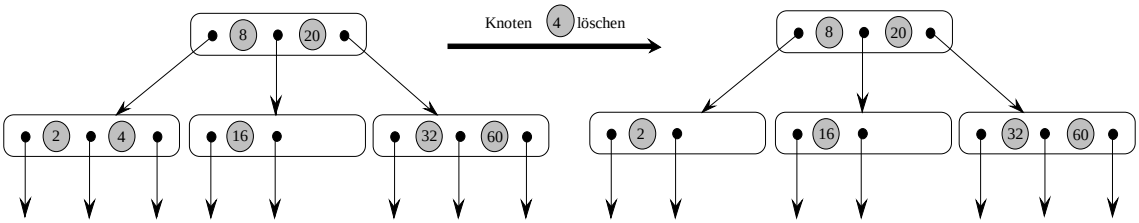



Damit haben wir wieder einen B-Baum hergestellt, der den neuen Knoten $s = 14$ enthält.

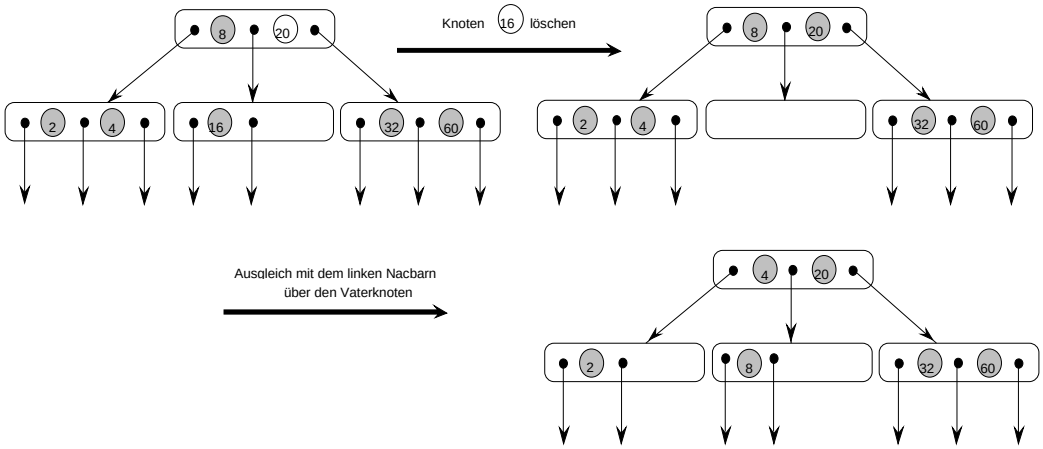
### Löschen in B-Bäumen

Wenn in einem B-Baum ein Schlüssel zu löschen ist, dann können wir uns darauf beschränken, daß er in einem Knoten auf der untersten Ebene des B-Baumes liegt. Andernfalls können wir ihn durch Schlüssel-Transfer durch seinen größten Vorgänger oder seinen kleinsten Nachfolger ersetzen und dieser liegt sicher auf der untersten Baumebene. Wir unterscheiden dann folgende Fälle:

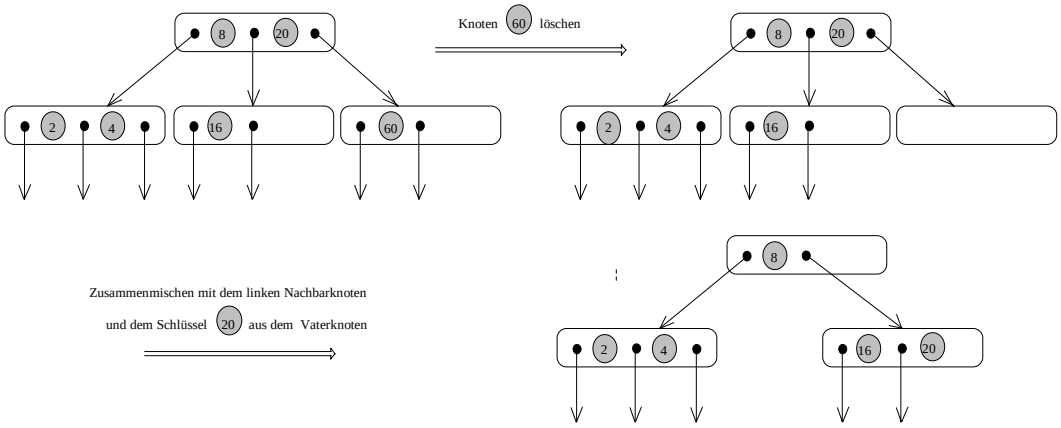
**1. Fall:** Der zu löschende Schlüssel liegt in einem Knoten auf der untersten Ebene des B-Baumes und der Knoten hat noch mehr als die minimale Anzahl von $\left\lceil \frac{m}{2} \right\rceil$ Knoten. In diesem Fall können wir den Schlüssel löschen, ohne daß weitere Maßnahmen erforderlich werden. Das folgende Beispiel zeigt diese Situation:




**2. Fall:** Löschen durch Ausgleichen:
In dem betroffenen Baumknoten auf der untersten Ebene entsteht durch das Löschen ein Unterlauf. Er hat dann $\left\lceil \frac{m}{2} \right\rceil$-1 Schlüssel. Wir nehmen zusätzlich an, daß in einem direkten Nachbarknoten mehr als die minimale Anzahl von Schlüsseln vorkommt und können daher von diesem über den gemeinsamen Vaterknoten Schlüssel herüberziehen bis beide Knoten ausgeglichen sind. Das folgende Beispiel zeigt diese Situation:




**3. Fall:** Löschen durch Mischen:
In dem betroffenen Baumknoten auf der untersten Ebene entsteht durch das Löschen ein Unterlauf. Er hat dann $\left\lceil \frac{m}{2} \right\rceil$-1 Schlüssel. Wir nehmen zusätzlich an, daß in keinem direkten Nachbarknoten mehr als die minimale Anzahl von $\left\lceil \frac{m}{2} \right\rceil$ Schlüsseln vorkommt. In diesem Fall bilden wir einen neuen Knoten aus dem aktuellen, einem seiner Nachbarn und dem Schlüssel im Vaterknoten, der zwischen beiden liegt. Der neue Knoten hat dann m Schlüssel. Im Vaterknoten kann ein Unterlauf entstanden sein, der schlimmsten falls bis zurück zur Wurzel verfolgt und ausgeglichen werden muß. Das folgende Beispiel zeigt diese Situation:



**Die C++ - Klasse Btree**

Sie realisiert eine einfache Dateiverwaltung mit B-Bäumen und ist wie folgt organisiert:

**– Basisstruktur der Datei:**
Alle Zugriffe auf die Datei erfolgen über logische Satznummern. Alle Sätze haben feste Länge. Einen Satz bezeichnen wir als eine Page.
Über dieser Basisstruktur ist die B-Baum-Struktur der Datei implementiert, wobei jeder Knoten des Baumes einen logischen Satz in der Datei belegt..

**– Baum-Anker:**
Der Satz mit der logischen Nummer 0 hat eine besondere Funktion: er enthält Anker-Informationen für die Datei-Organisation:

- FreeRec ist der Anker für die Liste freier Sätze in der Datei.
- RootRec ist die logische Satznummer der B-Baum-Wurzel.
- AllocRec ist die Satznummer des letzten logischen Satzes in der Datei.

Der Ankerknoten wird, solange die Datei geöffnet ist, resident im Arbeitsspeicher gehalten und nur bei Veränderungen der Ankerinformation zurückgeschrieben.

**– Baum-Knoten:**
Alle anderen Sätze der Datei enthalten je einen Knoten und haben folgenden Aufbau:

| numItems | item[0] | item[1] | item[2] | ..... | item[nn] |
|---|---|---|---|---|---|

- numItems ist die Anzahl der gültigen Items in diesem Knoten

Die Items haben den folgenden Aufbau

| key | info | ref |
|---|---|---|

- key ist der ganzzahlige Schlüssel eines Item
- info ist ein char-Feld, als Repräsentation der Item-Information
- ref ist der Zeiger auf den nachfolgenden Unterbaum

Da es einen Zeiger mehr als Schlüssel gibt, wird der erste Zeiger des Knotens in item[0] abgelegt und item[0] wird nur dafür verwendet. Der erste Schlüssel eines Knotens steht in item[1].

**– Freispeicher-Verwaltung:**
Wenn Knoten des B-Baums anläßlich von Löschoperationen freigegeben werden, werden sie in eine Freispeicherliste einghängt und bei Bedarf wiederverwendet.

### Header-Datei mit der Klassendefinition

```
#include <string.h>
#define  INFLEN  7          // Länge der Knoten-Information
#define  NAMLEN  100        // Länge der Knoten-Information
#define  nn   3             // max. Zahl von Items/Page = Ordnung-1
#define  n    2             // Pos. des nach oben zu verlagernden Item

struct Item
{  int  key;                // Knoten-Schlüssel
   char  info[INFLEN+1];    // Knoten-Information
   int   ref;               // Referenznummer
};

struct DPage
{  int numItems;            // Anzahl belegter Items
   Item item[nn+1];         // Item-Liste der DB-Page
};

struct SPage
{  int  FreeRec;            // Anker der Freispeicherliste
   int  AllocRec;           // Anzahl belegter Records in der DB
   int  RootRec;            // Zeiger zur Wurzel
};

struct Buffer               // Puffer für DB-Pages
{  int  pageNum;            // Page-Nummer
   union  {  DPage   dPage; // 1. Alternative: normale Daten-Page
             SPage   sPage; // 2. Alternative: Page 0
          } page;
};

class Btree
{
  public:
    Btree (char *DBname)   { openDatabase(DBname);   }
    virtual ~Btree()       { closeDatabase();        }
    void  openDatabase     (char * DBname);
    void  closeDatabase    (void);
    void  insertKey        (int key, char * info);
    void  deleteKey        (int delKey);
    void  printDatabase    (void);
    void  inOrderDatabase  (void);

    int     freeRec;       // Zeiger zum 1. Satz der Freispeicherliste
    int     rootRec;       // Index der Baumwurzel
    int     allocRec;      // Größe der Datei in Record-Anzahl

    FILE    *DBfile;       // Datenbank-Datei
    Buffer  BF;            // Puffer für Ein-/Ausgabe von Pages
    Buffer  rootBF;        // Puffer für Root-Record
    Buffer  Page0BF;       // Puffer für Page 0

    bool    found;         // globaler Schalter für Update
    int     risen;         // globaler Schalter für Update
    int     x;             // globaler Index für Suche und keyTransfer
```

```
  private:
    void    writePage0     (void);
    void    allocateRec    (int& x);
    void    savePage       (Buffer& BF);
    void    loadPage       (int x);
    void    searchItem     (int SearchKey, int& x, bool& found);
    void    splitPage      (Buffer& BF, Item& item, int x);
    void    update         (int node, int& rise, Item& risenItem,
                                                  int key, char *data);
    void    copyItem       (Item& ziel, Item quelle);
    void    compensate     (int precedent, int node, int path,
                                                     bool& underflow);
    void  keyTransfer      (Buffer &BF, int node1, bool& underflow);
    void  delKey           (int node, bool& underflow, int delKey);
    void  releasePage      (int z);
    void  printDB          (int x, int sp);
    void  inOrderDB        (int x, int sp);
};
```

### Funktionen für den Dateizugriff auf unterer Ebene

```
//---------------------------------------------------------------------
//  openDatabase : Legt eine leere Datenbank an mit einer Page 0
//                 oder öffnet eine vorhandene Datenbank.
//---------------------------------------------------------------------
void Btree::openDatabase (char * DBname)
{   // Versuchen, eine vorhandene Datenbank zu oeffnen
    if ( (DBfile=fopen(DBname,"rb+")) == NULL )
    { // Datenbank neu anlegen, wenn es noch keine gibt
      if ( (DBfile=fopen(DBname,"wb+")) == NULL )
      { cout <<"Datei" << DBname << "kann nicht geöffnet werden !\n";
        return;
      }
      // Datenbank-Ankerfelder initialisieren und Page 0 ausgeben
      freeRec  =  0;       // Anker der Freispeicherliste
      allocRec =  0;       // Größe der Datenbank
      rootRec  =  0;       // Zeiger zur Wurzel
      writePage0 ();       // Page 0 in die Datenbank schreiben
    }
    else
    { // Anker der vorhandenen Datenbank einlesen
      fseek (DBfile, 0, SEEK_SET);
      fread (&(Page0BF.page),sizeof (DPage), 1, DBfile);
      freeRec  = Page0BF.page.sPage.FreeRec;
      allocRec = Page0BF.page.sPage.AllocRec;
      rootRec  = Page0BF.page.sPage.RootRec;

      // Root-Page der vorhandenen Datenbank einlesen
      fseek (DBfile, rootRec * sizeof (DPage), SEEK_SET);
      fread (&(rootBF.page), sizeof (DPage), 1, DBfile);
      rootBF.pageNum = rootRec;
    };
}
```

```
//---------------------------------------------------------------------
//  CloseDatabase :  Schließt die Datenbank-Datei
//---------------------------------------------------------------------
void Btree::closeDatabase (void)
{  fclose (DBfile); }

//---------------------------------------------------------------------
//  writePage0 :   Gibt den Datenbank-Root-Record auf die Datei aus
//---------------------------------------------------------------------
void Btree::writePage0 (void)
{   Page0BF.page.sPage.AllocRec = allocRec;    // Puffer für fwrite
    Page0BF.page.sPage.FreeRec  = freeRec;     // initialisieren
    Page0BF.page.sPage.RootRec  = rootRec;
    fseek (DBfile,(size_t)0,SEEK_SET);         // Page 0 in die DB
    fwrite(&(Page0BF.page), (size_t)sizeof(DPage), (size_t)1, DBfile);
}

//---------------------------------------------------------------------
//  allocateRec : Ermittelt die Satznummer einer freien Page in der DB
//---------------------------------------------------------------------
void  Btree::allocateRec (int& x)
{   Buffer  BF;
    if (freeRec == 0)   // Falls Freispeicher-Liste leer ist:
    { allocRec++;       // Datenbankgröße erhöhen durch neuen Record
      x = allocRec;     // neue Größe in den Parameter x
      writePage0();     // Page 0 aktualisieren
    }
    else
    { x = freeRec;      // Nächste freien Page liefern
      fseek (DBfile, (x) * sizeof(DPage), SEEK_SET);
      fread (&(BF.page.dPage), sizeof(DPage), 1, DBfile);
      freeRec = BF.page.sPage.FreeRec; // aus der FSB-Liste aushängen
      writePage0();     // Page 0 aktualisieren
    }
}

//---------------------------------------------------------------------
//  loadPage :  Lädt eine Page aus der Datenbank in den Puffer BF
//              Die Root-Page kann direkt aus rootBF kopiert werden
//---------------------------------------------------------------------
void Btree::loadPage (int x)
{  if (x == rootRec)    // Root-Page aus rootBF holen
     memcpy (&BF, &rootBF, sizeof (Buffer));
    else                // andere Pages holen
    { BF.pageNum = x;   // Page-# im Puffer eintragen
      fseek (DBfile, (size_t)(x * sizeof(DPage)), SEEK_SET);
      fread (&(BF.page), (size_t)sizeof(DPage), (size_t)1, DBfile);
    }
}
```

```
//---------------------------------------------------------------------
//  savePage :  Schreibt eine Page zurück in die Datenbank
//---------------------------------------------------------------------
void Btree::savePage (Buffer& BF1)
{  // falls es sich um die Root-Page handelt, rootBF aktualisieren
  if (BF1.pageNum == rootRec) memcpy (&rootBF, &BF1, sizeof(Buffer));

  // Page in die Datenbank schreiben
  fseek (DBfile, BF1.pageNum * sizeof(DPage), SEEK_SET);
  fwrite (&(BF1.page), (size_t)sizeof(DPage), (size_t)1, DBfile);
}

//---------------------------------------------------------------------
//  searchItem :  Binäre Suche nach einem Item in der aktuellen Page
//---------------------------------------------------------------------
void  Btree::searchItem (int SearchKey, int& x, bool& found)
{  intr, l;                                // Zeiger für binäre Suche
   l = 0;                                  // vor das erste Item zeigen
   r =(int)(BF.page.dPage.numItems + 1); // hinter letztes Item zeigen
   found = false;                          // Item bisher nicht gefunden
   while ( (l+1 < r) && (found==false) ) // Binäres Suchen
   { x = (l+r) / 2;
     if (SearchKey == BF.page.dPage.item[x].key)      found = true;
     else if (SearchKey < BF.page.dPage.item[x].key)  r = x;
      else                                             l = x;
   }
   if (found == false) x=l;               // gesuchtes Item ist rechts
}                                         // von Pos. x einzuordnen
```

### B-Baum-Operationen

```
//---------------------------------------------------------------------
// insertKey : trägt ein neues Item in die Datenbank ein
//---------------------------------------------------------------------
void Btree::insertKey (int key, char *data)
{   Buffer  newRoot;
    int     rootSplit;
    Item    rootItem;

    rootSplit = false;
    update (rootRec, rootSplit, rootItem, key, data);
    if (rootSplit == true)               // neue Root anlegen
    { allocateRec (newRoot.pageNum);     // neue Page zuweisen
      newRoot.page.dPage.numItems = 1;   // sie wird 1 Item enthalten
      newRoot.page.dPage.item[0].ref = rootRec;// Zeiger auf alte Root
      copyItem (newRoot.page.dPage.item[1], rootItem);
      savePage (newRoot);                // neue Wurzel sichern
      memcpy(&rootBF, &newRoot, sizeof(BF));   // residente Wurzel
      rootRec = newRoot.pageNum;         // DB-Ankerfeld aktualisieren
      writePage0();                      // Page 0 aktualisieren
    }
}
```

```
//---------------------------------------------------------------------
//  deleteKey :  löscht einen Schlüssel aus der Datenbank
//---------------------------------------------------------------------
void Btree::deleteKey  (int key)
{   int  toDel;
    bool underflow;

    delKey (rootRec, underflow, key); // lösche den Schlüssel
    if ((underflow) && (rootBF.page.dPage.numItems)==0)
    {  toDel   = rootRec;             // Wurzel zum Löschen vormerken
       loadPage(rootBF.page.dPage.item[0].ref);// neue Wurzel laden
       memcpy (&rootBF, &BF, sizeof (Buffer)); // residente Root
       releasePage(toDel);            // alte Wurzel löschen
       rootRec = rootBF.pageNum;      // Satznummer der neuen Wurzel
       writePage0();                  // Page0 aktualisieren
    }
}
```

### Hilfsfunktionen für das Einfügen im B-Baum

```
//---------------------------------------------------------------------
//  copyItem  : kopiert ein Item
//---------------------------------------------------------------------
void  Btree::copyItem (Item& ziel, Item quelle)
{   ziel.key = quelle.key;
    strcpy(ziel.info, quelle.info);
    ziel.ref = quelle.ref;
}

//---------------------------------------------------------------------
//  splitPage : Spaltet eine Page auf
//---------------------------------------------------------------------
void  Btree::splitPage (Buffer& BF, Item& item, int x)
{   Item SplitItem;                // Platz f.d. herauszuhebende Item
    Buffer SplitBF;                // Zusatzpuffer für die neue Page
    int z;                         // Laufvariable

    allocateRec(SplitBF.pageNum);   // reserviere eine freie Page
    if (x < n) // neues Item kommt in die linke Page:
    { // Item auf Position n nach SplitItem bringen
      copyItem (SplitItem, BF.page.dPage.item[n]);
      // Items rechts von Position x nach rechts verschieben
      // und neues Item einsetzen
      for (z=n-1; z>=x+1; z--)
         copyItem(BF.page.dPage.item[z+1], BF.page.dPage.item[z]);
      copyItem(BF.page.dPage.item[x+1], item);
    }
    else if (x > n) // neues Item kommt in die rechte Page
    { // herauszunehmendes Item nach SplitItem bringen
      copyItem(SplitItem, BF.page.dPage.item[n+1]);
      // Platz schaffen für das neue Item und dieses einsetzen
      for (z=n+2; z<=x; z++)
         copyItem(BF.page.dPage.item[z-1], BF.page.dPage.item[z]);
      copyItem(BF.page.dPage.item[x], item);
    }
    else  // neues Item kommt selbst nach SplitItem
```

```
      copyItem(SplitItem, item);

    // Puffer für die rechte Page aufbereiten
    SplitBF.page.dPage.item[0].ref = SplitItem.ref; // erster Zeiger
    SplitItem.ref = SplitBF.pageNum; // re. Page an SplitItem anhängen
    copyItem(item, SplitItem);       // SplitItem übergeben
    for (z=n+1; z <= nn; z++)        // Items in SplitBF übernehmen
      copyItem(SplitBF.page.dPage.item[z-n], BF.page.dPage.item[z]);

    // Anzahlen gültiger Items aktualisieren
    BF.page.dPage.numItems = n;
    SplitBF.page.dPage.numItems = nn - n;

    // rechten Puffer in die Datenbank schreiben
    savePage (SplitBF);
}

//---------------------------------------------------------------------
//  update : fügt ein Item in e. (Teil-)Baum mit der Wurzel node ein
//---------------------------------------------------------------------
void  Btree::update (int node, int& rise, Item& risenItem,
                                    int key, char *data)
{   int  x,z;
    if (node == 0)                    // Item ersetzt ein Blatt:
    { rise           = true;          // Höhe wird gößer
      risenItem.key  = key;           // Schlüssel eintragen
      strcpy(risenItem.info, data);   // Datenfeld eintragen
      risenItem.ref  = 0;             // Zeiger initialisieren
    }
    else                              // Wurzel des Unterbaumes laden
    { loadPage(node);
      searchItem (key, x, found);     // Item im Wurzel-Record suchen
      if (found == true)              // ggf. Datenfeld ersetzen
      {  strcpy (BF.page.dPage.item[x].info, data);
         savePage (BF);
      }
      else                       // falls nicht vorhanden, weitersuchen
      {  risen = false;          // Flag für aufsteigendes Item=false
         update(BF.page.dPage.item[x].ref,risen,isenItem,key,data);
         if (risen == true)      // falls ein Item aufsteigen muss
         { loadPage (node);      // Wurzel-Page laden
           if (BF.page.dPage.numItems<nn)// falls Wurzel Platz hat
           {  BF.page.dPage.numItems++ ; // Item an Pos. x+1 eintragen
              for (z=(int)(BF.page.dPage.numItems-1) ; z>=x+1; z--)
              copyItem(BF.page.dPage.item[z+1],BF.page.dPage.item[z]);
              copyItem(BF.page.dPage.item[x+1], risenItem);
              rise = false;
           }
           else                   // falls Wurzel voll ist, zerlegen
           {  splitPage (BF, risenItem, x);
             rise = true;         // risenItem muss aufsteigen
           }
           savePage (BF);         // aktuelle Wurzel sichern
         }
      }
    }
}
```

### Hilfsfunktionen für das Löschen im B-Baum

```
//---------------------------------------------------------------------
//  releasePage  Baumknoten löschen
//---------------------------------------------------------------------
void Btree::releasePage (int z)
{   BF.page.sPage.FreeRec = freeRec;     // vor die FSB-Liste hängen
    fseek (DBfile, z * sizeof (DPage), SEEK_SET);// Record holen
    fwrite (&(BF.page), sizeof (DPage), 1, DBfile);
    freeRec = z;                         // neuer Anker der FSB-Liste
    writePage0();
}

//---------------------------------------------------------------------
//  compensate :  Ausgleich nach einer Löschoperation
//---------------------------------------------------------------------
void Btree::compensate (int precedent, int node, int path, bool& un-
derflow)
{   int    neighbour;
    int    numBF2, numBF3;
    int    x, z;
    Buffer BF1, BF2, BF3;

    loadPage (node);                     // Sohnknoten in BF1 bringen
    memcpy (&BF1, &BF, sizeof(Buffer));
    loadPage (precedent);                // Vaterknoten in BF3 bringen
    memcpy (&BF3, &BF, sizeof (Buffer));
    numBF3 = BF3.page.dPage.numItems;

    // Sohn ist >>nicht<< der rechteste Knoten
    if (path < numBF3)
    { path++;               // Zeiger zum rechten Nachbarn des Sohnes
      neighbour = BF3.page.dPage.item[path].ref;
      loadPage (neighbour);  // rechten Nachbarn in BF2 bringen
      memcpy (&BF2, &BF, sizeof (Buffer));
      numBF2 = BF2.page.dPage.numItems;

      // Key vom Vaterknoten in den untergelaufenen Knoten schieben
      BF1.page.dPage.item[n-1].key = BF3.page.dPage.item[path].key;
      BF1.page.dPage.item[n-1].ref = BF2.page.dPage.item[0].ref;
      strcpy (BF1.page.dPage.item[n-1].info,
              BF3.page.dPage.item[path].info);
      // feststellen, ob BF2 Schluessel abgeben kann:
      x = ((numBF2-n+2) / 2);
      if (x > 0)  // BF2 kann abgeben ---> Ausgleichen
      { // die Schlüssel von BF2 nach BF1 übertragen
        for (z=1; z<x; z++)
            BF1.page.dPage.item[z+n] = BF2.page.dPage.item[z];

        // einen Key auch in den Vaterknoten bringen
        copyItem(BF3.page.dPage.item[path], BF2.page.dPage.item[x]);
        BF3.page.dPage.item[path].ref = neighbour;
```

```
      // im Knoten BF2 aufräumen
      BF2.page.dPage.item[0].ref = BF2.page.dPage.item[x].ref;
      numBF2 = (numBF2 -x);
      for (z = 1; z <= numBF2; z++)
        copyItem(BF2.page.dPage.item[z], BF2.page.dPage.item[z+x]);
      BF2.page.dPage.numItems = numBF2;
      BF1.page.dPage.numItems = (n+x-2);

      // alle beteiligten Knoten zurückschreiben
      savePage (BF1); savePage (BF2); savePage (BF3);
      underflow = false;
    }
    else  // BF2 kann nichts abgeben --> Mischen
    {  for (z=1; z<n; z++)         // BF2 nach BF1 übertragen
          copyItem(BF1.page.dPage.item[n+z-1], BF2.page.dPage.item[z]);
       for (z=path; z<numBF3; z++) // Keys in BF3 zusammenschieben
          copyItem(BF3.page.dPage.item[z], BF3.page.dPage.item[z+1]);
      // Schlüselanzahlen in BF2 und BF3 korrigieren,
      // Flag für Unterlauf im Vaterknoten setzen
      BF1.page.dPage.numItems = 2*(n-1);
      BF3.page.dPage.numItems = (numBF3-1);
      if (numBF3<n) underflow = true;
      else          underflow = false;

      // alle beteiligten Knoten zurückschreiben, BF2 löschen
      savePage (BF1); savePage (BF3);
      releasePage (neighbour);
    }
  }
  // Sohn ist der rechteste Knoten
  else
  { // linken Nachbarn in BF2 laden
    neighbour = BF3.page.dPage.item[path-1].ref;
    loadPage (neighbour);
    memcpy (&BF2, &BF, sizeof (Buffer));

    // feststellen, ob BF2 Schluessel abgeben kann:
    numBF2 = BF2.page.dPage.numItems;
    x = ((numBF2-n+2) / 2);
    if (x > 0) // BF2 kann abgeben ---> Ausgleichen
    { // am Anfang von BF1 Platz schaffen für n-1 Items
      for (z=n-1; z>0; z--)
         copyItem(BF1.page.dPage.item[z+x], BF1.page.dPage.item[z]);

      // Position x aus dem Vaterknoten übernehmen
      copyItem(BF1.page.dPage.item[x], BF3.page.dPage.item[path]);
      BF1.page.dPage.item[x].ref = BF1.page.dPage.item[0].ref;

      // x Items aus BF2 übernehmen in BF1
      numBF2 = (numBF2-x+1);
      for (z=x-1; z>0; z--)
         copyItem(BF1.page.dPage.item[z],BF2.page.dPage.item[z+numBF2]);
      // linkesten Zeiger in BF2 korrigieren
      BF1.page.dPage.item[0].ref = BF2.page.dPage.item[numBF2].ref;
```

```
        // Key aus BF2 in den Vaterknoten holen, Key-# korrigieren
        copyItem(BF3.page.dPage.item[path], BF2.page.dPage.item[numBF2]);
        BF3.page.dPage.item[path].ref = node;
        BF2.page.dPage.numItems = numBF2-1;
        BF1.page.dPage.numItems = (n-2+x);

        // alle beteiligten Knoten zurückschreiben
        savePage (BF1); savePage (BF2); savePage (BF3);
        underflow = false;
      }
      else  // BF2 kann nichts abgeben --> Mischen
      { //  einen Key aus dem Vaterknoten nach BF2 übertragen
        copyItem(BF2.page.dPage.item[numBF2+1], BF3.page.dPage.item[path]);
        BF2.page.dPage.item[++numBF2].ref = BF1.page.dPage.item[0].ref;

        // Zum Mischen BF1 nach BF2 übertragen
        for (z=1; z<n-1; z++)
           BF2.page.dPage.item[z+numBF2] = BF1.page.dPage.item[z];
        // Anzahl der Knoten korrigieren
        BF2.page.dPage.numItems = nn-1;
        BF3.page.dPage.numItems = numBF3 - 1;
        if (numBF3<n) underflow = true;
        else          underflow = false;

        // beteiligte Knoten zurückschreiben , Knoten BF1 löschen
        savePage (BF2); savePage (BF3);
        releasePage (node);
      }
    }
}

//---------------------------------------------------------------------
//  keyTransfer:   löscht per Keytransfer mit dem symm. Vorgänger
//---------------------------------------------------------------------
void Btree::keyTransfer (Buffer& BF3, int node1, bool& underflow)
{   int node2;
    int numBF;
    Buffer BF1;

    loadPage (node1);                     // Teilbaumwurzel laden
    memcpy (&BF1, &BF, sizeof (Buffer));  // und in lokalen Puffer
    numBF = BF1.page.dPage.numItems;
    node2 = BF1.page.dPage.item[numBF].ref;// rechtester Vorw.zeiger
    if (node2 != 0) // Falls symm. Vorgänger weiter rechts unten liegt
    { keyTransfer (BF3, node2, underflow); // symm. Vorg. weitersuchen
      if (underflow) compensate (node1, node2, numBF, underflow);
    }
    else
    { // symmetrischer Vorgänger gefunden: Keytransfer durchführen
      BF3.page.dPage.item[x].key = BF1.page.dPage.item[numBF].key;
      strcpy (BF3.page.dPage.item[x].info, BF1.page.dPage.item[numBF].info);
      // Falls Ziel des Keytransfers die Wurzel ist, rootBF updaten
      if (BF3.pageNum == rootRec) memcpy (&rootBF, &BF3, sizeof(Buffer));
      BF1.page.dPage.numItems = --numBF;  // in der Quelle des
      savePage (BF1);                     // Keytransfers größten Key
                                          // entfernen
```

```
      if (numBF<n-1) underflow = true;     // ggf. Unterlauf melden
      else           underflow = false;
    }
}

//---------------------------------------------------------------------
//  delKey :  Hilfsfkt.: löscht einen Schlüssel aus der Datenbank
//---------------------------------------------------------------------
void Btree::delKey (int node, bool& underflow, int key)
{   int x, y, z;
    Buffer BF3;

    if (node == 0)  underflow = false;     // leerer Baum oder Blatt
    else
    {
      loadPage (node);
      searchItem (key, x, found);
      if (found)                           // zu Schlüssel gefunden
      { y = BF.page.dPage.item[x-1].ref;   // → vorausgehender U-Baum
        if (y == 0)                        // jetzt ist dieser leer
        { // lösche Schlüssel direkt in node
          BF.page.dPage.numItems=BF.page.dPage.numItems-1;
          if (BF.page.dPage.numItems < n-1) underflow = true;
          else                              underflow = false;
          for (z = x; z <= BF.page.dPage.numItems; z++)
            copyItem(BF.page.dPage.item[z], BF.page.dPage.item[z+1]);
          savePage (BF);                   // schreibe Knoten zurück
        }
        else                               // falls U-Baum nicht leer
        { memcpy (&BF3, &BF, sizeof(Buffer));
          keyTransfer (BF3, y, underflow); // Löschen per Keytransfer
          savePage (BF3);                  // mit symm. Vorgänger
          if (underflow) compensate (node, y, x-1, underflow);
        }
      }
      else                                 // weitersuchen
      { y = BF.page.dPage.item[x].ref;
        delKey (y, underflow, key);
        if (underflow) compensate (node, y, x, underflow);
      }
    }
}
```

# 6.11 B-Bäume und VSAM-Dateiorganisation

Die Analyse von B-Bäumen hat gezeigt, daß sie sich zur Implementierung von Zugriffsmethoden vor allem deshalb gut eignen, weil sie über weite Bereiche konstanten Zugriffsaufwand erfordern. Für einen praktikablen Einsatz sind allerdings noch einige weitere Maßnahmen erforderlich, die wir nun am Beispiel der Zugriffsmethode VSAM kurz skizzieren.

VSAM bietet drei Zugriffsvarianten an:

- eine sequentielle Dateiorganisation (ESDS = Entry sequenced Data Sets),
- eine Dateiorganisation, die Sätze gleicher Länge über ihre Satznummer adressiert (RRDS = Relative Record Data Sets) und
- eine Dateiorganisation mit direkter Adressierung über Schlüssel, die an einer festen Position in den Sätzen stehen (KSDS = Key Sequenced DataSets). Diese letzte Zugriffsmethode basiert auf einer Variante der B-Bäume, den B'-Bäumen.

## 6.11.1 Einfache Blockungsverfahren

An der Schnittstelle zwischen Anwendung und Zugriffsmethode sind für die Anwendungen logische Sätze die Elementareinheit für den Informationsaustausch. Beim Abspeichern auf externe Speichermedien ist es aus verschiedenen Gründen zweckmäßig, logische Sätze zu übergeordneten Einheiten zusammenzufassen, den logischen Blöcken bzw. Kontrolleinheiten.

Logische Blöcke sind also eine Zusammenfassung logischer Datensätze zu unterscheiden von den physikalischen Blöcken, den elementaren Speicherbereichen auf dem externen Datenträger. Physikalische Blöcke spielen für unsere Thematik keine Rolle.

Die Vorteile der Blockung liegen in der Platzersparnis auf dem externen Datenträger und dem Zeitgewinn beim Zugriff auf Informationen, der Preis dafür ist der erhöhte Bedarf an Pufferspeicher und der zusätzliche Laufzeit-Aufwand für das Blocken und Entblocken der Daten.

Die Anzahl der Blöcke ist i.a. wesentlich kleiner als die Anzahl der Blöcke. Daraus resultiert eine Zeit- und Platzersparnis bei Zugriffen:

#### Zeitgewinn durch Blockung

Beim Zugriff auf geblockte Sätze ergibt sich die Zeiteinsparung dadurch, daß jeder Zugriff auf den externen Datenträger mehrere logische Sätze auf einmal in den Arbeitsspeicher lädt. Diese können von dort bei Bedarf wesentlich dann schneller geholt werden als durch weitere Plattenzugriffe.

Bei einer indizierten Dateiorganisation führt die geringere Anzahl von außerdem zu einem weniger tiefen Index und damit zu einer geringere Zahl von Index-Zugriffen pro Leseauftrag.

#### Platzgewinn durch Blockung

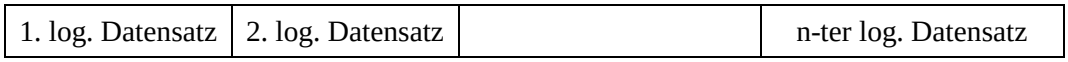
Durch die geringere Anzahl von Blöcken verringert sich auch die Anzahl der Blocklücken und der Umfang der Adressierungsinformation zwischen den Sektoren. Bei Platten mit fester physikalischer Blocklänge werden die Sektoren durch die Blockung besser ausgenutzt, wodurch sich ebenfalls ein Platzvorteil ergibt.

### Logische Blöcke

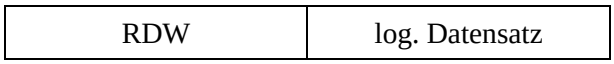
Bei der Abspeicherung logischer Sätze in Dateien unterscheiden wird nach den Merkmalen geblockt/ungeblockt und feste/variable Satzlänge die folgenden vier Blockformate:

ungeblockt, feste Satzlänge:


geblockt, feste Satzlänge**:**




ungeblockt, variable Satzlänge**:**




geblockt, variable Satzlänge**:**


Bei festen Satz- und Blocklängen genügt es, die Netto-Information abzuspeichern, während bei variablen Längen zusätzliche Verwaltunginformationen abgespeichert werden müssen. Dabei bedeuten:

BDW Block Descriptor Word; es enthält die Länge des gesamten Datenblocks

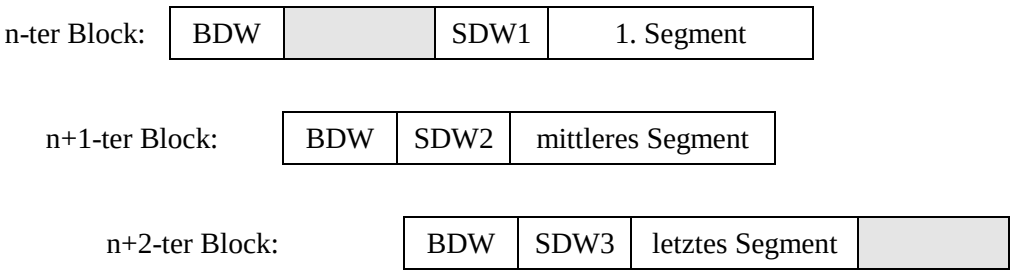
RDW Record Descriptor Word; es enthält die Länge des logischen Datensatzes

### Gespannte Datensätze

Alle vier oben angegebenen Blockformate setzen voraus, daß kein Satz die Grenzen eines logischen Datenblocks übersteigt. Dies wird aber gerade bei variablen Satzlängen gewünscht, um eine optimale Ausnutzung des Blockformats zu erreichen.

Man zerlegt dazu einen Datensatz, der sich über mehrere Blöcke erstreckt so in Segmente, daß diese keine Blockgrenzen überschreiten. Segmente werden durch Segment-Deskriptoren (SDW) beschrieben, die ähnlich wie RDW interpretiert werden, aber zusätzlich erlauben, den originalen Satz beim Entblocken wieder zusammenzusetzen.

Die folgende Skizze zeigt, als Beispiel einen Satz, der sich über drei Segmente erstreckt. Der Segment-Deskriptor enthält neben der Segmentlänge auch Informationen darüber, ob es sich um das erste, ein mittleres oder das letzte Segment eines logischen Satzes handelt.



## 6.11.2 Kontroll-Intervalle

Logische Blockung führt zwar hinsichtlich Platzbedarf und Zugriffszeit zu einer effizienten Speichertechnik. Sie besitzt aber erhebliche Nachteile, sobald gespeicherte Sätze dynamisch verändert werden sollen, z.B. bei

- nachträglicher Änderung der Satzlängen,
- Löschen von Sätzen mit Wiederverwendung des freigegebenen Platzes sowie
- Einfügen von Sätzen.

Kontrollintervalle wurden von IBM im Zusammenhang mit der Zugriffsmethode VSAM eingeführt, um diese Nachteile zu vermeiden. Sie besitzen eine Leerraum-Verwaltung, welche die dynamische Veränderungen ihres Inhalts gestatten.

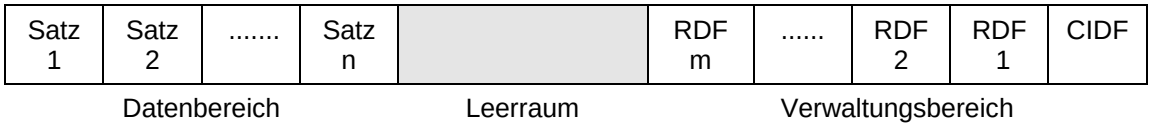
### Struktur von Kontrollintervallen

Die Größe des Kontroll-Intervalls ist ein Datei-Attribut und stets ein Vielfaches von 512 Bytes.

Ein Kontrollintervall besteht aus drei Bereichen:

- dem Datenbereich, der die logischen Sätze aufnimmt,
- dem Leerraum zur Verlängerung vorhandener Sätze bzw. zur Aufnahme neuer Sätze
- dem Bereich zur Verwaltung von Datensätzen und Leerraum

Die folgende Skizze zeigt diesen Aufbau:




Dabei bedeuten:

CIDF Das Control Interval Definition Field: es steht stets ganz am Ende des Kontrollintervalls und gibt Position und Länge des Leerraums an.

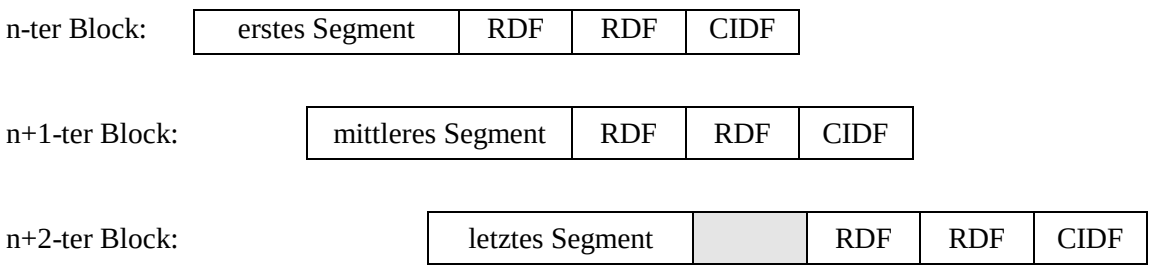
RDF Record Definition Field: Die RDF stehen am Ende des Kontrollintervalls vor dem CDIF. Sie haben die umgekehrte Reihenfolge wie de logischen Sätze.

### Kontrollintervalle mit gespannten Datensätzen

Ähnlich wie bei logischer Blockung kann man Datensätze, die nicht in ein Kontrollintervall passen, in mehrere Segmente zerlegen. Dabei gelten folgende Regeln:

- Jedes beteiligte Kontrollintervall enthält genau ein Segment und keine weiteren Sätze
- Ein Segment wird durch zwei RDF beschrieben, die seine Länge und einen Änderungsstand beinhalten. Falls bei einem Sytemzusammenbruch nicht alle zum selben gespannten Datensatz gehörenden Kontroll-Intervalle zurückgeschrieben werden, kann man dies am Änderungszähler feststellen.

Die folgende Skizze zeigt die Aufteilung eines gespannten Datensatzes auf mehrere Kontrollintervalle:



#### Zugriff auf logische Blöcke und Sätze

Wenn ein Anwendungsprogramm einen logischen Satz anfordert, muß die Zugriffsmethode diesen bereitstellen. Dazu geht sie in zwei Schritten vor:

Schritt 1: Zunächst holt sie den zugehörigen logischen Block bzw. das Kontroll-Intervall vom externen Speichermedium.

Schritt 2: Wenn der logische Block im Arbeitsspeicher steht, wird mit einem internen Suchverfahren der gewünschte Satz im Block gesucht.

Zum effizienten Ablegen und Wiederfinden von Datenblöcken in der Datei können verschiedene Organisationsformen und Strategien eingesetzt werden, z.B. die folgenden:

- Sequentielles Abspeichern der Blöcke in der Reihenfolge der Ankunft ist die einfachste, aber auch die unflexibelste Methode.
- Direkter Zugriff über Satznummern ist möglich, wenn alle Datensätze gleich lang sind.
- Mit Hash-Verfahren kann man jeden Datenblock durch eine Adreßrechnung finden kann.
- Mit Baumstrukturen läßt sich der Zugriff über einen Index effizient realisieren

### 6.11.3 VSAM-Dateiorganisation

Die Analyse von B-Bäumen hat gezeigt, daß sie sich zur Implementierung von Zugriffsmethoden vor allem deshalb gut eignen, weil sie über weite Bereiche konstanten Zugriffsaufwand erfordern. Für einen praktikablen Einsatz sind allerdings noch einige weitere Maßnahmen erforderlich, die wir nun am Beispiel der Zugriffsmethode VSAM kurz skizzieren. VSAM bietet drei Zugriffsvarianten an:

- eine sequentielle Dateiorganisation (ESDS = Entry sequenced Data Sets),
- eine Dateiorganisation, die Sätze gleicher Länge über ihre Satznummer adressiert (RRDS = Relative Record Data Sets) und
- eine Dateiorganisation mit direkter Adressierung über Schlüssel (KSDS = Key Sequenced DataSets). Sie basiert auf einer Variante der B-Bäume, den B'-Bäumen.

#### Kontroll-Intervalle als Knotenpunkte von B-Bäumen

Oben haben wir als eine Weiterentwicklung der einfachen Blockung Kontroll-Intervalle kennengelernt. Sie stellen zusätzlich zur Blockung der logischen Datensätze eine Leerraum-Verwaltung zur Verfügung, die das nachrägliche Einfügen, und Löschen ganzer Sätze, aber auch Längenänderungen der Sätze erlauben. Zur Erinnerung wird hier noch einmal das Format eines Kontroll-Intervalls gezeigt:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Satz 1 | Satz 2 | ..... | Satz n | Leerraum | RDF m | ..... | RDF 2 | RDF 1 | CIDF |

Diese Struktur macht Kontroll-Intervalle optimal geeignet zur Aufnahme je eines B-Baum-Knotens, da die Leerraum-Verwaltung das nachträgliche Einfügen und Löschen von Schlüsseln im Knoten leicht ermöglicht.

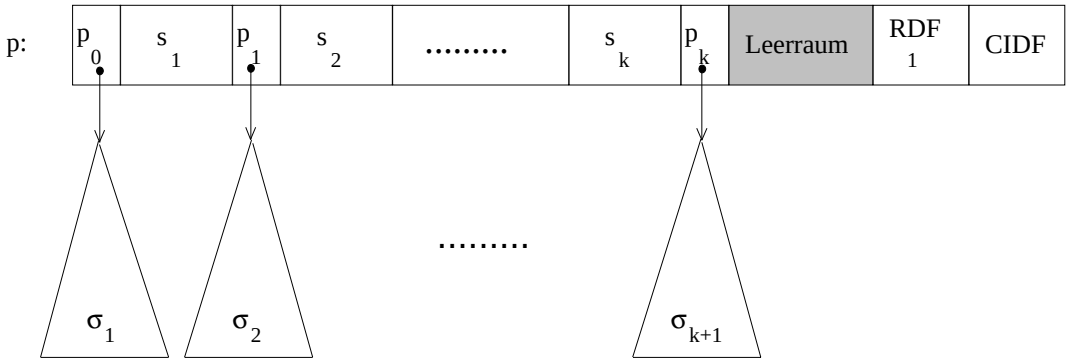
#### B'-Bäume

Da bei B-Bäumen die Baumhöhe und damit der Zugriffsaufwand stark von der Ordnung m abhängt, modifiziert man die B-Baum-Definition dahingehend, daß die Knoten auf den oberen Levels nur Schlüssel enthalten, während die Datensätze ausschließlich in den Knoten der untersten Ebene des Baums gespeichert werden. Diese Variante wird als B'-Baum bezeichnet.

Da die Schlüssel-Länge i.a. nur wenige Bytes groß ist, kann man die Ordnung m sehr hoch wählen, etwa $m$=100 bis $m$=200. Eine Datei zerfällt daher in zwei Komponenten mit jeweils eigenem Kontroll-Intervall-Format: eine Index-Komponente und eine Datenkomponente.

### Die Index-Komponente

Sie enthält Kontroll-Intervalle, welche je einen Baum-Knoten aus den oberen Levels eines B'-Baums repräsentieren. Je Kontroll-Intervall wird ein einziger Satz gespeichert, der einen vollständigen B'-Baum-Knoten enthält. Das Kontroll-Intervall hat daher den folgenden Aufbau:



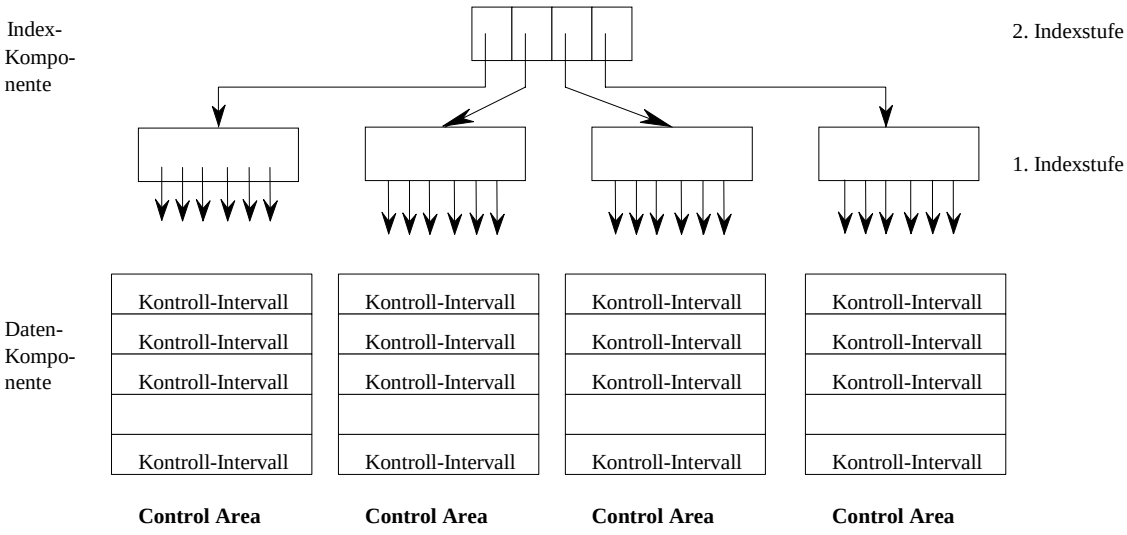

Dabei sind in allen Bäumen mit der Wurzel $p_i$ die darin gespeicherten Schlüssel $s_{i+1}$ größer oder gleich dem Schlüssel $s_i$ aber kleiner oder gleich dem Schlüssel $s_{i+1}$ .

Im Gegensatz zu den B-Bäumen dienen die Schlüssel der Index-Komponente von B'-Bäumen also nur als Wegweiser für die Suche. Sie trennen die Schlüsselbereiche der Datensätze und müssen selbst nicht unbedingt als Schlüssel eines Datensatzes vorkommen. Sie sind in dieser Hinsicht eine Verallgemeinerung der binären Blattsuchbäume.

### Die Datenkomponente

Die Kontroll-Intervalle der Datenkomponente enthalten die Datensätze einer Datei und zwar aufsteigend nach Schlüsseln sortiert. Jedes Kontroll-Intervall der Datenkomponente repräsentiert einen Knoten aus der niedrigsten Ebene des B'-Baumes.

Aus Gründen der Zugriffs-Optimierung wird zusammenhängenden Index-Bereichen je ein Block zusammenhängender Kontroll-Intervalle, eine Control Area, zugewiesen. Die Größe der Control Areas und ihre Zuordnung zu den Indexbereichen ist ein Parameter bei der Datei-Kreation.



# 7 Hash-Verfahren

In den vorausgehenden Kapiteln haben wir Baumstrukturen kennengelernt als einen Mechanismus zum Ablegen und Wiederfinden von Datensätzen aufgrund ihres Schlüssels. Wesentliche Merkmale dieser Methode sind :

- Die Menge der möglichen Schlüssel ist eine geordnete Menge, so daß je zwei Schlüssel verglichen werden können.
- Die Schlüssel charakterisieren einen Datensatz eindeutig.
- Suchen, Einfügen und Löschen von Datensätzen basieren ausschließlich auf dem Vergleichen von Schlüsseln.

Die Verfahren der gestreuten Speicherung (Hash-Verfahren), die wir in diesem Kapitel betrachten, benutzen einen grundsätzlich anderen Ansatz: Sie verzichten auf den Vergleich von Schlüsseln und berechnen stattdessen direkt aus einem Schlüssel die Adresse eines Satzes. Ähnlich wie bei ausgeglichenen Bäumen kommt man bei guten Hash-Verfahren mit einer konstanten Zahl von Zugriffen aus.

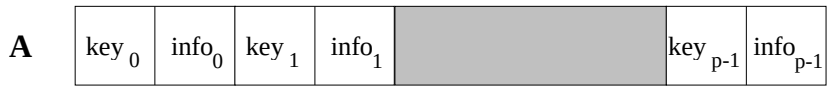
## 7.1 Grundlagen der gestreuten Speicherung

### Hash-Tabellen

Ebenso wie Baumstrukturen können die Hashverfahren sowohl für den Zugriff auf residente Datenstrukturen wie auch auf Datensätze in Dateien verwendet werden. Zur Darstellung der grundsätzlichen Methoden beschränken wir uns hier auf die folgende spezielle Situation:

- Der Bereich zur Aufnahme von Datensätzen ist ein Feld **A** (Hash-Tabelle), das mit einer festen Zahl von Zellen vorreserviert wird.
- Alle Datensätze sind gleich lang und in jeder Zelle kann ein Datensatz gespeichert werden.
- Die zulässigen Schlüssel sind positive, ganze Zahlen.

Daher können wir uns eine Hash-Tabelle wie folgt vorstellen:




### Die Hash-Funktion

Ein Hash-Verfahren soll direkt aus dem numerischen Schlüssel die Adresse eines Datensatzes bestimmen, also den Index der Hash-Tabelle, unter dem der Satz abgelegt oder gefunden werden kann. Wir benötigen daher Funktionen der Form

$$H : \boldsymbol{K} \rightarrow [0 \cdots p-1]$$

welche die Menge $\boldsymbol{K}$ der möglichen Schlüssel abbilden auf den Indexbereich der Hash-Tabelle.Weil $\boldsymbol{K}$ i.a. wesentlich mehr als $p$ Elemente hat, müssen wir zwei Punkte besonders beachten:

- $H$ ist in der Regel keine injektive Funktion. Es sind daher Vorkehrungen notwendig für den Fall, daß zwei Schlüssel $k$ und $k'$ synonym sind, d.h. $H(k) = H(k')$.
- Um die Gefahr von Kollisionen gering zu halten, sollte $H$ die Schlüssel möglichst gleichmäßig auf die Indexmenge $[0 \cdots p-1]$ verteilen.

**Beispiel**

Wir nehmen an, daß die Hash-Tabelle $p = 31$ Elemente aufnehmen kann. Für die Schlüsselmenge

71, 107, 134, 162, 190, 344, 364, 495, 496, 640, 699, 769, 801, 833, 963,

und die Hash-Funktion $H(k) = k \bmod 31$ ergibt sich die folgende Belegung der Hash-Tabelle:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 496 | | 963 | 344 | 190 | | | 162 | | 71 | 134 | | | | 107 | |

| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 699 | | | 640 | | | 364 | | 769 | 801 | 833 | | | 495 |

Für diese spezielle Wahl der Schlüssel treten noch keine Kollisionen auf. Falls als nächstes jedoch der Schlüssel 552 eingeordnet werden soll, müßte er auf der selben Position wie der Schlüssel 769 eingetragen werden.

**Offene und geschlossene Hash-Verfahren**

Die Techniken zur Behandlung von Kollisionen führen zu einer Unterscheidung in offene und geschlossene Adressierungs-Strategien:

- Offene Adressierungsverfahren suchen im Fall einer Kollision eine Ersatzadresse innerhalb der Hash-Tabelle.
- geschlossene Adressierungsverfahren verwenden neben der Hash-Tabelle noch einen Überlaufbereich. Dort werden alle Sätze untergebracht, die wegen einer Kollision nicht direkt in der Hash-Tabelle abgelegt werden können. Eine Möglichkeit in diesem Fall ist es, "synonyme" Schlüssel als lineare Liste an den ersten Schlüssel mit dem selben Hash-Index anzuhängen.

**Umwandlung nichtnumerischer Schlüssel**

In der Praxis finden wir häufig nichtnumerische Zeichenfolgen als Schlüssel. Diese müssen wir zunächst in eine nichtnegative Zahl umwandeln.

- Eine direkte Interpretation der Zeichenfolge als Binärzahl ist ungünstig, weil die dabei entstehenden Zahlen unhandlich groß werden und in der Regel kein zusammenhängendes Intervall bilden. Der Ortsname "Wien" entspricht der ASCII-Zeichenfolge 57 69 65 6E . Dies würde bei direkter Interpretation zu einer Binärzahl mit dem dezimalen Wert 1.466.525.038 führen.
- Bildet man stattdessen die in den Schlüsseln erlaubten Alphabetzeichen bijektiv auf ein Intervall $[1 \cdots q]$ ab, also zum Beispiel durch eine Abbildung $\chi$ :

| $\chi$ : | A | B | C | ..... | Z | a | b | c | ..... | z |
|---|---|---|---|---|---|---|---|---|---|---|
| | ↓ | ↓ | ↓ | | ↓ | ↓ | ↓ | ↓ | | ↓ |
| | 1 | 2 | 3 | ····· | 26 | 27 | 28 | 29 | ..... | 52 |

so kann man einer Zeichenfolge $z_0 z_1 z_2 \cdots z_s$ die folgende Zahl zuordnen:

$$\sum_{i=0}^{ss} \chi(z_i)(q+1)^i$$

Dabei wird der Position $i$ im Schlüssel die Potenz $(q+1)^i$ und der Faktor $\chi(z_i)$ zugeordnet. Aufgrund zahlentheoretischer Überlegungungen weiß man, daß eine solche Zuordnung die Menge der Schlüssel bijektiv auf ein Intervall nichtnegativer ganzer Zahlen abbildet.

Der Schlüssel "Wien" wird nach dieser Vorschrift mit `q=53` wie folgt abgebildet auf die Zahl:

$$6.044.037 = 23*53^0 + 35*53^1 + 31*53^2 + 40*53^3$$

Umgekehrt entspricht der Zahl $4.878.100$ der Schlüssel "Genf":

$$4.878.074 = 7*53^0 + 31*53^1 + 40*53^2 + 32*53^3$$

## 7.2 Hash-Funktionen

Bei der Wahl einer Hash-Funktion sind zwei Kriterien wesentlich für die Qualität des Verfahrens:

- Die Hash-Funktion muß sich einfach berechnen lassen und
- sie muß die Schlüssel gleichmäßig über den Indexbereich der Hash-Tabelle verteilen, damit Kollisionen möglichst selten auftreten.

Genaugenommen wird eine Gleichverteilung der tatsächlich auftretenden Schlüssel benötigt. Darüber hinaus sollen die Adressen $H(k)$ auch dann noch gleichmäßig verteilt sein, wenn die Schlüssel selbst dies nicht sind. Da dies nur schwierig zu realisieren ist, begnügt man sich in der Regel mit einer Gleichverteilung aller möglichen Schlüssel.

### Beispiel

Die Vorliebe von Programmierern, Variablennamen in einem Programm systematisch zu wählen, z.B. `x1, x2, y1, y2, y3` führt zu einer Häufung von Schlüsseln. Die Hash-Funktion muß diese trotzdem gleichmäßig auf den Indexbereich der Symboltabelle verteilen.

### Das Divisionsrest-Verfahren

Eine einfache, aber in vielen Fällen brauchbare Methode liefert mit einer geeignet gewählten **Primzahl** $p$ die Abbildung

$$H : K \to [0 \cdots p-1]$$
$$k \to H(k) = k \bmod (p)$$

### Beispiel

Mit dem Primzahlwert $p$ = 59.791 liefert die Divisionsrest-Methode die folgenden Adressen:

| Schlüssel | numerischer Schlüsselwert | Hash-Adresse |
|---|---|---|
| Wien | 6.044.037 | 5.146 |
| Genf | 4.878.074 | 35.003 |

### Die Multiplikationsmethode

Eine alternative Methode besteht darin, den numerischen Schlüssel mit einer irrationalen Zahl $F$ zu multiplizieren und den ganzzahligen Anteil des Resultats zu subtrahieren. Als Ergebnis

erhält man Werte, die im Intervall [*0 .. 1*) gleichmäßig gestreut sind. Multiplikation mit der Tabellenlänge m ergibt dann einen Index im Bereich [*0 .. m*):

$$H : K \to [0 \cdots m)$$
$$k \to H(k) = \lfloor m * (F * k - \lfloor F * k \rfloor) \rfloor$$

**Beispiel**

Für den goldenen Schnitt, d.h. die Zahl $F = \frac{\sqrt{5}-1}{2} \approx 0.6180339887$ gilt $1 : F = F : (1-F)$.

Dafür kann man eine besonders gleichmäßige Verteilung nachweisen. Mit $m$=10 erhält man als Hash-Adresse der Zahlen 1, 2, 3, ..... , 10 genau eine Permutation der Adressen 0, 1, ..... , 9:

$$H(1) = \lfloor 10 * (F * 1 - \lfloor 0.618 \cdots * 1 \rfloor) \rfloor = \lfloor 10 * (0.618 \cdots * 1 - \lfloor 0.618 \cdots * 1 \rfloor) \rfloor = 6$$
$$H(2) = \lfloor 10 * (F * 2 - \lfloor 0.618 \cdots * 2 \rfloor) \rfloor = \lfloor 10 * (0.618 \cdots * 2 - \lfloor 0.618 \cdots * 2 \rfloor) \rfloor = 2$$
$$H(3) = \lfloor 10 * (F * 3 - \lfloor 0.618 \cdots * 3 \rfloor) \rfloor = \lfloor 10 * (0.618 \cdots * 3 - \lfloor 0.618 \cdots * 3 \rfloor) \rfloor = 8$$
$$\cdots\cdots\cdots\cdots$$

**Andere Schlüsseltransformationen**

In speziellen Fällen können die oben angegebenen Verfahren versagen: Wenn z.B. numerische Schlüssel den Abstand $p$ haben, dann bildet sie das Divisionsrest-Verfahren mit dem Divisor $p$ alle auf die selbe Hash-Adresse ab. Deshalb werden oft zusätzliche Schlüsseltransformationen vorgeschaltet, z.B.

- *Schlüssel-Faltung*
  Dabei wird der numerische Schlüsselwert in mindestens zwei Teile zerlegt. Die Teile werden dann addiert und auf das Ergebnis eine der obigen Hash-Funktionen angewandt. Wir erhalten z.B. für das Divisionsrestverfahren mit $p$ = 997 :

| Schlüssel | numerischer Schlüsselwert | gefalteter Schlüsselwert | Hash-Adresse |
|---|---|---|---|
| Wien | 6.044.037 | 604+4.037 = 4.641 | 653 |
| Genf | 4.878.074 | 487+8.074 = 8.561 | 585 |

- *Basis-Transformation*
  Dabei wird der dezimale Wert des Schlüssels in einem Zahlen system mit einer anderen Basis dargestellt. Die Zifferndarstellung bezüglich dieser neuen Basis wird wieder als Dezimalzahl aufgefaßt und mit einer der obigen Hash-Funktionen weiterverarbeitet. Für eine Transformation in das Oktal-System und das Divisionsrest-Verfahren mit p=997 erhalten wir z.B.:

| Schlüssel | numerischer Schlüsselwert | oktaler Schlüsselwert | Hash-Adresse |
|---|---|---|---|
| Wien | 6.044.037 | 27.034.605 | 950 |
| Genf | 4.878.074 | 22.467.372 | 974 |

# 7.3 Geschlossene Hash-Verfahren

Geschlossene Hash-Verfahren sehen für den Fall einer Adreßkollision in der Hash-Tabelle einen separaten Überlaufbereich vor. Dort werden Datensätze mit synonymen Hash-Adressen in linearen Listen verwaltet. Im Gegensatz zur Hash-Tabelle kann der Überlaufbereich dynamisch verwaltet werden. Wir betrachten hier eine geschlossene Hash-Methode, die sich besonders gut für den Zugriff auf externe Dateien eignet.

### Gestreuter Index (Scattered Index)

Diese, auch als direkte Verkettung bezeichnete Methode, nimmt in die Hash-Tabelle keine Datensätze, sondern nur Zeiger zu den Listen im Überlaufbereich auf. Ein Zeiger auf der Position $\alpha$ der Hash-Tabelle verweist auf die Liste aller Datensätze, deren Hash-Adresse den Wert $\alpha$ hat.

Die wesentlichen Vorteile dieser Methode sind folgende:

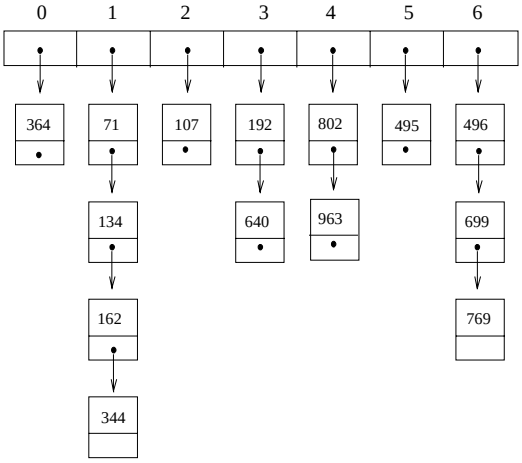
- Die Hash-Tabelle bietet bei relativ geringem Platzbedarf viele Hash-Adressen.
- Das Zugriffsverfahren kann die Hash-Tabelle resident im Arbeitsspeicher halten und die Datensätze bei Bedarf vom Externspeicher laden.
- Datensätze können gelöscht werden, wobei ihr Platz freigegeben werden kann.
- Alle Datensätze werden gleich behandelt, weil nicht unterschieden werden muß, ob ein Satz in der Hash-Tabelle oder im Überlaufbereich steht.

### Beispiel

Wir betrachten die Einordnung der Schlüsselmenge

{ 71, 107, 134, 162, 192, 344, 364, 495, 496, 640, 699, 769, 802, 963 }

nach dem Divisionsrest-Verfahren mit p=7 nach der Methode des gestreuten Index und erhalten:




### Zugriffe bei gestreutem Index

*Suchen*: Zunächst wird mit der Hash-Funktion der Anker der zugehörigen Überlaufliste ermittelt. Danach wird der Satz linear in der Überlaufliste gesucht.

*Einfügen*: Zunächst wird nach dem einzufügenden Schlüssel gesucht. Wenn die Suche erfolglos am Ende der zugehörigen Überlaufliste abgebrochen wird, wird der neue Satz dort angehängt.

*Löschen*: Zunächst wird der zu löschende Satz gesucht. Falls er vorhanden ist, wird er aus der Überlaufliste entfernt und gelöscht.

### Analyse

Die worst case-Effizienz aller Hash-Verfahren ist sehr schlecht, denn man kann immer einen Fall konstruieren, bei dem alle belegten Positionen inspiziert werden müssen, bevor ein Schlüssel gefunden wird oder eine Fehlanzeige erstattet werden kann. Da aber der günstigste Fall immer mit einem Zugriff auskommt, ist nur die Bestimmung des mittleren Aufwands interessant.

Die Effizienz der gestreuten Speicherung hängt hauptsächlich vom mittleren Suchaufwand ab. Diesen schätzen wir nun ab unter den folgenden Annahmen:

- Die Hash-Tabelle habe die Größe $p$.
- Es seinen bereits $n$ Datensätze gespeichert. Der *Füllungsgrad* der Tabelle beträgt dann $\alpha = n / p$ .
- Die Hash-Funktion liefere alle Hash-Adressen mit gleicher Wahrscheinlichkeit und unabhängig voneinander.

Bei einer *erfolglosen Suche* müssen wir eine Überlauf-Liste bis ans Ende durchsuchen. Da die Listen im Mittel die Länge $\alpha = n / p$ haben, sind für die erfolglose Suche stets $\alpha$ Vergleiche notwendig, d.h.

$$C_n' = \alpha$$

Um den mittleren Aufwand bei *erfolgreicher Suche* zu bestimmen, betrachten wir zunächst die Zahl der Vergleiche, die zum Einfügen des $j$-ten Schlüssels notwendig waren für $1 \le j \le n$ :

Da zu diesem Zeitpunkt die durchschnittliche Listenlänge $\frac{j-1}{p}$ Elemente betrug, müssen wir $1 + \frac{j-1}{p}$ Elemente inspizieren, um das $j$-te Elements zu finden. Der mittlere Aufwand zum Auffinden eines beliebigen, vorhandenen Elementes beträgt daher

$$C_n = \frac{1}{n} * \sum_{j=1}^{n} \left( 1 + \frac{j-1}{p} \right) = 1 + \frac{1}{np} * \frac{n(n-1)}{2} \approx 1 + \frac{\alpha}{2}$$

Die folgende Tabelle zeigt numerische Werte für $C_n$ und $C_n'$ in Abhängigkeit vom Belegungsgrad $\alpha$ der Tabelle:

| Belegungsgrad $\alpha$ | Vergleiche zum Einfügen neuer Elemente ($C_n'$) | Vergleiche zum Suchen vorhandener Elemente ($C_n$) |
|---|---|---|
| 0,50 | 0,50 | 1,250 |
| 0,90 | 0,90 | 1,450 |
| 0,95 | 0,95 | 1,475 |
| 1,00 | 1,00 | 1,500 |

Ein Vergleich der Werte $C_n$ und $C_n'$ zeigt, daß für einen Belegungsfaktor $\alpha \le 1$ das erfolglose Suchen und damit auch das Einfügen neuer Elemente schneller ist als die Suche nach vorhandenen Elementen. Eine zusätzliche Effizienzsteigerung für diesen Fall kann man durch sortiertes Einfügen in die Überlauflisten erreichen.

Ist für eine Anwendung das Aufsuchen vorhandener Sätze wichtiger, dann kann man den Aufwand dafür vermindern, indem man die Überlauflisten als selbstanordnende Listen organisiert, in denen häufig benutzte Elemente an den Anfang gestellt werden.

# 7.4 Offene Hash-Verfahren

Offene Hash-Verfahren lösen das Kollisionsproblem dadurch, daß sie im Fall einer Kollision eine Folge von Alternativ-Positionen (Sondierungsfolge) innerhalb der Hash-Tabelle ermitteln und den neuen Satz an der ersten freien Position dieser Sondierungsfolge abspeichern. Es wird daher zusätzlich zur Hash-Tabelle kein eigener Überlaufbereich benötigt.

### Beispiel

Wir betrachten die am Anfang des Kapitels dargestellte Situation einer Hash-Tabelle und benutzen die gleiche Hash-Funktion wie dort:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 496 | | 963 | 344 | 190 | | | 162 | | 71 | 134 | | | | 107 | |

| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 699 | | | 640 | | | 364 | | 769 | 801 | 833 | | | 495 |

Wird als nächstes der Schlüssel 552 einfügt, kommt es zu einer Kollision mit dem Schlüssel 769 auf Position 25. Wir inspizieren als Sondierungsfolge nun die nachfolgenden Tabellenpositionen solange bis eine freie Position gefunden wird. Auf dieser tragen wir den neuen Schlüssel ein und erhalten:

| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 699 | | | 640 | | | 364 | | 769 | 801 | 833 | 522 | | 495 |

### Zugriffe bei offenen Hash-Verfahren

*Suchen*: Zunächst bestimmen wir die Hash-Adresse des gesuchten Schlüssels. Falls er auf dieser Position nicht gefunden wird, inspizieren wir der Reihe nach die Positionen der Sondierungsfolge solange, bis entweder der Schlüssel gefunden ist oder eine leere Tabellenposition angetroffen wird.

*Einfügen*: Der einzufügende Schlüssel wird zunächst gesucht. Falls die Suche erfolglos auf einem freien Platz endet, kann er dort eingetragen werden.

*Löschen*: Ein zu löschender Schlüssel kann zunächst nur markiert werden. Für nachfolgende Suchoperationen wird seine Tabellenposition behandelt wie eine mit einem anderen Schlüssel belegte. Bei Einfüge-Operationen wird seine Tabellenposition wie eine freie behandelt.

Wesentliche Eigenschaften offener Hash-Verfahren sind die folgenden:

- Der Datenbereich muß vorab fest reserviert werden. Nachträgliche Erweiterungen erfordern eine Reorganisation der Hash-Tabelle.
- Für Zugriffsverfahren zu externen Dateien eignen sich offene Hash-Verfahren weniger gut als geschlossene, weil sie keine Aufspaltung in einen Index und die Datenbereiche gestatten.
- Die Qualität offener Hash-Verfahren hängt wesentlich von dem Algorithmus zur Bestimmung der Sondierungsfolge ab.

Wir betrachten nun verschiedene Sondierungsverfahren:

## 7.4.1 Lineares Sondieren

Beim Linearen Sondieren in einer Hash-Tabelle der Länge $p$ benutzen wir, ausgehend von der originalen Hashfunktion $H = H_0$ eine Folge weiterer Hash-Funktionen

$$H = H_0, H_1, \cdots, H_{p-1}$$

mit $$H_i(k) = (H_0(k) + a*i + b) \bmod(p)$$

wobei gilt: $0 < a < p$ und $0 \le b$.

Wegen $$H_i(k) - H_j(k) = \big(a*(i-j)\big) \bmod(p) \neq 0 \quad \text{für} \quad i \neq j$$

sind die Werte $H_i(k)$ alle verschieden, d.h. die Sondierungsfolge $H = H_0, H_1, \cdots, H_{p-1}$ durchläuft alle Positionen der Hash-Tabelle.

Für $a = 1$ und $b = 0$ erhalten wir die im vorigen Beispiel bereits benutzte Strategie, welche die nachfolgenden Hash-Adressen der Reihe nach durchläuft.

Das Lineare Sondieren ist zwar sehr einfach, es hat aber auch Nachteile:

Wenn zwei Schlüssel $k_1$ und $k_2$ synonym sind, dann ergibt sich für sie die gleiche Sondierungsfolge (sekundäre Häufung). Außerdem verlaufen auch Sondierungsfolgen nicht synonymer Schlüssel synchron, sobald sie aufeinandertreffen (primäre Häufung). Dies führt dazu daß große, voll belegte Bereiche tendenziell stärker anwachsen als kleine, wodurch sich in der Tabelle belegte Cluster bilden. Dieser Effekt verstärkt sich, wenn die der Belegungsfaktor gegen 1 tendiert..

#### Effizienz

Eine Analyse der notwendigen Vergleiche bei erfolgloser bzw. erfolgreicher Suche ergibt:

$$C_n' = \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) \quad \text{und} \quad C_n = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

Die folgende Tabelle zeigt numerische Werte für $C_n$ und $C_n'$ in Abhängigkeit vom Belegungsgrad $\alpha$ der Tabelle:

| Belegungsgrad $\alpha$ | Vergleiche zum Einfügen neuer Elemente ($C_n'$) | Vergleiche zum Suchen vorhandener Elemente ($C_n$) |
|---|---|---|
| 0,50 | 2,5 | 1,5 |
| 0,90 | 50,5 | 5,5 |
| 0,95 | 200,5 | 10,5 |

## 7.4.2 Quadratisches Sondieren

Durch die Wahl anderer Sondierungsfunktionen kann man die beim Linearen Sondieren auftretenden primären Häufungen vermeiden. Beim Quadratischen Sondieren sucht man deshalb mit quadratisch wachsendem Abstand und wechselnder Richtung nach dem nächsten freien Platz. Ausgehend von der originalen Hashfunktion $H = H_0$ bilden wir eine Folge weiterer Hash-Funktionen

$$H = H_0, H_1, \cdots, H_{p-1}$$

mit $$H_i(k) = \left( H_0(k) + \left\lceil \frac{i}{2} \right\rceil^2 * (-1)^i \right) \bmod(p)$$

Damit wird erreicht, daß aufeinandertreffende Sondierungsfolgen, d.h. wenn $H_i(k_1) = H_j(k_2)$ ist, sich wieder trennen, falls j≠i ist. Sondierungsfolgen synonymer Schlüssel verlaufen weiterhin synchron, d.h. sekundäre Häufungen kommen auch beim quadratischen Sondieren vor.

### Effizienz

Eine Analyse der notwendigen Vergleiche bei erfolgloser bzw. erfolgreicher Suche ergibt:

für erfolgreiches Suchen: $$C_n \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$$

für erfolgloses Suchen: $$C_n' \approx 1 + \frac{\alpha^2}{1-\alpha} + \ln\left(\frac{1}{1-\alpha}\right)$$

Die folgende Tabelle zeigt numerische Werte für $C_n$ und $C_n'$ in Abhängigkeit vom Belegungsgrad $\alpha$ der Tabelle:

| Belegungsgrad α | Vergleiche zum Einfügen neuer Elemente ( $C_n'$ ) | Vergleiche zum Suchen vorhandener Elemente ( $C_n$ ) |
|---|---|---|
| 0,50 | 2,19 | 1,44 |
| 0,90 | 11,40 | 2,85 |
| 0,95 | 22,05 | 3,52 |

Verglichen mit dem Linearen Sondieren wird also eine wesentliche Verbesserung erreicht, vor allem dann, wenn der Belegungsgrad gegen 1 tendiert.

Weitere Verbesserungen lassen sich mit Sondierungsfunktionen erreichen, bei denen die Schrittweite auch vom Schlüssel selbst abhängt. Solche Verfahren sind z.B.

- Sondieren mit einer Pseudozufallsfolge
- Sondieren mit Doppel-Hash-Verfahren (Brent'sche Methode)
- Binärbaum-Sondieren
- Robin Hood-Hashing

Eine weitere Methode, die wir im folgenden noch besprechen werden, greift auf eine Verkettung der Überläufer zurück, die wir schon bei der Methode des Hashing mit gestreutem Index kennengelernt haben.

## 7.4.3 Coalesced Hashing

Vergleicht man alle bisher besprochenen Hashing-Verfahren, so ist festzustellen, daß das geschlossene Hashing-Verfahren mit gestreutem Index auch bei einem Belegungsgrad von 95% die beste Effizienz aufweist mit einer mittleren Einfügezeit von 2 bzw 2,5 Zugriffen zum Einfügen bzw. Suchen vorhandener Elemente. Der Preis dafür ist der zusätzliche Aufwand für die Verkettung der Überläufer und der zusätzliche Platz für die Datensätze, der auch dann gebraucht wird, wenn der Hashing-Index noch relativ leer ist.

Die Methode des Coalesced Hashing verbindet die Platzvorteile des offenen Hashing mit den Zeitvorteilen der Überläufer-Verkettung. Dabei gehen wir wie folgt vor:

### Methode

Die Elemente der Hash-Tabelle bestehen aus Datensätzen mit einem Schlüssel und einem Kettungszeiger.

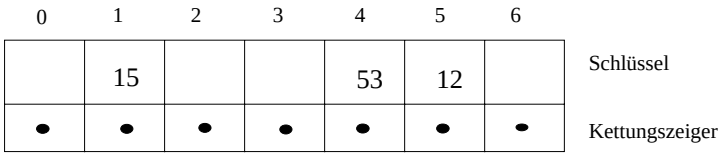
*Suchen*: Ein Schlüssel wird gesucht, indem man zunächst seine Hash-Adresse ermittelt und die dort beginnende lineare Liste solange durchläuft, bis man ihn gefunden hat oder das Ende der Überlauf-Liste erreicht wird.

*Einfügen*: Ein Satz wird eingefügt, indem man zunächst nach seinem Schlüssel sucht. Wird dabei das Ende der Überlauf-Liste erreicht, ohne ihn zu finden, so reserviert man dasjenige freie Element der Hash-Tabelle mit der größten Adresse, trägt den neuen Satz dort ein und hängt ihn an das Ende der Überlauf-Liste an.

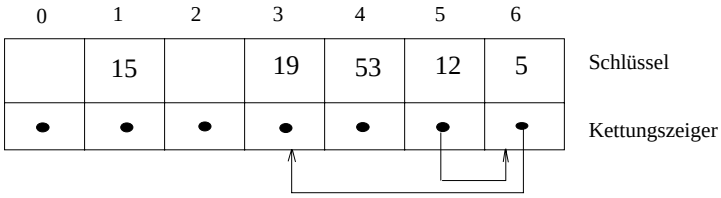
*Löschen*: Der Schlüssel des zu löschenden Satzes wird gesucht, das Element der Hash-Tabelle aus der Überlaufliste ausgehängt und freigegeben.

### Beispiel

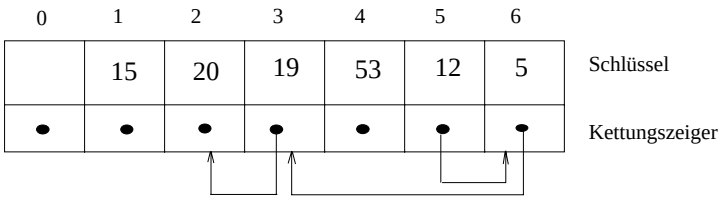
Wir tragen zunächst die Schlüssel 53, 15 und 12 in eine Hashtabelle der Größe p=7 nach dem Divisionsrestverfahren ein und erhalten:




Wenn wir anschließend die Schlüssel 5 und 19 eintragen, dann müssen wir beide in die bei der Hash-Adresse 5 beginnende Überlaufkette einhängen und erhalten:




Wenn wir nun den Schlüssel 20 eintragen wollen, dann ist der Platz 6, der die Hash-Adresse von 20 darstellt, bereits belegt mit dem Schlüssel 5. Der neue Schlüssel wird daher ans Ende der auf Position 6 angetroffenen Überlaufkette angehängt und wir erhalten:



Die Überlauf-Liste enthält daher im Gegensatz zum Hashing mit gestreutem Index nicht nur synonyme Schlüssel. Die Überlaufketten mit den Hash-Adressen 5 und 6 sind verschmolzen (coalesced). Dies beeinflußt zwar nicht die Korrektheit des Verfahrens. Es ist aber geringfügig weniger effizient als das gestreute Hashing, weil sich längere Überlaufketten ergeben.

### Effizienz

Eine Analyse der notwendigen Vergleiche bei erfolgloser bzw. erfolgreicher Suche ergibt:

für erfolgreiches Suchen: $C_n \approx 1 + \frac{1}{8\alpha}\left(e^{2\alpha} - 1 - 2\alpha\right) + \frac{\alpha}{4}$

für erfolgloses Suchen: $C_n \approx 1 + \frac{1}{4}\left(e^{2\alpha} - 1 - 2\alpha\right)$

Die folgende Tabelle zeigt numerische Werte für $C_n$ und $C_n'$ in Abhängigkeit vom Belegungsgrad $\alpha$ der Tabelle:

| Belegungsgrad $\alpha$ | Vergleiche zum Einfügen neuer Elemente ($C_n'$) | Vergleiche zum Suchen vorhandener Elemente ($C_n$) |
|---|---|---|
| 0,50 | 1,18 | 1,30 |
| 0,90 | 1,81 | 1,68 |
| 0,95 | 1,95 | 1,74 |
| 1,00 | 2,10 | 1,80 |

# 8 Datenkompression

Bisher haben wir die Optimierung von Algorithmen hauptsächlich unter dem Gesichtspunkt der Minimierung von Rechenzeiten betrachtet. Häufig wurde dabei eine kürzere Laufzeit mit einem erhöhten Platzbedarf erkauft.

In diesem Kapitel ist unser primäres Ziel die Verringerung des Datenvolumens und dies wird mit dem Zeitaufwand für die Kompression und Dekompression bezahlt. Die wesentlichen Motive für die Datenkompression sind die folgenden:

- Speicherplatz ist immer noch ein entscheidender Kostenfaktor in der Datenverarbeitung. Die Speicheranforderungen heutiger Software sind vor allem durch die Integration von Bild-, Ton- und Video-Daten wesentlich stärker gewachsen als die Kosten pro Byte gesunken sind.
- Mit der Einbindung der meisten Rechner in lokale und weltweite Netze wird auch die Bandbreite, die für die Übertragung von Daten zur Verfügung steht, zunehmend zum Engpaß. Die Kompression der übertragenen Daten spart dabei in erster Linie Bandbreite, aber auch wieder Zeit.

Die mit Kompressionstechniken erzielbaren Einsparungen beim Datenvolumen hängen sehr stark von den Daten selbst ab. Typische Werte sind:
- 20% bis 50% für Texte
- 50% bis 90% für Binärdaten.

Man beachte, daß es zu jedem Kompressionsverfahren auch Datenpakete geben muß, die es aufbläht. Andernfalls könnte man jedes Datenpaket durch iterierte Kompression beliebig verkleinern.

Wir betrachten im folgenden Kapitel zwei grundlegende Ansätze für Kompressionsverfahren:

- Mit der Lauflängen-Codierung lernen wir ein einfaches und schnelles Verfahren kennen, das aber nur für geeignetes Datenmaterial brauchbar ist.
- Mit der Huffman-Codierung sehen wir eine optimale Methode, die die Basis vieler in der Praxis benutzter Kompressions-Utilities ist.

## 8.1 Lauflängen-Codes

Wir gehen davon aus, daß die zu komprimierenden Datenpakete beliebige Folgen von Zeichen sind, wobei jedes Zeichen einen gleich großen Speicherplatz belegt, z.B. ein Byte. Als Zeichenwert soll jede mögliche Bitfolge vorkommen können.

Die Idee der Lauflängen-Codierung beruht darauf, längere Abschnitte aus aufeinanderfolgenden identischen Zeichen („Runs“) durch die Verwendung von Wiederholungsfaktoren zu verkürzen. Statt

| A | A | A | A | B | B | B | B | B | C | C | C | C | C | C | D | D | D | D | D | D | D | A | A | A | A | A | A | A | A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

schreiben wir dann:

| 4 | A | 5 | B | 6 | C | 7 | D | 8 | A |
|---|---|---|---|---|---|---|---|---|---|

Pro Run werden also zwei Speicherplätze benötigt. Mit der Methode erreichen wir für dieses Beispiel eine Reduktion des Speicherplatzbedarfs von urprünglich 30 Zeichen auf 10 Zeichen.

Um diesen Ansatz in der Praxis verwenden zu können sind aber noch einige Probleme zu beachten:

– *Effekt kurzer Runs*
Die Wiederholungsfaktoren sind nur aufgrund ihrer Position von den eigentlichen Zeichen unterscheidbar. Wir müssen daher auch Einzelzeichen mit dem Wiederholungsfaktor 1 codieren. Normale Texte werden dadurch stark aufgebläht, wie das folgende Beispiel demonstriert. Der 10 Zeichen umfassende String

| R | E | G | E | N | S | B | U | R | G |
|---|---|---|---|---|---|---|---|---|---|

wird codiert als

| 1 | R | 1 | E | 1 | G | 1 | E | 1 | N | 1 | S | 1 | B | 1 | U | 1 | R | 1 | G |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Der Platzbedarf wird in diesem Beispiel genau verdoppelt. Generell sollten wir daher kurze Runs nicht codieren. Für die Codierung normaler Text-Information ist das Verfahren kaum brauchbar, da sie längere Runs hauptsächlich bei Leerzeichen-Ketten aufweisen. Für die Codierung von Bildern ist die Methode aber vielfach sehr nützlich.

– *Verwendung von Escape-Zeichen*
Wenn wir kurze Runs uncodiert stehen lassen, sind Wiederholungsfaktoren nicht mehr durch ihre Stellung erkennbar. Wir müssen daher Wiederholungsfaktoren explizit markieren, z.B., indem wir ihnen ein sog. Escape-Zeichen voranstellen. Eine Codierungsvorschrift könnte dann wie folgt aussehen:

- Runs mit mindestens vier identischen Zeichen codieren wir durch eine Escape-Sequenz der Form

| [Escape-Zeichen] | [Wiederholungsfaktor] | [Zeichenwert] |
|---|---|---|

- Runs mit drei oder weniger identischen Zeichen übernehmen wir uncodiert.
- Das Escape-Zeichen selbst codieren wir durch die Sequenz

| [Escape-Zeichen] | 0 |
|---|---|

*Beispiel:*

Die Zeichenfolge

| A | A | A | A | B | B | B | C | C | C | C | C | D | E | E | E | E | E | E | F | F | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

wird mit „$“ als Escape-Zeichen codiert durch

| $ | 4 | A | $ | 3 | B | $ | 5 | C | $ | D | $ | 6 | E | F | F | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

– *Aufteilung langer Runs*
Wenn die Länge eines Run einen Wiederholungsfaktor ergibt, dessen numerischer Wert nicht mehr in eine Speicherzelle paßt, muß er in mehrere einzelne Runs zerlegt werden.

Beachtet man die oben aufgeführten Punkte, wird durch eine Lauflängen-Codierung jeder Zeichenblock zumindest nicht aufgebläht, solange er keine Escape-Zeichen enthält. Das Escape-Zeichen sollte daher ein seltenes Zeichen sein. Bereits komprimierte Datenblöcke werden in der Regel aber vergrößert, da sie dieser Forderung nicht genügen können.

# 8.2 Codes mit variabler Länge

## 8.2.1 Methodische Grundlagen

Die Verfahren dieser Gruppe versuchen, den Speicherplatzbedarf dadurch zu reduzieren, daß sie sich nicht auf eine feste Zahl von Bits für die Codierung einzelner Zeichen beschränken. Dabei wird die Codierung aber von dem zu codierenden Datenblock abhängig, so daß man die Decodierungsvorschrift den codierten Daten hinzufügen muß. Sie können daher erst dann eine positive Bilanz aufweisen, wenn derKompressionsgewinn den Platzbedarf der Decodierungsvorschrift übersteigt.

### Beschränkung auf die tatsächlich vorkommenden Zeichenmenge

Dabei wird weiterhin jedes Zeichen mit einer festen Anzahl von Bits codiert, aber man codiert nur diejenigen Zeichen, die tatsächlich in dem Datenblock vorkommen. Das folgende Beispiel demonstriert diese Methode:

Die Zeichenfolge

| S | I | M | S | A | L | A | B | I | M |
|---|---|---|---|---|---|---|---|---|---|

belegt als normaler Textstring 8 Bit pro Zeichen. Eine Codierung, die nur die tasächlich vorkommenden Zeichen berücksichtigt, kommt aber mit drei Bit pro Zeichen aus. Sie könnte wie folgt aussehen

Codierungsvorschrift:

| A | B | I | L | M | S |
|---|---|---|---|---|---|
| 000 | 001 | 010 | 011 | 100 | 101 |

Die Anwendung dieser Codierungsvorschrift auf den String liefert dann:

| S | I | M | S | A | L | A | B | I | M |
|---|---|---|---|---|---|---|---|---|---|
| 101 | 010 | 100 | 101 | 000 | 011 | 000 | 001 | 010 | 100 |

Statt der 80 =10 x 8 Bit für den Originalstring benötigen wir nur 30 = 10 x 3 bit für den codierten String.Genaugenommen müßten wir aber den Platzbedarf der Code-Tabelle hinzurechnen.

### Berücksichtigung von Zeichenhäufigkeiten

Eine weitere Verbesserung dieses Ansatzes erreichen wir, wenn wir die tatsächlichen Häufigkeiten der im Datenblock vorkommenden Zeichen bei der Codierung berücksichtigen. Dann können wir nicht nur alle Zeichen vernachlässigen, die gar nicht vorkommen, sondern häufige Zeichen mit kürzeren Codes darstellen als seltene. Das folgende Beispiel demonstriert demonstriert das Prinzip dieser Methode und ihre Probleme. Wir legen zunächst willkürlich die folgende Codierungsvorschrift fest:

Codierungsvorschrift:

| A | B | I | L | M | S |
|---|---|---|---|---|---|
| 1 | 0 | 00 | 01 | 10 | 11 |

Die Anwendung dieser Codierungsvorschrift auf den String SIMSALABIM liefert dann eine Codierung mit 17 Bit:

| S | I | M | S | A | L | A | B | I | M |
|---|---|---|---|---|---|---|---|---|---|
| 11 | 00 | 10 | 11 | 1 | 01 | 1 | 0 | 00 | 10 |

Bei dieser Codierungsvorschrift läßt sich die ursprüngliche Zeichenfolge nicht mehr rekonstruieren, denn die gleiche Bitfolge könnte z.B. auch zu dem String gehören.

| 1 | 10 | 0 | 1 | 01 | 1 | 10 | 11 | 00 | 01 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | M | B | A | L | A | M | S | I | L | A |

Das hier erkennbare Problem wird dadurch verursacht, daß die Codierung für manche Zeichen auch Anfang der Codierung anderer Zeichen ist, z.B. ist

| A = | 1 |
|---|---|
| M = | 10 |

Wenn wir also eine **1** gelesen haben, dann ist unklar, ob wir diese bereits zu **A** decodieren sollen, oder zusammen mit einer darauuffolgenden **0** zu **M.** Wir können dieses Problem vermeiden, wenn wir einen Code finden, bei dem kein Codewort Anfang eines anderen ist, z.B. den folgenden

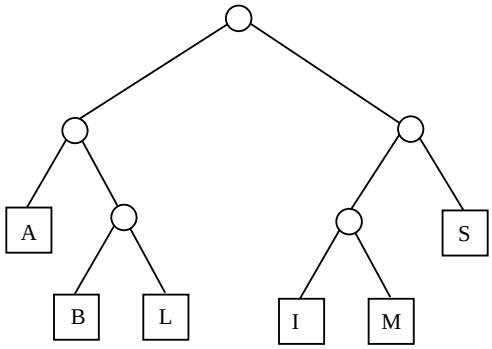
Codierungsvorschrift:

| A | B | I | L | M | S |
|---|---|---|---|---|---|
| 00 | 010 | 100 | 011 | 101 | 11 |

dann erhalten wir die folgende Codierung für SIMSALABIM, die 25 Bit benötigt. Das Original läßt sich jetzt daraus wieder eindeutig rekonstruieren:

| S | I | M | S | A | L | A | B | I | M |
|---|---|---|---|---|---|---|---|---|---|
| 11 | 100 | 101 | 11 | 00 | 011 | 00 | 010 | 100 | 101 |

Ein Verfahren zur Gewinnung einer solchen Codierungsvorschrift können wir mit Hilfe von Blatt-Suchbäumen (Tries) angeben. Die Blätter enthalten dabei die zu codierenden Zeichen und die Codierung eines Zeichens ist die Beschreibung des Weges von der Wurzel zu ihm: wenn wir in einem Baumknoten nach links gehen, dann notieren wir eine 0, wenn wir nach rechts gehen eine 1. Damit erhält jedes Zeichen ein eindeutiges Codewort und kein Codewort ist Anfang eines anderen. Aus dem folgenden Suchbaum ist der oben benutzten Code abgeleitet:




Die im folgenden Abschnitt betrachtete Huffman-Codierung realisiert diese Idee.

## 8.2.2 Die Huffman-Codierung

Die Huffman-Codierung stellt ein Verfahren dar, das zu einem Datenblock einen Suchbaum liefert, aus dem ein optimaler Code mit variabel vielen Bits pro Codewort abgeleitet werden kann. Wir demonstieren zunächst die Methode an einem konkreten Beispiel:

Als Datenblock zur Konstruktion des Code verwenden wir die Zeichenfolge:

DONAUDAMPFSCHIFFAHRTSGESELLSCHAFTSKAPITAEN

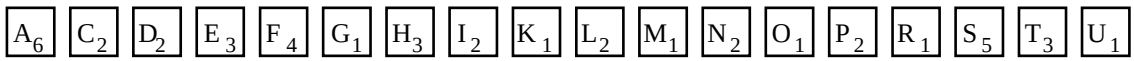
**1: Schritt:**

In einem ersten Verfahrensschritt zählen wird die Häufigkeiten der einzelnen Buchstaben und erhalten:

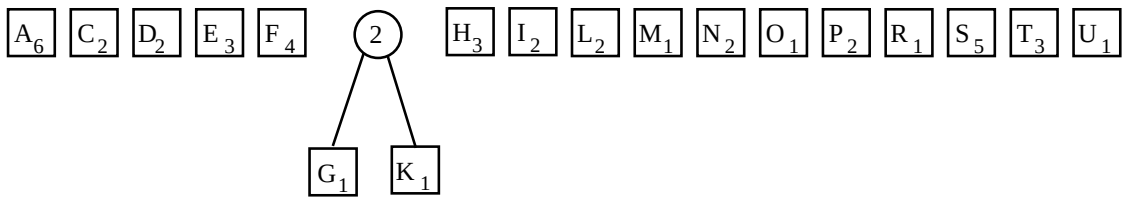
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 0 | 2 | 2 | 3 | 4 | 1 | 3 | 2 | 0 | 1 | 2 | 1 | 2 | 1 | 2 | 0 | 1 | 5 | 3 | 1 | 0 | 0 | 0 | 0 | 0 |

**2: Schritt:**

Die nächsten Schritte dienen dem Aufbau des Suchbaums, aus dem der Code abzuleiten ist. Zum Aufbau des Baums bilden wir zunächst aus jedem überhaupt vorkommenden Zeichen einen Blattknoten. In diesem vermerken wir zusätzlich zum entsprechenden Zeichen seine Häufigkeit. Diese bezeichnen wir als sein Gewicht. Jeder solche Blattknoten stellt einen Teilbaum mit der Höhe 1 dar.

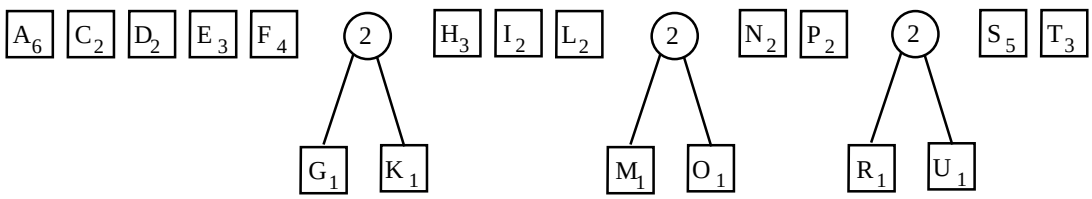



Unter allen Blättern wählen wir nun zwei mit minimalem Gewicht aus und bilden daraus einen binären Baum, der aus einer Wurzel und diesen beiden Blättern besteht. Als Gewicht der Wurzel verwenden wir die Summe der beiden Blattgewichte. Die Anzahl der Teilbäume nimmt dadurch um 1 ab.

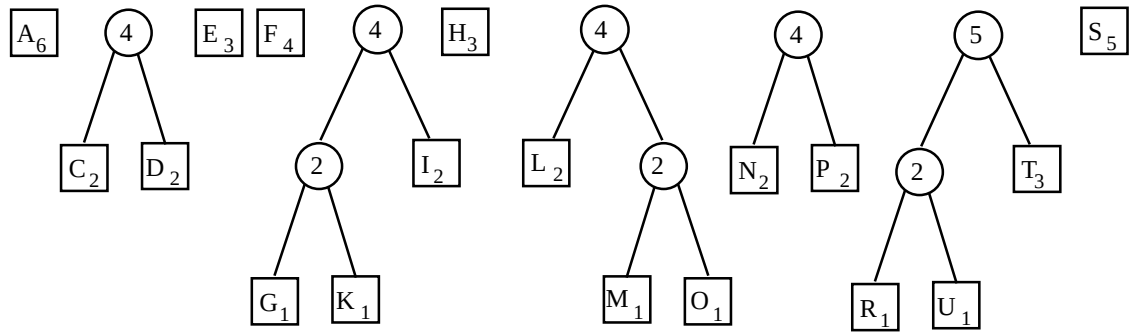



weitere Schritte:

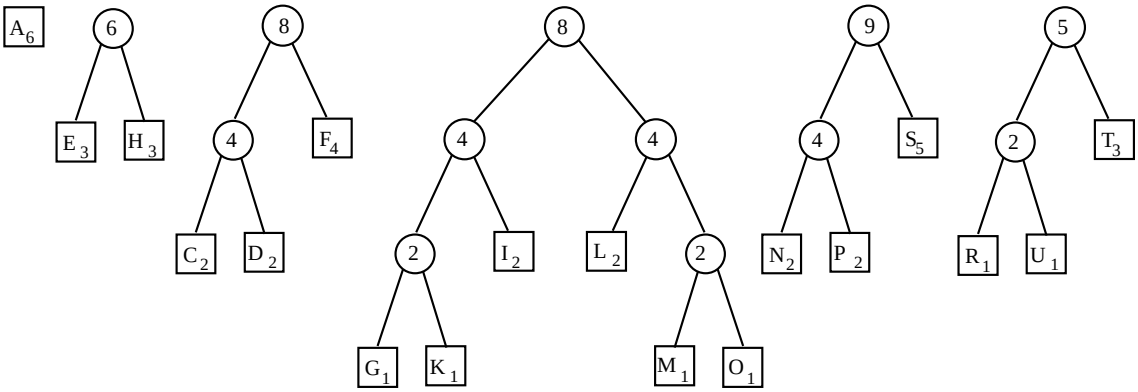
Wir fahren mit dem Ersetzen von Blättern/Teilbäumen solange fort, bis wir einen einziegen Baum erhalten. Die folgenden Abbildungen zeigen einzelne Stationen aus diesem Ablauf. Zunächst fassen wir nur Blätter zusammen:



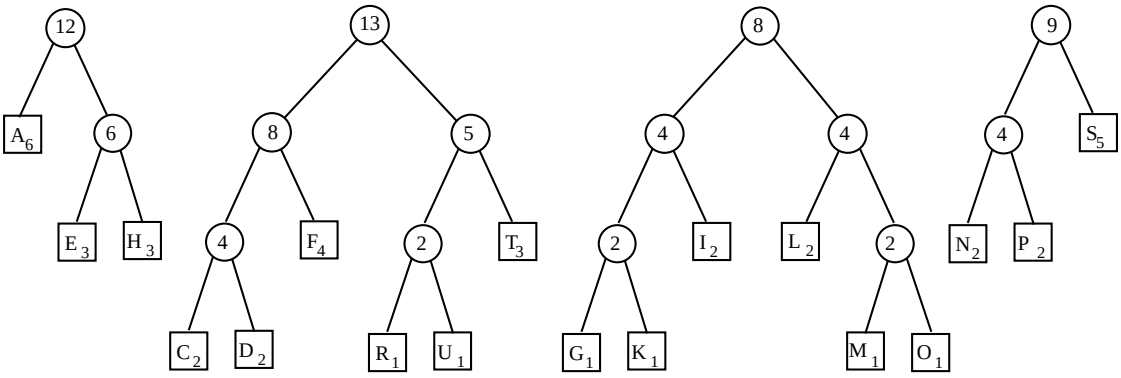
Im nächsten Schritt werden schon Teilbäume einbezogen, da sie ein geringeres Gewicht haben als einige Blätter:



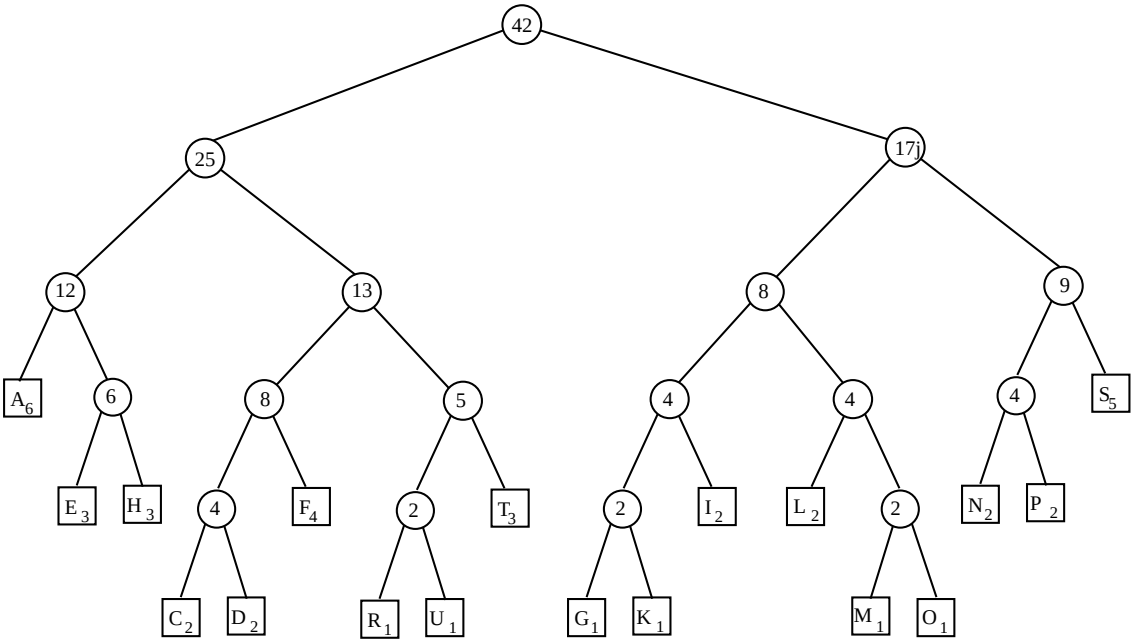

Mit weiteren Schritten erhalten wir:




Mit zwei weiteren Zusammenfassungen wird daraus:



Mit drei weiteren Zusammenfassungen erhalten wir daraus schließlich den endgültigen Suchbaum:




Damit können wir die Codierungsvorschrift ableiten, die die fol´gende Tabelle zeigt:

| A | C | D | E | F | G | H | I | K | L | M | N | O | P | R | S | T | U |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 000 | 10000 | 10001 | 0010 | 1001 | 01000 | 0011 | 0101 | 01001 | 0110 | 01110 | 1100 | 01111 | 1101 | 10100 | 111 | 1011 | 10101 |

Die Codierung des String DONAUDAMPFSCHIFFAHRTSGESELLSCHAFTSKAPITAEN ist damit:

| D | O | N | A | U | D | A | M | P | F | S | C | H | I | F | F | A | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10001 | 01111 | 1100 | 000 | 10101 | 10001 | 000 | 01110 | 1101 | 1001 | 111 | 10000 | 0011 | 0101 | 1001 | 1001 | 000 | 0011 |

| R | T | S | G | E | S | E | L | L | S | C | H | A | F | T | S | K | A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10100 | 1011 | 111 | 01000 | 0010 | 111 | 0010 | 0110 | 0110 | 111 | 10000 | 0011 | 000 | 1001 | 1011 | 111 | 01001 | 000 |

| P | I | T | A | E | N |
|---|---|---|---|---|---|
| 1101 | 0101 | 1011 | 000 | 0010 | 01111 |

### Folgerungen aus der Methode

- Jedes Zusammenfassen von Teilbäumen verringert die Gesamtzahl von Teilbäumen um 1.
- Die Blätter mit dem geringsten Gewicht landen auf den untersten Ebenen des resultierenden Baumes, so daß ihnen die längsten Codeworte zugeordnet werden.
- Ein so konstruierter Code ist nicht eindeutig, da bei der Wahl der zusammenzufassenden Teilbäume eine gewisse Freiheit besteht. Er ist aber in jedem Fall optimal zur Codierung des zugrundeliegenden Datenpakets im Sinn der folgenden beiden Bemerkungen:

*Bemerkung 1*
Die Codierung des Datenpakets erfordert genausoviele Bits, wie die „gewichtete äußere Pfadsumme“ des Baumes beträgt. Diese ist definiert durch:

$$\sum_{\text{alle Blätter}} [\text{Weglänge zum Blatt}][\text{Gewicht des Blatts}]$$

*Bemerkung 2*
Kein anderer Suchbaum mit den gleichen Blattgewichten hat eine geringere äußere Pfadsumme.

Die erste Bemerkung bedarf keiner besonderen Begründung, denn die Gewichte geben die Häufigkeit an, mit der die Zeichen auftreten und die Weglängen sind die Längen der Zeichen-Codes.

Die zweite Bemerkung läßt sich induktiv beweisen. Sie ist aber plausibel, denn man kann jeden anderen Suchbaum nach der obigen Methode konstruieren. Wenn dabei aber nicht stets Teilbäume mit minimalem Gewicht zusammengefaßt werden, dann geraten Blätter mit größerem Gewicht auf tiefere Baum-Ebenen, so daß zwangsläufig die äußere Pfadsumme stärker als nötig steigt.

### Hinweise zur Implementierung der Huffman-Codierung

Bei der Konstruktion des Blattsuchbaums müssen wir immer wieder kleineste Elemente aus einer Menge entfernen. Dazu eignet sich die bekannte Methode zum Aufbau eines Minimum-Heap:

- Zunächst bilden wir aus den Blättern einen Minimum-Heap aufgrund der Blattgewichte.
- Um zwei Teilbäume mit minimalem Gewicht zu entnehmen müssen wir sie an der Wurzel des Heap entnehmen.
- den neuen Teilbaum ordnen wir wieder korrekt im Heap ein, indem wir ihn an den Anfang stellen und absinken lassen.

### Adaptive Huffman-Codierung

Bei dem geschilderten Verfahren müssen vorab die Häufigkeiten aller Zeichen ermittelt werden. Eine sinvolle Alternative stellt die adaptive Huffman-Codierung dar. Dabei werden zunächst Startwerte für die Häufigkeiten willkürlich angenommen. Diese werden dann während der Codierung nach jedem codierten Zeichen angepaßt.

Die Methode besitzt den Vorteil, daß sie sich dem Datenmaterial besser anpassen kann, z.B. wenn dieses Text und Binärcode gemischt enthält. Darüber hinaus muß die Codierungsvorschrift nicht mitgeliefert werden, da der Decodierer die Anpassung während der Decodierung nachvollziehen kannn.

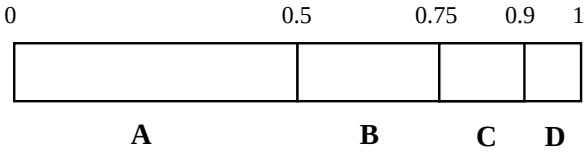
# 8.3 Arithmetische Kompression

Den arithmetischen Kompressionsverfahren liegt die Vorstellung zugrunde, daß die Häufigkeiten der Zeichen eines Zeichensatzes als Teilintervalledes Intervalls [0..1) aufgefaßt werden können, wobei alle Teilintervalle zusammen wieder das ganze Intervall ergeben müssen.

Ein Zeichen ist nun eindeutig charakterisiert, wenn man es durch eine beliebige Zahl aus seinem Teilintervall codiert.

### Beispiel

Der Zeichensatz bestehe aus den Zeichen A, B, C, D mit den Häufigkeiten 0.5, 0.25. 0.15 und 0.1. Dies führt zu folgender Aufteilung des Intervalls [0..1)



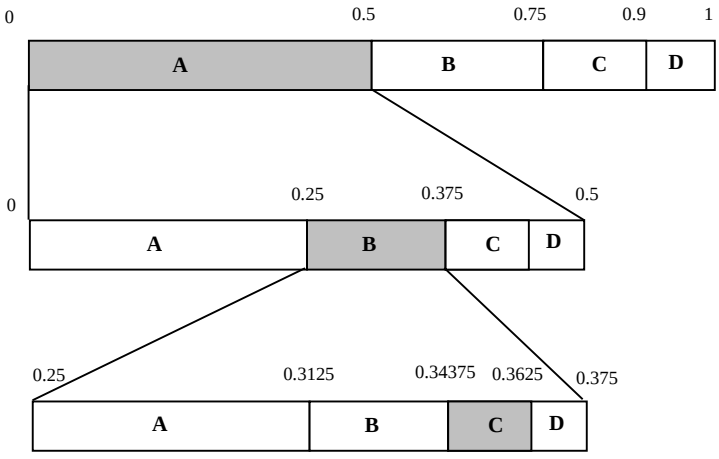

Die folgende Tabelle zeigt, wie den Zeichen aufgrund dieser Intervallteilung Codes zugeordnet werden können:

| Zeichen | Teilintervall | Binärbruch aus dem Teilintervall | Codewert |
|---|---|---|---|
| A | [0 .. 0.5) | 0.0b = 0 | 0 |
| B | [0.5 .. 0.75) | 0.1b = 0.5 | 1 |
| C | [0.75 .. 0.9) | 0.11b = 0.75 | 11 |
| D | [0.9 .. 10) | 0.1111b = 0.9375 | 1111 |

Die Arithmetische Codierung weitet diese für Einzelzeichen geschilderte Idee auf Zeichenfolgen aus:
Zunächst startet man mit einer Coddierung der Alphabetzeichen nach obigem Schema. Eine Zeichenfolge wird dann durch eine Intervallschachtelung codiert, wobei das erste Zeichen das Teilintervall des Intervalls [0 .. 1) angibt wie oben. Dieses Teilintervall wird für die Codierung des zweiten Zeichens wieder im gleichen Verhältnis wie das Gesamtintervall [0 .. 1) unterteilt und das zweite Zeichen einem der dadurch entstehenden Teilintervalle zugeordnet.

### Beispiel

Wenn das Alphabet und die Häufigkeiten der Zeichen wie im vorigen Beispiel angenommen werden, dann erhält man für die die Zeichenfolgen AB, AB, ABC die folgenden Teilintervalle:



| Zeichen-folge | Teilintervall | binäre Intervallgrenzen | Codewert |
|---|---|---|---|
| A | [0.0 .. 0.5) | [0.0b .. 0.1b) | 0 |
| AB | [0.25 .. 0.375) | [0.010b .. 0.011b) | 010 |
| ABC | [0.34375 .. 0.3625) | [0.010110b .. 0.010111b) | 010110b |

Zusätzlich zu diesem prinzipiellen Vorgehen sind bei dem Verfahren weitere Besonderheiten zu beachten:

- Die Bestimmung der Häufigkeiten kann wie bei der Huffmann-Kompression einmalig festgelegt oder während der Codierung adaptiert werden.
- Damit die Decodierung das Ende einer Zeichefolge erkennt muß der Zeichensatz um ein Sonderzeichen erweitert werden, das eine Intervallschachtelung beendet.
- Es muß verhindert werden, daß Rundungsfehler das Resultat unbrauchbar machen.
- Sobald die führenden Stellen der Intervallschachtelung konstant bleiben, können sie für die weitere Decodierung einer Zeichenfolge vernachlässigt werden, so daß die Codeworte verkürzt werden können.

# 8.4 Kompressionsverfahren

In den vorangehenden Abschnitten haben wir die gemeinsamen algorithmischen Grundlagen vieler Kompressionstechniken kennengelernt. In diesem Abschnitt betrachten wir nun konkreteAnwendungen dieser Techniken genauer.

## 8.4.1 Squeeze-Kompression

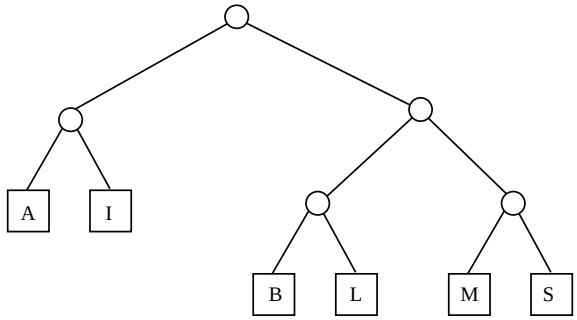
Das Squeezen kombiniert zur Kompression einer Bytefolge zwei Techniken:
- Zunächst wird eine Lauflängen-Codierung durchgeführt
- Danach wird das Ergebnis des ersten Schritts einer Huffman-Codierung unterworfen.

Für die Decodierung wird der Huffman-Baum in einem Array abgelegt. Jede Komponente dieses Array entspricht einem Baumknoten.

- Innere Baumknoten enthalten zwei Zeiger auf die Nachfolger-Knoten (16 Bit lang, höchstwertiges Bit = 0)
- Falls die Nachfolgerknoten Blätter sind enthält die Array-Komponente an der Stelle eines Zeigers im oberen Byte ein Flag ( X'FE') und im unteren Byte das Komplement des Zeichenwertes. Der Buchstabe A=X'41' wird somit durch X'FEBE' dargestellt.

**Beispiel**

Im Abschnitt 92. haben wir zur Codierung des String SIMSALABIM den folgenden Huffman-Baum aufgebaut:




Dieser entspricht der Code-Tabelle, die zur Codierung verwendet wird:

| A | B | I | L | M | S |
|---|---|---|---|---|---|
| 00 | 100 | 01 | 101 | 110 | 111 |

Zur Decodierung wird der Huffman-Baum wie folgt in eine Tabelle gespeichert:

| index: | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| left: | 0001 | FEBE=A | 0003 | FEBD=B | FEB2=M |
| right: | 0002 | FEB6=I | 0004 | FEB3=L | FEAC=S |

Der Code auf der nächsten Seite beschreibt die Decodierung in vereinfachter Form.

Die explizite Abspeicherung des Huffman-Baumes und die Interpretation dieser Datenstruktur bei der Decodierung kann alternativ auch so gelöst werden, daß statt der Tabelle eine Code-Sequenz generiert wird, die bei der Decodierung den selben Ablauf durch einen Block bedingter Verzweigungen realisert.

```
index = 0;                                  // zeige zur Baumwurzel
do
{
   liesNaechstesBit;                        // lies naechstes Bit ein
   if ( NaechstesBit == 0 )                 // NaechstesBit pruefen:
     wert =  HuffmanTree[index].left;       // falls =0, → links gehen
   else
     wert =  HuffmanTree[index].right;      // falls =1,→ links gehen
   if ( highBit(wert) == 0)                 // ist es ein Zeiger ?
     index = wert;                          // ja: weiter zum Unterbaum
   else
   {
     result = Komplement(lowByte(wert));    // nein: Zeichen abliefern
     break;                                 // und abbrechen
   }
} while (TRUE);
```

## 8.4.2 LZW-Kompressionstechniken

Die Squeeze-Kompression ging davon aus, daß in der zu komprimierenden Zeichenfolge die Zeichen statistisch unabhängig voneinander auftreten. Dies ist häufig nicht der Fall: nach einem ‘q’ kommt z.B. fast immer ein ‘u’. Das Lempel-Ziv-Welch-Kompressionsverfahren übersetzt daher nicht einzelne Zeichen, sondern Zeichenfolgen in je ein Codewort. Wir betrachten zwei Varianten davon:

- das LZW-Verfahren mit fester Codewort-Länge ohne Code-Recycling
- das LZW-Verfahren mit variabler Codewort-Länge mit Code-Recycling

### 8.4.2.1 LZW-Kompression mit fester Codewortlänge ohne Code-Recycling

#### Die Codierungstabelle

Die LZW-Komprimierung verwendet eine Tabelle, in der Zeichenfolgen, die schon einmal aufgetreten sind, abgelegt werden. Die Einträge in der Tabelle bestehen aus zwei Werten:

- dem ***Suffix***, das ein Zeichen darstellt, nämlich das jeweils letzte einer bekannten Zeichenfolge
- dem Index ***pred*** , der einen Zeiger auf das in der Zeichenfolge vorangehende Zeichen darstellt.

Der Zeiger ***pred*** kann außer einem Indexwert einige Spezialwerte annehmen, z.B.

- ***pred*** = ***free*** markiert eine freie Position der Tabelle
- ***pred*** = ***null*** markiert das linke Ende einer Zeichenfolge**.**

Das Codewort für eine Zeichenfolge ist der Tabellenindex ihres letzten Zeichens. Zu Beginn der Kompression und der Dekrompression wird die Tabelle mit dem erlaubten Zeichenvorrat initialisiert. Jedes Zeichen stellt eine Folge der Länge 1 dar:

| index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|---|
| ***suffix*** | A | B | C | D | E | F | G | H | ..... |
| ***pred*** | ***null*** | ***null*** | ***null*** | ***null*** | ***null*** | ***null*** | ***null*** | ***null*** | ..... |

Wenn eine neue Zeichenfolge gelernt werden muß, dann ist diese ein Zeichen länger als eine bereits bekannte. Es genügt daher das letzte Zeichen der Folge in die Tabelle einzutragen mit einem Verweis auf das letzte Zeichen des bereits bekannten Folgenanfangs.

**Beispiel:**

Die Zeichenfolge ABC stehe bereits in der Tabelle, z.B.

| index: | | i | | j | | k | | | u | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***suffix*** | | A | | B | | C | | | ..... | | | ..... |
| ***pred*** | | ***null*** | | **i** | | **j** | | | ***free*** | | | ..... |

Wenn nun die Zeichenfolge ABCD gelernt wird, dann wird D neu in die Tabelle eingetragen mit einem Verweis auf das Vorgängerzeichen C:

| index: | | i | | j | | k | | | u | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***suffix*** | | A | | B | | C | | | D | | | ..... |
| ***pred*** | | ***null*** | | **i** | | **j** | | | **k** | | | ..... |

In der ursprünglichen Version des LZW-Verfahrens war die Tabelle 4096 Einträge groß, d.h. die Adressierung erfolgte mit 12 Bit langen Indices und damit war auch die Codewortlänge 12 Bit. Neuere Versionen verwenden sogar 16 Bit lange Codeworte.

### Zugriffsverfahren zur Codierungstabelle

Die Adressierung der Tabelleneinträge wird über einen Hash-Algorithmus innerhalb der Funktion `Enter` realisiert. Die Hash-Funktion liefert als Zieladresse die mittleren 12 Bit von

$$(\boldsymbol{pred} + \boldsymbol{suffix})^2$$

Überläufer werden miteiander verkettet nach der Methode des **coalesced Hashing.** Die Codierungstabelle muß also außer den Feldern ***suffix*** und ***pred*** noch ein weiteres Feld ***link*** enthalten. Im Fall einer Kollision wird die Überläuferkette bis an ihr Ende durchlaufen. Der nächste frei Platz wird von dieser Stelle aus mit linearem Sondieren gesucht: Man sucht mit einer festen Schrittweite von 101 solange, bis einer gefunden ist, bringt das neue Element dort unter und hängt es an die Kette an.

### Codierung von Zeichenfolgen

Der folgende Pseudocode beschreibt die Codierung von Zeichenfolgen:

```
lastPred = null ;                  // am Anfang e.Zeichenfolge
do
{ liesNaechstesZeichen(Zeichen);// erstes Zeichen lesen
  do                               // Schleife über alle Zeich.
  {
    index=suche(lastPred, Zeichen); //suche Z. in Tabelle
    if (gefunden)                  // Zeich.folge bisher bekannt?
    {
      lastPred = index;            // ja: Vorgänger merken und
      break;                       // zu naechstem Zeichen
    }
    output(lastPred);              // nein:Code d.Folge bis hier
    Enter (lastPred, Zeichen);     // neues Zeichen als 1. der
    lastPred = null;               // naechsten Folge eintragen
  } while TRUE
} while TRUE
```

### Beispiel

Wenn die Folge ABC bereits bekannt ist und nun die Folge ABCD gelesen wird finden der Reihe nach die folgenden Aktionen statt:

1. 'A' lesen. Den Tabelleneintrag [A, ***null***] suchen. Dieser wird gefunden auf Position **i .**
2. 'B' lesen. Den Tabelleneintrag [B, **i**] suchen. Dieser wird gefunden auf Position **j** .
3. 'C' lesen. Den Tabelleneintrag [C, **j**] suchen. Dieser wird gefunden auf Position **k** .
4. 'D' lesen. Den Tabelleneintrag [D, **k**] suchen. Dieser wird nicht gefunden.
5. den Code **k** ausgeben als Codierung von ABC.
6. [D, **k**] neu in die Tabelle eintragen und an den Eintrag [C, **j**] anketten.
   Damit hat der Algorithmus die Zeichenfolge ABCD gelernt.
7. Die nächste Zeichenfolge codieren, wobei D als erstes Zeichen der neuen Folge verwendet wird. Im Beispiel ist diese bekannt und enthält DEF , d.h. der nächste generierte Codewort ist dann w.

Danach hat die Tabelle den folgenden Aufbau:

| index: | | i | j | k | | m | | u | v | w | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ***suffix*** | | A | B | C | | D | | D | E | F | ..... |
| ***pred*** | | ***null*** | **i** | **j** | | **k** | | **null** | **u** | **v** | ..... |

Es wird also nicht sofort der Code m der neu gelernten Folge ABCD ausgegeben, sondern zunächst nur die Folge ABC mt k codiert. Danach ist aber die Folge ABCD bekannt. Dies ist erforderlich, damit der Decodierer den Aufbau der Tabelle nachvollziehen kann.

### Decodierung von Zeichenfolgen

Wie bei der Kompression ist auch bei der Dekompression die Tabele nur mit der erlaubten Zeichenmenge initialisiert. längere Folgen werden erst im Verlauf der Decodierung gelernt. Ein Codewort der komprimierten Zeichenfolge ist der Index ihres letzten Zeichens in der Tabelle. Der folgende Pseudocode beschreibt den Verlauf der Decodierung:

```
lastPred = null;                  // neue Zeichenfolge beginnen
do
{
  input  (newPred);               // nächstes Codewort lesen
  Decode (newPred);               // Code rekursiv decodieren
  enter  (lastPred, firstChar); // verlängerte Z.folge lernen
  lastPred = newPred;             // f. Ankettung weiterer Zei.
}
```

Zum Decodieren der Zeichenfolge dient die folgende rekursive Funktion:

```
Decode (Index)
{
  pred   = Tabelle[index].pred;    // Zeiger aus Tabelle holen
  suffix = Tabelle[index].suffix;  // Zeich. aus Tabelle holen
  if (pred != null)                // 1. Zeichen erreicht ?
    Decode (pred);           // nein: der Zeichenkette folgen
  else                       // Anf. d. Zeichenkette erreicht
    firstChar = suffix;      // 1. Zeichen in globale Variable
  output (suffix);           // aktuelles Zeichen ausgeben
}
```

Um die Funktionsweise des Decodierers zu verstehen nehmen wir eine Situation an, in der:
– die Folgen ABC und DEF bereits gelernt wurden, die Folge ABCD aber noch nicht und
– daß danach die Folge ABCDEF codiert wurde.

Der Zustand der Tabelle war dabei zunächst:

| index: | | i | j | k | | | | u | v | w | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ***suffix*** | | A | B | C | | | | D | E | F | ..... |
| ***pred*** | | ***null*** | **i** | **j** | | | | **null** | **u** | **v** | ..... |

Die Folge ABCDEF wurde daher codiert zu **kw** und die Folge ABCD wurde gelernt, so daß die Tabelle danach den folgenden Inhalt hatte:

| index: | | i | j | k | | m | | u | v | w | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ***suffix*** | | A | B | C | | D | | D | E | F | ..... |
| ***pred*** | | ***null*** | **i** | **j** | | **k** | | **null** | **u** | **v** | ..... |

Bei der Decodierung an der gleichen Stelle hat die Tabelle noch den Aufbau.

| index: | | i | j | k | | | | u | v | w | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ***suffix*** | | A | B | C | | | | D | E | F | ..... |
| ***pred*** | | ***null*** | **i** | **j** | | | | ***null*** | **u** | **v** | ..... |

Es wird zunächst der Code **k** zu ABC decodiert. lastPred hat dann den Wert **k** .Danach finden folgende Aktionen statt:

1. Es wird der Code **w** gelesen und Decode (**w**) aufgerufen
2. Es wird F in der lokalen Variablen suffix gemerkt und `Decode(v)` aufgerufen.
3. Es wird E in der lokalen Variablen suffix gemerkt und `Decode(u)` aufgerufen.
4. Es wird D in der lokalen Variablen suffix gemerkt und `Decode(`***null***`)` aufgerufen.
5. Die globale Variable `firstChar` erhält den Wert des ersten Zeichens der Folge, nämlich D.
6. Es wird D ausgeben und die jüngste Instanz der Funktion Decode beendet.
7. Es wird E ausgegeben und auch diese Instanz der Funktion Decode beendet.
8. Es wird F ausgeben. Damit ist die Aufruf-Hierarchie von Decode wieder abgebaut.
9. Es wird [lastPred, firstChar] = [D, **j** ] neu in die Tabelle eingetragen.. Damit hat das Verfahren auch die Zeichenfolge ABCD gelernt.
10. `lastPred` wird auf F, das Schlußzeichen der zuletzt decodierten Folge gestellt.

Die Tabelle ist damit vom Decodierer in gleicher Weise ergänzt worden wie während der Codierung und hat den Aufbau:

| index: | | i | j | k | | m | | u | v | w | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ***suffix*** | | A | B | C | | D | | D | E | F | ..... |
| ***pred*** | | ***null*** | **i** | **j** | | **k** | | **null** | **u** | **v** | ..... |

### 8.4.2.2 LZW-Kompression mit variabler Codewortlänge und Code-Recycling

Diese Erweiterung des Verfahrens besitzt die folgenden beiden Besonderheiten:

– *variable Adresslänge*

Die Tabelle wird von unten her dicht belegt. In den ersten 256 Plätzen stehen die Folgen mit der Länge 1, also der Zeichenvorrat. Zur Adressierung des ersten freien Tabelleneintrags werden damit mindestens 9 Bit benötigt. Sind alle Tabellenpositionen, die mit 9 Bit adressiert werden können belegt, dann geht man zu 10 Bit langen Adressen über usw.

Dies setzt aber voraus, daß die Tabellenplätze nicht frei mit einem direkten Hashing vergeben werden, sondern die Tabelle von unten her dicht belegt wird.

Um trotzdem nicht auf eine Hash-Adressierung nicht verzichten zu müssen, verwendet man eine Hash-Tabelle `Hash` die auf die Positionen in der Code-Tabelle verweist. Die Hash-Tabelle ist größer als die eigentliche Code-Tabelle, um Kollisionen zu vermeiden. Bei Kollisionen verwendet wird lineares Sondieren eingesetzt, um eine Ausweichposition zu finden.

– *Code-Recycling*

Wenn eine Zeichenfolge gelernt und dann nie mehr benötigt wurde, kann ihr Tabelleneintrag bei voller Tabelle freigegeben und wiederverwendet werden. Dazu müssen aber benutzte Folgen markiert werden, um sie gegen Wiederverwendung zu schützen.

#### Zugriffsverfahren zur Codierungstabelle

Die Adressierung der Tabelleneinträge wird über indirektes Hashing realisiert. Der folgende Pseudocode beschreibt den Ablauf der Funktion `Enter`:

```
Enter (pred, suffix)
{
  Hindex = Hash(pred,suffix);      // Hash-Ziel des Eintrags
  while (Hash[Hindex] belegt)      // Überläuferkette verfolgen
    if (Link[Hindex] != null)      // wenn Zielpos. belegt ist,
      Hindex = (Hindex+5003) mod 5119;// freie Pos. in Hash suchen
  Hash[Hindex] nextCode;          // Zeiger auf CodeTabelle setzen
  CodeTable[nextCode] = (pred,suffix); // neue Folge → Codetabelle
  nextCode = nextCode +1;          // nächste freie Position in
                                   // der Codetabelle festhalten
}
```

Als Hashing-Funktion wird verwendet:

$256 * low(\mathbf{pred}) \text{ AND } 0Fh + high(\mathbf{pred} * 16) \text{ XOR } \mathbf{suffix}$

#### Variable Code-Länge

Sowohl bei der Codierung als auch bei der Decodierung wird die Anzahl der belegten Einträge der Code-Tabelle ständig aktualisiert, so daß die Codewortlänge angepaßt werden kann.

#### Decodierung von Zeichenfolgen

Sowohl bei der Codierung als auch bei der Decodierung können Zeichenfolgen gelernt werden, die dann nie mehr gebraucht werden. Bei voller Tabelle könne diese dann überschrieben werden. Der folgende Pseudocode zeigt wie dies realisiert wird:

```
  lastPred = null;                    // neue Zeichenfolge beginnen
  do                                  // Normales Decodieren solange
  {                                   // noch Platz in der Tabelle ist
    input  (newPred);                 // nächstes Codewort lesen
    Decode (newPred);                 // Codewort rekursiv decodieren
    enter  (lastPred, firstChar);     // verlängerte Z.folge lernen
    lastPred = newPred;               // für Ankettung weiterer Zeichen
  } while <CodeTable noch nicht voll>;
  do                                  // Decodieren mit Code-Recycling
  {                                   // bei voller Code-Tabelle
    input  (newPred);                 // nächstes Codewort lesen
    Decode (newPred);                 // Codewort rekursiv decodieren
    ReUse  (lastPred, firstChar);     // verlängerte Z.folge lernen
    lastPred = newPred;               // für Ankettung weiterer Zeichen
  } while <CodeTable voll>;
```

Beim Decodieren einer Zeichenfolge müssen alle während des Prozeses benutzten Tabelleneintrage markiert werden, um sie vor dem Löschen zu schützen. Die Funktion Decode ist daher wie folgt zu modifizieren:

```
  Decode (Index)
  {
    Markiere(Tabelle[Index];          // Position als benutzt markieren
    pred   = Tabelle[index].pred;     // Zeiger aus der Tabelle holen
    suffix = Tabelle[index].suffix;   // Zeichen aus der Tabelle holen
    if (pred != null)                 // erstes Zeichen erreicht ?
      Decode (pred);                  // nein: der Zeichenkette folgen
    else                              // Anfang der Z.kette erreicht
      firstChar = suffix;             // 1. Zeichen in globale Variable
    output (suffix);                  // aktuelles Zeichen ausgeben
  }
```

Die Funktion `ReUse` ist ähnlich wie die Funktion Enter aufgebaut:

```
ReUse (pred, suffix)
{
  Hindex = Hash(pred,suffix);              // Hash-Ziel des Eintrags
  do
  {
    if (Hash[Hindex] == frei) return;      // freie Einträge ignorieren
                                           // wenn Zielpos. belegt ist
    while (Tabelle[Hash[Hindex]] markiert) // nicht markierte,
      Hindex = (Hindex+5003) mod 5119;     // belegte Pos. suchen
  }
  CodeTable[Hash[Hindex]] = (pred,suffix); // neue Folge eintragen
}
```

Man beachte, daß ReUse nur belegte Einträge in der Hash-Tabelle verwenden darf. Wenn die Hash-Funktion den Eintrag der neuen Folge auf einem solchen Platz beabsichtigt, wird die Folge nicht gelernt. Wenn ein belegter, aber als benutzt markierter Eintrag gefunden wird, dann such die Funktion mit linearem Sondieren eine geeignete andere Stelle..

### 8.4.2.3 LZSS- und LZH-Kompression

Das Lempel-Ziv-Storer-Szymanski (LZSS)-Verfahren ist eine Abwandlung des LZW-Verfahrens. Statt der Codierungstabelle verwendet dieses Verfahren eine Ringpufferin dem die zuletzt gelesenen bzw. decodierten Zeichen der Original-Zeichenfolge stehen. Eine Zeichenfolge wird bei der Codierung im Ringpuffer gesucht und dann durch eine Längenangabe und eine Position im Ringpuffer codiert. Falls sie dort nicht gefunden wird, wird sie uncodiert weitergegeben.

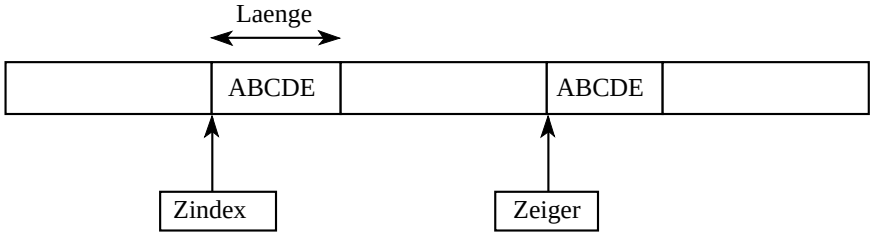
Dieser Ansatz erspart das Hashing beim Zugriff auf die Code-Tabelle. Trotz seiner Einfachheit liefert er gute Resultate, da der Ringpuffer im Gegensatz zur Code-Tabelle des LZW-Algorithmus sich besser den Merkmalen der zu codierenden Zeichenfolge anpassen kann als das Code-Recycling beim LZW-Verfahren.

Der folgende Pseudocode beschreibt die Codierung. Er verwendet drei charakteristische Konstanten:

| | |
|---|---|
| `Minlaenge` | Minimale Länge einer codierten Zeixchenfolge |
| `Maxlänge` | Maximale Länge einer codierten Zeixchenfolge |
| `BufferSize` | Größe des Rinpuffers |

```
LiesZeichenfolge;                          // Erläuterung unten
if (Laenge(Zeichenfolge) > Minlaenge       // Zeichenfolge lang genug ?
{
  output (Laenge(Zeichenfolge));           // Laenge ausgeben
  offset = Zeiger - Zindex;                // relative Adresse bestimmen
  output (offset );                        // relative Adresse ausgeben
}
else
  output (Zeichenfolge);                   // Zeichenfolge uncodiert
                                           // ausgeben
Zeiger = Zeiger+Laenge(Zeichenfolge);      // Zeiger aktualisieren
```

Erläuterung zu `LiesZeichenfolge:`
Die Funktion liest solange Zeichen in den Ringpuffer ein, wie diese Zeichenfolge im Ringpuffer vorkommen oder bis höchstens `Maxlänge` Zeichen gelesen sind. Ergebnis sind die Länge der Zeichenfolge und der Index Ihres Vorkommens im Puffer (`Zindex`). Wenn nicht mindestens `Minlaenge` Zeichen gelesen werden können, die schon früher vorkamen, wird die Zeichenfolge uncodiert in den Puffer übernommen und der Ringpuffer-Zeiger aktualisiert.




Es ist zu bemerken, daß der Algorithmus zwischen normalen Zeichen und Längenangaben unterscheiden muß. Die wird dadurch erreicht, daß man die Werte von ooh bis Ffh als Zeichencodes beläßt und Längenangaben ab dem Hexadezimawert 100h coddiert. 100h entspricht dann `MinLaenge`. Die Decodierung wird durch den folgenden Pseudocode beschrieben.

```
if  (nächstes Zeichen ist Längenangabe)
{
  Lies Laenge;
  Lies Offset;
  for  (i=1; i<= Laenge; i++)
  {
    Zeichen = Puffer[Zeiger - Offset];// Zeichen aus Puffer holen
    output (Zeichen);                 // Zeichen ausgeben
    Puffer[Zeiger] = Zeichen;         // Zeichen in den Ringpuffer
    Zeiger = Zeiger +1;               // Pufferzeiger aktualisieren
  }
}
else
{
  Lies Zeichen;                 // Jetzt kommt ein normales Zeichen
  output (Zeichen);             // Zeichen ausgeben
  Puffer[Zeiger] = Zeichen;     // Zeichen in den Ringpuffer
  Zeiger = Zeiger + 1;          // Pufferzeiger aktualisieren
}
```

**LZH-Kompression**

Der LZH-Codierung liegt das LZSS-Verfahren zugrunde. Es wird jedoch zusätzlich für die Ablage der Informationen im Ringpuffer eine Huffman-Codierung benutzt und Adressen werden nach einem besonderen Schema so codiert, daß niedrige Adreßwerte die kürzesten Codelaengen erhalten.

## 8.5 Kompression in Bilddatei-Formaten

Zur Archivierung von Bildern auf Externspeichern hat sich eine unübersehbare Vielfalt von Dateiformaten entwickelt. Neben spezifischen Techniken zur Speicherung der Bildinformation sind vor allem die Aspekte der Datenkompression hier von Interesse. Wir finden sowohl 'konventionelle' Kompressionsmethoden, wie Lauflängen-Codierung oder Huffmann-Codierung, bei denen die Original-Information wieder vollständig herstellbar ist. Hinzu kommen aber auch Techniken, die einen Teil des Komporessionsgewinns mit einem Informationsverlust bezahlen.

Eine Standardisierung ist bisher nicht in Sicht. Nur wenige Formate sind aufgrund ihrer Verbreitung als weithin akzeptierter Quasi-Standard anzusehen. Grundsätzlich sind drei unterschiedliche Speicherungstechniken für Bilder zu unterscheiden:

#### Vektorielle Formate

Dabei wird die Bildinformation objektbezogen abgespeichert: Ein Polygon wird dabei z.B. spezifiziert durch seine Stützpunkte und und globale Attribute wie Linientyp und Linienfarbe. Flächen werden charakterisiert durch ihr Randpolygon und Attribute wie Füllfarbe und Transparenz. Bilder, die als Vektor-Bilder vorliegen, benötigen wenig Speicherplatz und ihre Objekte können nachträglich leicht verändert werden. Vektorielle Formate eignen sich gut für Zeichnungen. Bekannte Vektorformate sind:

- das DXF-Format: Autodesk -Drawing Exchange-Format
- das CDR-Format: Corel Draw-Format
- das HPGL-Format: Plotter-Steuersprache für HP-kompatible Plotter

#### Metafile-Formate

Bei Metafile-Formaten wird die Bild-Information in einer Beschreibungssprache niedergelegt. Sie eignen sich gut für den Datenaustausch oder zur Bildausgabe. In diese Kategorie fallen u.a. die folgenden Dateiformate:

- das WMF-Format: Windows Metafile-Format
- das CGM-Format: Computer Graphics Metafile-Format
- PS- und EPS-Format: (encapsulated) Postskript-Format

#### Pixelformate

Bildpunkt-orientierte Dateiformate sind für die Bildverarbeitung die weitaus wichtigste Speicherungsform für Bilder und in vielen Fällen auch die einzig mögliche. Da sie oft sehr viel Speicherplatz benötigen, spielen Kompressionstechniken eine wichtige Rolle. Neben der Einsparung von Speicherplatz ist die Verringerung der notwendigen Übertragungsbandbreite ein wichtiges Ziel für die Bildkompression, insbesondere bei der Übertragung von Bildern über Netze und beim Abspielen digitaler Filme von Plattenspeichern. Man unterscheidet

- unkomprimierte Formate,
- Formate mit verlustloser Kompression und
- Formate mit verlustbehafteter Kompression.

Manche Dateiformate, z.B. das TIF-Format ermöglichen unterschiedliche Kompressionstechniken, bei verlustbehafteten Formaten ist die Bildqualität durch Parameter steuerbar. Wir betrachten im folgenden exemplarisch die folgenden Dateiformate:

- das PCX-Format
- das BMP-Format
- das TIF-Format
- das JPEG-Format

## 8.5.1 Das PCX-Format

Das PCX-Format wurde ursprünglich von Z-Soft entwickelt und wird heute von zahlreichen Graphikprogrammen unterstützt. In seiner Versionsgeschichte spiegelt sich auch die Entwicklung der Graphik-Fähigkeiten der Rechner: Während die PCX-Version 0 nur monochrome und vierfabige Bilder speichern konnte, unterstützt das PCX-Format in der Version 5 Palettenbilder mit 256 Farben und RGB-Bilder mit 16,7 Millionen Farben.

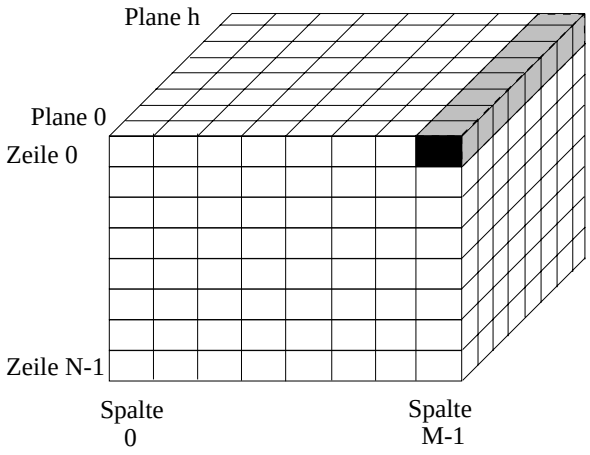



**Abbildung 8.1:** Ebenen-Struktur eines Bildes

Das PCX-Format verwendet eine einfache Lauflängen-Codierung, also eine verlustlose Komprimierung. Es ist einfach zu lesen und zu schreiben, aber die erreichbaren Kompressionsraten sind nicht sehr hoch.

Um das PCX-Format zu verstehen, stellen wir uns ein Bild mit $M$ Zeilen, $N$ Spalten und einer Pixeltiefe von $h$ Bit als einen Würfel vor mit $M*N*h$ Elementarzellen (s. Abbildung 8.1). Eine PCX-Datei besteht entsprechend diesem Modell aus bis zu drei Abschnitten:
- dem Header
- den Bilddaten und
- der Farbpalette

Während Header und Bilddaten immer vorhanden sind, kann die Farbpalette am Dateiende fehlen.

### Der PCX-Header

Der 128 Byte lange Header einer PCX-Datei enthält globale Informationen über das Bild. Die Bedeutung der einzelnen Header-Felder ist der Tabelle 8.1 zu entnehmen. Jedes Feld besitzt eine eindeutige Position im Header und Felder, die länger als ein Byte sind, legen das höherwertige Byte an der höheren Speicheradresse ab.

### Die PCX-Bild-Daten

Im Bilddaten-Bereich einer PCX-Datei sind die Farbwerte der Pixel Zeile für Zeile abgespeichert, wobei die oberste Bildzeile zuerst abgelegt wird. Die Zeilen-Länge ist dem Header zu entnehmen, so daß keine Trennzeichen zwischen den Zeilen benötigt werden.

Für Pallettenbilder mit 256 Farben wird pro Pixel 1 Byte benötigt. Bei Echtfarbenbildern werden für jede Zeile die drei Kanäle direkt nacheinander abgespeichert, zunächst alle Rot-Intensitäten, danach die Grün-Intensitäten dieser Zeile und schließlich die Blau-Werte.

Jede Zeile wird für sich komprimiert nach einem einfachen Schema: Für mehrere aufeinanderfolgende, gleiche Pixelwerte wird zunächst ein Wiederholungsfaktor in einem Zählerbyte abgelegt und im nächsten Byte der Pixelwert selbst. Zählerbytes sind dadurch gekennzeichnet, daß ihre beiden höchstwertigen Bits gesetzt sind. Der größtmögliche Wiederholungsfaktor ist somit 63.

| Adresse | Länge | Feldname | Bedeutung |
|---|---|---|---|
| 0 | 1 | Kennbyte | muß stets 10=X'0A' sein (Z-Soft) |
| 1 | 1 | Version | 0 = Version 2.5 ohne Palette<br>2 = Version 2.8 mit Palette<br>3 = Version 2.8 ohne Palette<br>5 = Version 3.0 mit Palette |
| 2 | 1 | Kodierung | 0 = keine Kompression<br>1 = Lauflängen-Kompression |
| 3 | 1 | Bits pro Pixel | 1 = Schwarz/Weiß-Bild<br>2 = Farbbild mit 4 Farben<br>8 = Farbbild mit 256 oder 16,7 Mio Farben |
| 4 | 8 | Bildgröße | 4 Werte mit je 2 Byte: left, top, right, bottom<br>Bildhöhe = bottom - top +1<br>Bildbreite = right - left +1 |
| 12 | 2 | horizontale Auflösung | Angaben in dpi für die Ausgabe |
| 14 | 2 | vertikale Auflösung | Angaben in dpi für die Ausgabe |
| 16 | 48 | Palette | Palette mit *3∗16* RGB-Einträgen |
| 64 | 1 | reserviert | immer 0 |
| 65 | 1 | Farbebenen | bis zu 4 Ebenen möglich |
| 66 | 2 | Bytes/Zeile | Anzahl der Bytes pro Zeile in jeder Farbebene |
| 68 | 2 | PalettenInformation | 1 = Farb-Paletten oder Schwarz/Weiß-Bild<br>2 = Graustufenbild oder RGB-Bild |
| 70 | 2 | horizontale Bildschirm-Größe | Anzahl von Bildpunkten -1 |
| 72 | 2 | vertikale Bildschirm -Größe | Anzahl von Bildpunkten -1 |
| 74 | 54 | reserviert | nicht verwendet |

**Tabelle 8.1:** Aufbau eines PCX-Header

Daraus folgt aber auch, daß Farbwerte, die größer als 192=X'C0' sind, immer mit einem Wiederholungsfaktor gespeichert werden und daher mindestens 2 Byte benötigen. Die Abbildung 8.1 zeigt das Verfahren zum Einlesen von PCX-Zeilen.

Die folgenden Beispiele verdeutlichen die Methode:

| Pixelwerte (hexadezimal): | gespeichert als: |
|---|---|
| *24 24 24 24 24 24 24* | *C7 24* |
| *C0* | *C0* |
| *C1* | *C1 C1* |
| *01 02 03 04* | *01 02 03 04* |

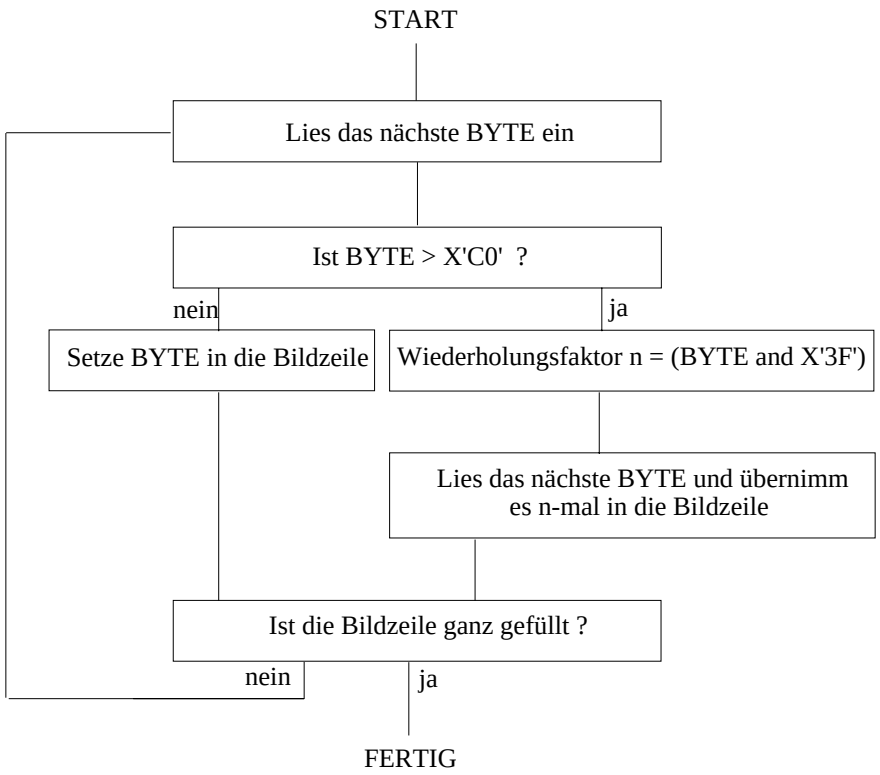



**Abbildung 8.9:** Verfahren zum Lesen von PCX-Zeilen

### Die PCX-Farbtabelle

Ursprünglich war für die Farbtabelle eines Bildes nur ein 48 Byte langer Bereich im Header vorgesehen. Für Bilder mit 256-Farben-Palette wird diese an die Bilddaten angehängt. Sie ist mit der Distanz *3*256=768* Byte vom Dateiende her aufzufinden und wird durch ein Byte mit dem Wert X‘0C’ eingeleitet.

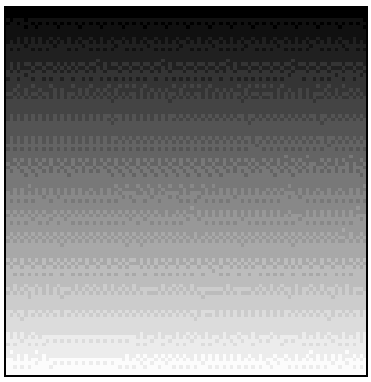


**Abbildung 8.10:** Graukeil mit 64 Graustufen

### Beispiel einer PCX-Datei

Wir betrachten nun den Dateiinhalt eines Beispielbildes, das als PCX-Datei gespeichert wurde. Es wird dazu der in Abbildung 8.10 gezeigte, horizontale Graukeil zugrunde gelegt.

Bei der Interpretation des folgenden hexadezimalen Datei-Auszugs ist zu beachten, daß in Feldern, die zwei oder mehr Bytes lang sind, stets das niederwertigste Byte zuerst abgespeichert wird. Da jede Bildzeile 100 Byte lang ist und nur einen Grauwert besitzt, benötigt die Lauflängen-Codierung zwei Zählerbytes X'FF' und X'E5' für die Wiederholungsfaktoren 63 und 37.

*Header:*

```
0A 05 01 08 00 00 00 00  63 00 63 00 64 00 64 00
```
Kennbyte, Version 5, 8 Bit/Pixel
Bildgröße: (0,0) .. (99,99)
Auflösung 100 dpi / 100 dpi

```
00 00 00 FC FC FC 0C 0C  0C 14 14 14 1C 1C 1C 20
20 20 28 28 28 30 30 30  38 38 38 3C 3C 3C 44 44
44 4C 4C 4C 54 54 54 5C  5C 5C 60 60 60 68 68 68
```
16-Farben-Palette

```
00 01 64 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```
0, eine Farbebene, 100 Bytes/Zeile,
Rest nicht verwendet

*Bilddaten*:

```
FF 00 E5 00 FF 24 E5 24  FF 24 E5 24 FF 2A E5 2A
FF 02 E5 02 FF 02 E5 02  FF 25 E5 25 FF 03 E5 03
FF 03 E5 03 FF 35 E5 35  FF 04 E5 04 FF 04 E5 04
FF 05 E5 05 FF 05 E5 05  FF 2E E5 2E FF 06 E5 06
.....................
FF 36 E5 36 FF 1F E5 1F  FF 1F E5 1F FF 20 E5 20
FF 20 E5 20 FF 39 E5 39  FF 21 E5 21 FF 21 E5 21
FF 3B E5 3B FF 22 E5 22  FF 22 E5 22 FF 3D E5 3D
FF 23 E5 23 FF 23 E5 23  FF 01 E5 01 FF 01 E5 01
```
Beginn der Bilddaten.

Ende der Bilddaten.

*256-Farben-Tabelle:*

```
0C 00 00 00 FC FC FC 0C  0C 0C 14 14 14 1C 1C 1C
20 20 20 28 28 28 30 30  30 38 38 38 3C 3C 3C 44
44 44 4C 4C 4C 54 54 54  5C 5C 5C 60 60 60 68 68
68 70 70 70 78 78 78 7C  7C 7C 84 84 84 8C 8C 8C
.....................
2C 2C EC EC EC 88 88 88  F4 F4 F4 50 50 50 90 90
90 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```
Kennbyte X'0C', RGB-Werte

Rest der Farbtabelle unbenutzt

```
.....................
```

## 8.5.2 Das BMP-Format

Als Standard-Format unter Microsoft Windows für Pixel-Bilder hat das BMP-Format weite Verbreitung gefunden. Auch BMP-Dateien betehen aus einem Header, einer Farbpalette und den Bilddaten.

### Der BMP-Header

Der Header einer BMP-Datei wird unterteilt in einen Core-Header und einen Info-Header. Wärend der Core-Header immer angegeben werden muß, sind beim Info-Header nur die ersten 16 Bytes obligatorisch. Tabelle 8.2 zeigt den Aufbau des Core-Header und des Info-Header.

*Core-Header:*

| Adresse | Länge | Feldname | Bedeutung |
|---|---|---|---|
| 0 | 2 | Dateityp | muß stets X'4D42' = 'BM' sein |
| 2 | 4 | Dateigröße | Größe der Bilddatei in Worten à 4 Bytes |
| 6 | 2 | reserviert | 0 |
| 8 | 2 | reserviert | 0 |
| 10 | 4 | Offset | Distanz der Bilddaten zum Dateianfang |

*Info-Header:*

| Adresse | Länge | Feldname | Bedeutung |
|---|---|---|---|
| 0 | 4 | Länge | Länge des Info-Header |
| 4 | 4 | Breite | Breite des Bildes in Pixeln |
| 8 | 4 | Höhe | Höhe des Bildes in Pixeln |
| 12 | 2 | Ebenen | stets 1 |
| 14 | 2 | BitsproPixel | zulässige Werte sind 1, 4, 8 und 24 |
| 16 | 4 | Komprimierung | sollte stets 0=unkomprimiert sein |
| 20 | 4 | Bildgröße | Anzahl von Bytes für die Bilddaten |
| 24 | 4 | horizontale Auflösung | in Pixel pro Meter angegeben |
| 28 | 4 | vertikale Auflösung | in Pixel pro Meter angegeben |
| 32 | 4 | Anzahl der Farben | Gesamtzahl verschiedener Farbtöne |
| 36 | 4 | Anzahl wichtiger Farben | Palette muß sortiert sein ! |

**Tabelle 8.2:** Struktur des BMP-Header

### Die BMP-Farbtabelle

Die Farbtabelle folgt unmittelbar auf die Header-Information. Wenn die Anzahl wichtiger Farben kleiner ist als die Gesamtzahl der Paletteneinträge, dann müssen die wichtigen Farbtöne am Anfang der Palette stehen. Alle darüber hinaus vorhandenen Farbwerte werden durch ähnliche Farben approximiert.

Jeder Eintrag der Farbtabelle besteht aus 4 Byte, wobei die ersten drei Byte die Rot- Grün- und Blau-Intensitäten angeben und der vierte Wert bisher nicht verwendet wird.

### Die BMP-Bilddaten

Die Bilddaten werden zeilenweise abgelegt, wobei die unterste Zeile zuerst abgespeichert wird. Die Anzahl der Bytes einer Zeile muß ein Vielfaches von 4 sein. Innerhalb einer Zeile stehen die Pixelwerte von links nach rechts. Je nach der Zahl der Bits pro Pixel belegt ein Bildpunkt mindestens 1 Bit und höchstens 3 aufeinanderfolgende Bytes.

### Beispiel einer BMP-Datei

Wir untersuchen nun, welchen Inhalt die BMP-Datei besitzt, die wir für den Graukeil der Abbildung 8.10 erhalten. Das Bild wurde als unkomprimiertes Palettenbild gespeichert. Es ist wieder zu beachten, daß stets das niederwertigste Byte zuerst abgespeichert wird.

*Core-Header:*

```
42 4D D2 0A 00 00 00 00  00 00 36 04 00 00        'BM', Dateigröße, Offset
                                                     der Bilddaten=X'436'
```

*Info-Header:*

```
                                          28 00      Info-Headerlänge=40 Byte,
00 00 64 00 00 00 64 00  00 00 01 00 08 00 00 00  100x100-Pixel, 1 Plane, 8 Bit/Pixel
00 00 10 27 00 00 00 00  00 00 00 00 00 00 00 00  unkompr, X'2710'=10000 Bytes,
00 00 00 00 00 00                                    Rest hier nicht benutzt
```

*Farbtabelle:*

```
                 00 00  00 00 00 00 BF 00 00 BF  4 Byte je Tabellenposition
00 00 00 BF BF 00 BF 00  00 00 BF 00 BF 00 BF BF
00 00 C0 C0 C0 00 C0 DC  C0 00 F0 C8 A4 00 04 04
04 00 08 08 08 00 0C 0C  0C 00 10 10 10 00 14 14
.....
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 F0 FB
FF 00 A4 A0 A0 00 80 80  80 00 00 00 FF 00 00 FF
00 00 00 FF FF 00 FF 00  00 00 FF 00 FF 00 FF FF
00 00 FF FF FF 00
```

*Bilddaten:*

```
                 46 46  46 46 46 46 46 46 46 46  unterste Zeile, alle Pixel weiß,
46 46 46 46 46 46 46 46  46 46 46 46 46 46 46 46  Index X'46'
46 46 46 46 46 46 46 46  46 46 46 46 46 46 46 46
46 46 46 46 46 46 46 46  46 46 46 46 46 46 46 46
.....
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  oberste Zeile, alle Pixel schwarz, Index 0
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

## 8.5.3 Das TIF-Format

Das Tag Image File (TIF) -Format wurde zunächst für den Desktop Publishing-Bereich entworfen. Aufgrund seiner Fleixibilität ist es aber zu einem weithin akzeptierten Quasi-Standard für Pixel-Bilder geworden. Die meisten Graphik-Applikationen akzeptieren TIF-Dateien, wobei es aber Unterschiede im Umfang der implementierten Funktionen gibt.

### Tags

Bei den oben besprochenen Dateiformaten werden alle Felder im Header oder in der Farbtabelle durch ihre Position innerhalb der Bilddatei aufgefunden. Dies führt zu Problemen bei Erweiterungen der Formatspezifikation.

Das TIF-Format stellt dagegen jedem Feld eine Marke voran, die als Tag bezeichnet wird und angibt, wie der Feldinhalt zu interpretieren ist. Man gewinnt dadurch große Flexibilität bei der Anordnung der Felder innerhalb der Datei, aber auch bei der Erweiterung des Formats um neue Eigenschaften. Eine TIF-Datei kann sogar mehrere Bilder enthalten, z.B. ein Originalbild und eine verkleinerte Kopie für Preview-Zwecke.

Insgesamt kennt das TIF-Format in der zur Zeit aktuellen Version bis zu 45 verschiedene Tags. Statt einer detaillierten Darstellung stellen wir daher weiter unten nur einige davon exemplarisch vor.

### TIF-Header

Eine TIF-Datei beginnt mit einem kurzen Header. Er gibt Auskunft über die verwendete Konvention zur Speicherung von Datenwerten (Intel-Format mit dem niederwertigsten Byte zuerst oder Motorola-Format mit dem höchstwertigen Byte zuerst). Außerdem enthält er die Identifikation X'002A' mit der die Datei als TIF-Datei zu identifiziern ist und einen Zeiger auf die Liste der Image File Directories (IFD).

Er besitzt den folgenden Aufbau:

| Adresse | Länge | Feldname | Bedeutung |
|---|---|---|---|
| 0 | 2 | Datenformat | X'4949'='II' niederwertigstes Byte zuerst<br>X'4D4D'='MM' höchstwertigstes Byte zuerst |
| 2 | 2 | Version | Immer = X'002A' |
| 4 | 4 | IFD-Offset | Zeiger zum Anfang der IFD-Liste |

**Tabelle 8.3:** Struktur des TIF-Header

### Image File Directories (IFD)

Zu jedem Bild, das eine TIF-Datei enthält gibt es ein eigenes IFD. Alle IFD's sind zu einer Liste verkettet, deren Lage innerhalb der Datei beliebig ist. Jedes IFD besteht aus drei Bereichen:
- der Anzahl von Tags im IFD
- einer Folge von Tags, nach Tag-Nummern geordnet.
- einem Zeiger zum nächsten IFD in der Liste

### Tag-Struktur

Tabelle 8.4 gibt einen Überblick über den Aufbau der Tags. Jedes Tag ist 12 Byte lang und besteht aus 4 Feldern. Je nach dem Datenformat werden die Werte der Felder im Intel- oder im Motorola-Format gespeichert. Das Datenfeld eines Tag ist nur 4 Byte lang, Falls dies nicht ausreicht, werden die Daten ausgelagert und das Datenfeld enthält dann einen Zeiger auf die Daten.

| Adresse | Länge | Feldname | Bedeutung |
|---|---|---|---|
| 0 | 2 | Tag | gibt den Tag-Typ an (254, ... , 320) |
| 2 | 2 | Datentyp | 1 = Byte<br>2 = ASCII-Strinng, mit 0-Byte abgeschlossen<br>3 = SHORT (16 Bit unsigned integer)<br>4 = LONG (32 Bit unsigned integer)<br>5 = RATIONAL<br>(Bruch aus zwei LONG-Werten) |
| 4 | 4 | Count | Anzahl von Tag-Daten; die Länge der Daten in Byte erhält man, indem man Count für die Datentypen 1, 2, 3, 4, 5 mit den Faktoren 1, 1, 2, 4 bzw. 8 multipliziert. |
| 8 | 4 | Offset/Value | Adresse des Datenfeldes in der Datei, falls die Daten länger als 4 Byte sind oder die Tag-Daten, wenn 4 Byte genügen. |

**Tabelle 8.4:** Aufbau der Tags

### TIF-Bildklassen

Das TIF-Format unterscheidet vier verschiedene Bild-Klassen:

- Klasse B: Binärbilder
- Klasse G: Graustufenbilder
- Klasse P: Palettenbilder
- Klasse R: RGB-Echtfarbenbilder.

Tags können je nach Bildklasse verschieden interpretiert werden und verschiedene Default-Werte besitzen.

### Beispiel einer TIF-Datei

Wir betrachten nun den Inhalt einer TIF-Datei, die den Graukeil der Abbildung 8.10 enthält. Das Bild wurde als unkomprimiertes Palettenbild gespeichert. Es ist wieder zu beachten, daß wegen des hier benutzten Intel-Formats das niederwertigste Byte zuerst abgespeichert wird. Es wird empfohlen, die Inhalte vom IFD ausgehend zu analysieren.

*TIF-Header (Offset X'00'):*

```
49 49 2A 00 90 29 00 00                    Intel-Format,
                                           Version X'002A'
                                           IFD-Offset X'2990'
```

*GrayResponseCurve(Offset X'08'):*

```
                           D7 07 D0 07 C8 07 C0 07
B8 07 B0 07 A8 07 A0 07    98 07 91 07 89 07 81 07
79 07 71 07 69 07 61 07    59 07 52 07 4A 07 42 07
3A 07 32 07 2A 07 22 07    1A 07 13 07 0B 07 03 07

.........................

95 00 8D 00 85 00 7D 00    76 00 6E 00 66 00 5E 00
56 00 4E 00 46 00 3E 00    37 00 2F 00 27 00 1F 00
17 00 0F 00 07 00 00 00
```

*Xresolution / Yresolution (Offset X'208' / X'210'):*

```
                        2C 01 00 00 01 00 00 00
2C 01 00 00 01 00 00 00
```

*Zeiger-Vektor (Offset X'218'):*

```
                        80 02 00 00 A0 05 00 00
C0 08 00 00 E0 0B 00 00  00 0F 00 00 20 12 00 00
40 15 00 00 60 18 00 00  80 1B 00 00 A0 1E 00 00
C0 21 00 00 E0 24 00 00  00 28 00 00
```

*StripByteCounts (Offset X'24C'):*

```
                                       20 03 00 00
20 03 00 00 20 03 00 00  20 03 00 00 20 03 00 00
20 03 00 00 20 03 00 00  20 03 00 00 20 03 00 00
20 03 00 00 20 03 00 00  20 03 00 00 90 01 00 00
```

*1. Streifen (Offset X'0280'):*

```
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   1. Bildzeile des 1. Streifens
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

.....

```
FC FC FC FC FC FC FC FC  FC FC FC FC FC FC FC FC   letzte. Bildzeile des 13. Streifens
FC FC FC FC FC FC FC FC  FC FC FC FC FC FC FC FC
FC FC FC FC FC FC FC FC  FC FC FC FC FC FC FC FC
FC FC FC FC FC FC FC FC  FC FC FC FC FC FC FC FC
```

*Image File Directory (IDF) (Offset X'2990'):*
Das Image File-Directory wurde wegen der besseren Lesbarkeit in die einzelnen Tags aufgeteilt:

```
00 0F                                          IFD enthält 15 Tags
FE 00 04 00 01 00 00 00 00 00 00 00            Tag 254: NewSubfileType
00 01 03 00 01 00 00 00 64 00 00 00            Tag 256: ImageWidth
01 01 03 00 01 00 00 00 64 00 00 00            Tag 257: ImageLength
02 01 03 00 01 00 00 00 08 00 00 00            Tag 258: BitsPerSample
03 01 03 00 01 00 00 00 01 00 00 00            Tag 259: Compression
06 01 03 00 01 00 00 00 01 00 00 00            Tag 262: PhotometricInterpretation
11 01 04 00 0D 00 00 00 18 02 00 00            Tag 273: StripOffsets
15 01 03 00 01 00 00 00 01 00 00 00          Tag 277: SamplesPerPixel
16 01 04 00 01 00 00 00 08 00 00 00            Tag 278: RowsPerStrip
17 01 04 00 0D 00 00 00 4C 02 00 00            Tag 279: StripByteCounts
1A 01 05 00 01 00 00 00 08 02 00 00            Tag 282: XResolution
1B 01 05 00 01 00 00 00 10 02 00 00            Tag 283: YResolutuion
22 01 03 00 01 00 00 00 03 00 00 00            Tag 290: GrayResponseUnit
23 01 03 00 00 01 00 00 08 00 00 00            Tag 291: GrayResponseCurve
28 01 03 00 01 00 00 00 02 00 00 00            Tag 296: Resolution Unit
00 00 00 00                                    Ende der IFD-Liste
```

*Ende der TIF-Datei* ----------

Im folgenden werden die hier benutzten Tags kurz erläutert. Eine vollständige Referenz ist in der TIF-Spezifikation zu finden [ ].

*Tag 254: NewSubfileType*

| | | | |
|---|---|---|---|
| 00 FE = 254 | 00 04 = LONG | 00 00 00 01 | 00 00 00 00 |

Der LONG-Wert dieses Tag wird als Flag-Vektor verwendet , wobei nur die folgenden Flag-Bits definiert sind:

| | |
|---|---|
| 00 00 00 00 | Bild mit voller Auflösung |
| Bit 0 gesetzt | Bild mit reduzierter Auflösung zu einem anderen Bild in dieser TIF-Datei |
| Bit 1 gesetzt | eine Seite eines mehrseitigen Bildes |
| Bit 2 gesetzt | Transparenzmaske für ein anderes Bild |

*Tag 256: Image Width*

| | | | |
|---|---|---|---|
| 01 00 = 256 | 00 03 = SHORT | 00 00 00 01 | 00 64 |

Gibt die Länge einer Bildzeile an, nämlich X'64'=100 Pixel:

*Tag 257: Image Length*

| | | | |
|---|---|---|---|
| 01 01 = 257 | 00 03 = SHORT | 00 00 00 01 | 00 64 |

Gibt die Länge einer Bildzeile an, nämlich X'64'=100 Pixel:

*Tag 258: BitsPerSample*

| | | | |
|---|---|---|---|
| 01 02 = 258 | 00 03 = SHORT | 00 00 00 01 | 00 08 |

Gibt an, daß 8 Bit pro Pixel benutzt werden.

*Tag 259: Compression*

| | | | |
|---|---|---|---|
| 01 03 = 258 | 00 03 = SHORT | 00 00 00 01 | 00 01 |

Gibt die verwendete Komprimierungtechnik an:
1 = unkomprimierte Bilddaten
2 = Huffman-kodierte Bilddaten, nur für Binärbilder (Klasse B) benutzt
5 = LZW-Codierung für Bilder der Klassen G, P, R

*Tag 262: PhotometricInterpretation*

| | | | |
|---|---|---|---|
| 01 06 = 262 | 00 03 = SHORT | 00 00 00 01 | 00 01 |

Gibt den Bildtyp an:
0 = Binär- oder Graustufenbild, mit Pixelwert 0=weiß
1 = Binär- oder Graustufenbild, mit Pixelwert 0=schwarz
2 = RGB-Farbbild
3 = Palettenbild
4 = Transparenzmaske

*Tag 273: StripOffsets*

| | | | |
|---|---|---|---|
| 01 11 = 273 | 00 04 = LONG | 00 00 00 0D | 00 00 02 18 |

Die Bilddaten sind in Streifen abgelegt (maximal 8 KB pro Streifen möglich). Der Datenwert des Tag ist ein Zeiger auf einen Zeiger-Vektor, über den man zu den Bildstreifen gelangt. Der Zeigervektor besitzt hier X'0D'=13 Zeiger auf Streifen.

*Tag 277: SamplesPerPixel*

| | | | |
|---|---|---|---|
| 01 15 = 277 | 00 03 = SHORT | 00 00 00 01 | 00 00 00 01 |

Anzahl von Werten pro Pixel; hat für Bilder der Klassen B, G, P den Wert 1 und für Bilder der Klasse R den Wert 3.

*Tag 278: RowsPerStrip*

| | | | |
|---|---|---|---|
| 01 16 = 278 | 00 04 = LONG | 00 00 00 01 | 00 00 00 08 |

Anzahl von Bildzeilen in jedem der Streifen = 8

*Tag 279: StripByteCounts*

| | | | |
|---|---|---|---|
| 01 17 = 279 | 00 04 = LONG | 00 00 00 01 | 00 00 02 4C |

Zeiger auf ein Array mit Längenangaben für die Streifen.

*Tag 282: XResolution*

| | | | |
|---|---|---|---|
| 01 1A = 282 | 0005=RATIONAL | 00 00 00 01 | 00 00 02 08 |

Zeiger auf einen 8 Byte langen RATIONAL-Wert, der die X-Auflösung bei der angibt.

*Tag 283: YResolution*

| | | | |
|---|---|---|---|
| 01 1B = 283 | 0005=RATIONAL | 00 00 00 01 | 00 00 02 10 |

Zeiger auf einen 8 Byte langen RATIONAL-Wert, der die Y-Auflösung bei der Ausgabe angibt.

*Tag 290: GrayResponseUnit*

| | | | |
|---|---|---|---|
| 01 22 = 290 | 0003=SHORT | 00 00 00 01 | 00 03 |

Einheiten der Graustufen-Kurve, hier 3=1/1000

*Tag 291: GrayResponseCurve:*

| | | | |
|---|---|---|---|
| 01 23 = 291 | 0003=SHORT | 00 00 00 01 | 00 08 |

Zeiger auf eine Tabelle, die die originalgetreue Wiedergabe der Grauwerte steuert.

*Tag 296: ResolutionUnit:*

| | | | |
|---|---|---|---|
| 01 28 = 283 | 0003=SHORT | 00 00 00 01 | 00 02 |

Dimension für die X- und Y-Auflösung :
1 = keine Angabe
2 = Inch
3 = Zentimeter

## 8.5.4 Das JPEG-Format

Das JPEG-Verfahren wurde von einem internationalen Normungsgremium, der Joint Photographic Experts Group konzipiert für die Speicherung einzelner Pixel-Bilder. Um Kompressionsraten bis zu 90% bei akzeptabler Bildqualität zu erreichen, werden verschiedene Methoden kombiniert eingesetzt. Abbildung 8.11 zeigt den schematischen Ablauf einer JPEG-Kodierung.

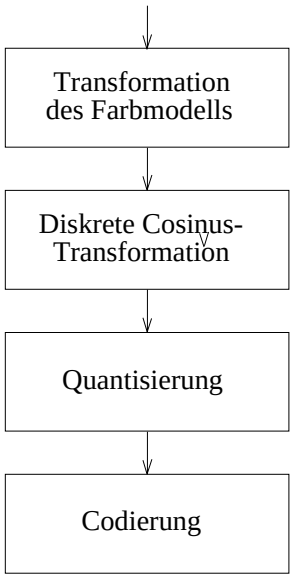



**Abbildung 8.11:** Verfahrensschritte der JPEG-Methode

### Die Transformation des Farbmodells

Zunächst wird das Bild in den YUV-Raum transformiert. Dazu müssen die RGB-Farbwerte umgerechnet werden in einen Luminanzwert Y und zwei Chrominanzwerte *Cb* und *Cr* nach der Beziehung:

$$\begin{bmatrix} Y \\ Cr \\ Cb \end{bmatrix} = \begin{bmatrix} 0.2990 & 0.5870 & 0.1140 \\ -0.1687 & -0.3313 & 0.5000 \\ 0.5000 & -0.4187 & -0.0813 \end{bmatrix} * \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Unser Auge ist für Helligkeitsunterschiede empfindlicher als für Farben, daher genügt es, die Chrominanzwerte mit geringerer räumlicher Auflösung zu speichern. Für ein 2x2 Pixel großes Quadrat werden dann 4 Luminanzwerte und nur je ein *Cr*- und *Cb*-Wert gespeichert. Damit wird bereits eine 50%-ge Datenreduktion erreicht. In den folgenden Stufen des Verfahrens wird jede der Komponenten *Y, Cb* und *Cr* getrennt weiterbehandelt.

### Die Diskrete Cosinus-Transformation

In dieser Verfahrensstufe wird das Bild in Teilquadrate mit je 8x8 Pixeln zerlegt, die einzeln mit der diskreten Cosinus-Transformation aus dem Ortsraum in den Ortsfrequenzraum transformiert werden. Dabei entsteht aus der 8x8-Pixel-Matrix eine 8x8-Matrix, deren Komponenten die 64 Frequenzen des Spektums darstellen, aus denen das Bild aufgebaut ist. In Abbildung 8.12 a) und b) ist ein Beispiel-Bild und seine Transformierte dargestellt.

### Die Quantisierung

Wie in Abbildung 8.12 zu erkennen ist, besitzt die Cosinus-Transformierte des Bildes in der oberen, linken Ecke ihre größten Werte. Diese entsprechen den niedrigen Frequenzen im Bild, also den ausgedehnten Strukturen. Nach rechts unten werden die Werte zunehmend kleiner. Die Werte rechts unten entsprechen den höchsten Frequenzen im Bild. Sie besitzen eine wesentlich geringere Amplitude und sind für das Erkennen des Bildinhalts auch weniger wichtig. Sehr viele Matrix-Elemente haben den Wert 0.

Diese Tatsache wird benutzt, um durch eine Quantisierung zu einer Datenreduktion zu gelangen. Im einfachsten Fall werden alle Komponenten der Cosinus-Transformierten zur nächsten ganzen Zahl hin gerun-

det. Andere Verfahren quantisieren die höheren Fequenzen in zunehmend größeren Schritten. Das Ergebnis der Quantisierung ist eine ganzzahlige Matrix, bei der die meisten Komponenten den Wert 0 haben (Abbildung 12.13).

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| 20 | 20 | 20 | 0 | 0 | 20 | 20 | 20 |
| 20 | 20 | 0 | 0 | 0 | 0 | 10 | 20 |
| 20 | 0 | 0 | 20 | 20 | 0 | 0 | 20 |
| 20 | 0 | 0 | 20 | 20 | 0 | 0 | 20 |
| 20 | 20 | 0 | 0 | 0 | 0 | 10 | 20 |
| 20 | 20 | 20 | 0 | 0 | 20 | 20 | 20 |
| 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |

a) Originalbild

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 29.99 | 0.00 | 31.54 | -0.00 | 9.99 | -0.00 | -2.24 | 0.00 |
| -0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | -0.00 | 0.00 |
| 31.54 | 0.00 | -2.85 | 0.00 | -29.00 | 0.00 | 7.07 | -0.00 |
| 0.00 | -0.00 | -0.00 | 0.00 | -0.00 | 0.00 | -0.00 | 0.00 |
| 10.00 | -0.00 | -29.00 | -0.00 | 30.0 | -0.00 | 2.24 | 0.00 |
| -0.00 | 0.00 | -0.00 | 0.00 | -0.00 | 0.00 | -0.00 | 0.00 |
| -2.24 | 0.00 | 7.07 | -0.00 | 2.24 | 0.00 | -17.07 | 0.00 |
| 0.00 | -0.00 | -0.00 | -0.00 | -0.00 | 0.00 | 0.00 | -0.00 |

b) Cosinus-Transformierte

**Abbildung 8.12:** Ein 8x8-Bild und seine Cosinus-Transformierte

### Codierung

Das Matrix-Element in der linken oberen Ecke der Cosinus-Transformierten entspricht dem Mittelwert aller 64 Pixelwerte. Es wird daher als DC-Komponente (direct current component) bezeichnet und die übrigen Matrix-Elemente als AC-Komponenten (alternating current components).

Für die DC-Komponeneten codiert man nur die erste mit ihrem Absolut-Wert, für alle weiteren wird nur ihre Differenz zum Vorgänger codiert, wozu kleinere Zahlenwerte genügen. Die Folge der DC-Komponenten wird dann mit einem Huffman-Algorithmus verlustlos gepackt.

Die AC-Komponenten ordnet man nach dem Diagonalschema der Abbildung 2.13 linear an. Dabei entstehen aufgrund der Matrix-Struktur längere Folgen, die nur Nullen enthalten und mit einer Lauflängen-Codierung gut gepackt werden.

### Weitere Merkmale des JPEG-Methode

Das JPEG-Verfahren kann weitere Techniken einsetzen, z.B.

*Interleaving*
Dabei werden die drei Kanäle des Bildes nicht nacheinander gespeichert bzw. übertragen, sondern Blöcke gebildet, die alle Kanäle eines Bildbereichs enthalten.

– *verlustlose Codierung*
Falls eine vollständige Rekonstruktion des Originalbildes möglich sein soll, werden auch andere Codierungstechniken eingesetzt (nearest neigbors prediction mit Huffman-Codierung).

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *30* | *0* | *32* | *0* | *10* | *0* | *-2* | *0* |
| *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* |
| *32* | *0* | *-3* | *0* | *-29* | *0* | *7* | *0* |
| *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* |
| *10* | *0* | *-29* | *0* | *30* | *0* | *2* | *0* |
| *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* |
| *-2* | *0* | *7* | *0* | *2* | *0* | *-17* | *0* |
| *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* |

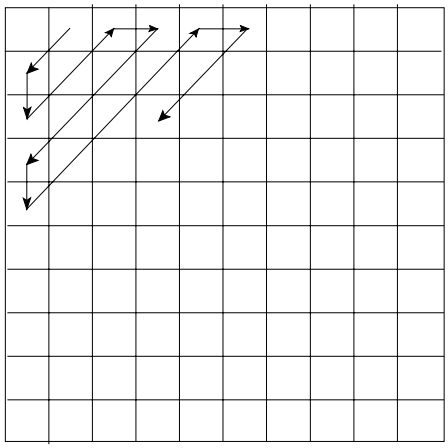


**Abbildung 8.13:** Quantisierte Cosinus-Transformierte und Durchlaufschema für die Lauflängen-Codierung der AC-Komponenten.

JPEG (1000%)

JPEG (20%)

**Abbildung 8.14:** Darstellung eines JPEG-Bildes mit unterschiedlicher Wiedergabequalität
rechtes Bild: JPEG-Kompression auf 20% des Originalumfangs (Ausschnitt)
mittleres Bild: Originalaufnahme
linkes Bild: verlustlose JPEG-Kompression (Ausschnitt)