

Lehrziele

Nach der Durcharbeitung dieses Kapitels sollten Sie

- die verschiedenen Typen von Dateien und Datenbanken beschreiben können,

- die wichtigsten Verfahren nachvollziehen können, die zur Ablage und zum Wiederauffinden von Daten auf Direktzugriffspeichern dienen,
- anhand ausgewählter Anwendungen darlegen können, welche Zugriffsmethoden für die Dateien in Frage kommen und welche auszuschließen sind,

- die Komponenten von Datenbanksystemen kennzeichnen und erläutern können, welche Vorteile Datenbanksysteme gegenüber der herkömmlichen Dateiorganisation haben,
- die Phasen bei der Entwicklung einer Datenbank erläutern können,
- mit einer verbreiteten Methode (Entity-Relationship-Diagramme) das konzeptionelle Datenmodell eines kleinen Beispiel-Informationsystems beschreiben können,
- die wichtigsten, in Datenbanksystemen realisierten Datenmodelle unterscheiden können,
- die Funktionen eines Datenbankverwaltungssystems erläutern können,
- das Tätigkeitsprofil eines Datenbankadministrators darstellen können,
- eine einfache Datenbankabfrage in der Sprache SQL formulieren können,
- die Verarbeitung von Datenbanktransaktionen an einem praktischen Beispiel erklären können,
- das Angebot an externen Datenbanken skizzieren können,
- einen Überblick über aktuelle Entwicklungen der Datenbanktechnik geben können.

In diesem Kapitel werden Sie zunächst die *Grundkonzepte der Datenspeicherung* kennenlernen: Wie Daten gespeichert werden, um sie später je nach Bedarf möglichst schnell und einfach wieder auffinden zu können.

Die Datenorganisation bei Programmen wird durch die *Datenstruktur* (vgl. Abschnitt 9.2.1) festgelegt.

Bei *Assemblerprogrammen* geschieht das durch die Bestimmung der Speicheradressen und die Anzahl der Bytes, die die verwendeten Variablen und Konstanten benötigen. Die *Zusammengehörigkeit* von Datenfeldern wird durch die physische Hintereinanderreihung ausgedrückt. Die Namen, die dabei vergeben werden, sind nichts anderes als symbolische Adressen.

So werden zum *Beispiel* durch den *Assemblerbefehl „TAG DS 2“* für die Variable mit dem Namen „TAG“, der bei diesem Beispiel für die Adresse „23 644“ stehen soll, zwei Bytes mit den Adressen „23 644“ und „23 645“ reserviert.

```
23 643  
23 644 ... TAG  
23 645  
23 646 ... MONAT  
23 647
```

Abb. 14/1: Variablen bei einem Assemblerprogramm

Durch die *Verwendung von Namen* wird einerseits die *Programmierung erleichtert* (Namen merkt man sich besser als große Zahlen) und andererseits das *Programm etwas portabler*. Letzteres heißt, daß bei der Verlegung des Programms in einen anderen Speicherbereich weniger Programmänderungen nötig sind. Bei einer Änderung einer Adresse muß sie so oft geändert werden, wie sie im Programm vorkommt. Bei der Verwendung von Namen ist das für jede Variable und Konstante nur einmal der Fall, was aber *dennoch oft einen enormen Aufwand* bedeutet.

Außerdem wird weder festgelegt, wie die adressierten Bytes interpretiert werden sollen (als Zahlen, Alphabetezeichen usw.), noch ist gewährleistet, daß die Speicherplätze, die für Variablen oder Konstanten wie oben reserviert wurden, nicht fehlerhaft, zum Beispiel durch eine falsche Adressberechnung, verwendet werden.

Diese Schwachpunkte werden mit der Verwendung höherer Programmiersprachen behoben:

- Durch die Kompilierung und das Binden werden erst zu diesem Zeitpunkt den Konstanten und Variablen (zum Beispiel: TAG) die entsprechenden Adressen (zum Beispiel: 23 644) zugewiesen. Diese dynami-

- sche Zuweisung hängt von dem Adressbereich des verwendeten Rechners und dem gewählten Zeitpunkt ab. Den Benutzern eines Rechners wird ja nicht ein fester, sondern der jeweils freie Arbeitsspeicherbereich zugeordnet.
- Die Datenstruktur bestimmt den logischen Zusammenhang einzelner Elemente, die physische Reihenfolge im Arbeitsspeicher ist belanglos.
 - Die zugewiesenen Speicherbereiche werden vor unerlaubtem Zugriff geschützt. So ist es während des gesamten Programmablaues nur möglich, über den Variablennamen auf die ihm zugewiesenen Bytes zuzugreifen. Außerdem wird durch die Verwendung von Typen die Interpretation der Speicherinhalte fixiert. Damit wird verhindert, daß zum Beispiel eine logische Und-Verknüpfung mit einer numerischen Variablen durchgeführt wird oder deren Inhalt nicht als Zahl, sondern als eine aus Ziffern bestehende Zeichenkette interpretiert wird.

Die Probleme der physischen Organisation sind dem Benutzer damit abgenommen. Welche *Probleme* ferner zu bewältigen sind, soll Ihnen eine Gegenüberstellung zweier Datenstrukturen am *Beispiel einer Bibliotheksverwaltung* verdeutlichen.

Abb. 14/2: Vergleich zweier Datenstrukturen (vereinfachtes Anschauungsbeispiel)

Feldbez.	Länge	Typ	Feldbez.	Länge	Typ
Inv.-Nr:	6	numerisch	Inv.-Nr.:	6	numerisch
Titel:	30	alphanumerisch	Titel:	30	alphanumerisch
Autoren			Autor:	20	alphanumerisch
Author 1:	20	alphanumerisch	Datum		
Author 2:	20	alphanumerisch	Tag:	2	numerisch
Author 3:	20	alphanumerisch	Monat:	2	numerisch
Datum:	8	numerisch	Jahr:	4	numerisch

Betrachten Sie die Unterschiede:

1. Datum

Bei der einen Variante ist es möglich, direkt durch die Angabe „Datum.Jahr“ auf das Erscheinungsjahr zuzugreifen. Dadurch lassen sich besonders rasch alle in einem bestimmten Jahr erschienenen Titel auflisten. Bei der anderen Variante muß dieser Zugriff erst durch ein „Zusammensuchen“ der Zeichen, die das Jahr bestimmen, erreicht werden.

2. Autor

Die eine Variante läßt nur einen Autor zu, die andere drei. Bei einem Buch mit nur einem Autor wird dann allerdings Speicherplatz verschwendet.

Dieser Vergleich zeigt folgendes:

- Programme mit denselben Dateninhalten, aber verschiedenen Datenstrukturen sind nur schwer zu kombinieren.
- Die Datenstruktur bestimmt die Attribute eines Objekts, die im Rechner berücksichtigt werden.
- Speicherplatzverschwendungen und auch meistens lange Zugriffszeiten haben ihre Ursachen in einer schlechten Datenstrukturierung.

Das verdeutlicht Ihnen, welche wichtige Rolle die Datenstruktur in einem Programm spielt. Daher sollen anhand des *Bibliotheksbeispiels* weitere Begriffe und Verfahrensweisen erläutert werden, die für die *Datenstrukturierung* von Bedeutung sind.

Übungsaufgabe Nr. 296 im Arbeitsbuch

14.1 Aufbau und Verarbeitung von Dateien

Für die folgenden Überlegungen ist es unerlässlich, den *Einfluß der Datenträger auf die Dateiorganisation* zu betrachten. Es können *zwei grundlegende Speicherformen* unterschieden werden:

1. Sequentielle Speicher und
2. direkt adressierbare Speicher.

Bei sequentiellen Speichern (wie zum Beispiel Magnetband) werden die Datensätze unmittelbar nacheinander abgespeichert und können nur in der gespeicherten Folge verarbeitet werden.

Es ist zum Beispiel nicht möglich, Datensätze zu überspringen oder einmal den nächsten und dann wieder den vorhergehenden Satz zu lesen oder zu schreiben.

Diese Beschränkungen führen zu einer weitgehenden *Verdringung der sequentiellen Speichermedien* zugunsten der direkt adressierbaren. Magnetbänder werden im allgemeinen zur langfristigen Aufbewahrung großer Datenmengen verwendet, für die Archivierung und die Datensicherung, aber kaum mehr zur Speicherung häufig verarbeiter Daten.

Bei direkt adressierbaren Speichern (wie zum Beispiel Zentraleinspeicher, Magnetplatte, Diskette, optische Speicherplatte) kann jeder beliebige Datensatz mit Kenntnis der Adresse sofort gelesen, geändert, gelöscht oder eingefügt werden.

→ Übungsaufgabe Nr. 297 im Arbeitsbuch

Aufbauend auf diese Speicherformen können Sie **zwei Grundtypen der Dateiorganisation** unterscheiden:

1. Sequentiell gespeicherte Datenbestände und
2. direkt adressierbar gespeicherte Datenbestände.

Sequentiell gespeicherte Datenbestände: Sie erlauben nur ein systematisches Durcharbeiten einer Datei von Beginn an. Diese Organisationsform kann auf sequentiellen oder direkt adressierbaren Speichern angewendet werden.

Die einzige mögliche Suchstrategie ist hier das sequentielle Suchen. Die Datei kann also nur Satz für Satz durchsucht werden, bis die gewünschte Information gefunden wird.

Auf unser *Bibliotheksbeispiel* angewandt sind bei 1.000 Büchern im Durchschnitt 500 Suchschritte durchzuführen, bis das gewünschte Buch gefunden ist.

Direkt adressierbar gespeicherte Datenbestände: Sie erlauben einen schnellen Zugriff bei Kenntnis der Adresse. Voraussetzung ist ein direkt adressierbares Speichermedium.

Der direkte Zugriff ist an eine wesentliche Voraussetzung gebunden: Die genaue **Adresse des Datensatzes muß bekannt sein**. Wir wollen dieses Problem anhand von zwei *Beispielen* genauer untersuchen.

1. Beispiel:

In einer *Bibliothek* werden laufend Bücher inventarisiert. Der Bibliothekar legt für jedes Exemplar ein eigenes Karteikärtchen an, auf das er die Inventarnummer, Titel, Autoren, Verlag, Eingangsdatum, Standortnummer und ähnliches mehr einträgt.

Soll das System auf einem Rechner implementiert werden, so ist das ganz einfach. Für jedes Karteikärtchen wird ein Datensatz angelegt, dessen relative Adresse (RA) der Inventarnummer entspricht (siehe Abb. 14.1/1). Der Bibliothekar kann jederzeit leicht über diesen rein numerischen Schlüssel direkt auf jeden Datensatz (= Karteikärtchen) zugreifen. Die Datenstruktur wird in der Abb. 14.1/2 gezeigt.

Bei der relativ adressierten Datei ist der *Primärschlüssel* (die Inventarnummer) nicht in der Datenstruktur und daher *nicht in den Datensätzen* enthalten. Es kann ausschließlich über den numerischen Primärschlüssel auf einen Datensatz direkt zugriffen werden. Der Satz wird durch folgende Berechnung gefunden:

Arfangsadresse des gesuchten Datensatzes = Anfangsadresse der Datei + (relative Adresse - 1) × Länge des Datensatzes

DATEI



Abkürzung: RA = relative Adresse

Abb. 14.1/1: Karteikärtchen – relativ adressierbare Datei

Abb. 14.1/2: Datenstruktur eines Datensatzes in der Bibliotheksverwaltung mit einer relativen Datei (vereinfachtes Anschaubungsbeispiel)

Feldbezeichnung	Länge	Typ
Titel:	30	alphanumerisch
Autor:	20	alphanumerisch
Verlag:	30	alphanumerisch
Eingangsdatum:	8	numerisch
Standort-Nr.:	6	numerisch

Voraussetzung ist in diesem Fall, daß alle Datensätze gleich lang sind.

Zum *Beispiel* kann durch die Eingabe des Wertes „7“ direkt auf den Datensatz „Müller G.“ Informationsstrukturierung in...“ zugegriffen werden. „7“ ist hier die relative Adresse dieses Datensatzes. Gehen wir von den Werten der Abb. 14.1/1 und 2 sowie davon aus, daß die Datei bei der Adresse „23 642“ beginnt, so wird der siebte Datensatz unter der Adresse „24 206“ gefunden [$= 23\ 642 + (7-1) \times (30 + 20 + 30 + 8 + 6)]$.

Übungsaufgabe Nr. 298 im Arbeitsbuch

2. Beispiel:

Der Bibliothekar steigert seine Ansprüche. Er will nun ebenfalls über Autoren, Titel, Schlagwörter und noch vieles mehr zugreifen.

Wie kann dem Bibliothekar geholfen werden? Aus der Anfrage nach einem numerischen Schlüssel ist nun die Anfrage nach einem alphanumerischen Schlüssel (Autor, Titel...) geworden. Die Adresse jedes Datensatzes der Datei ist jedoch zwingend rein numerisch.

←

→ Autoren, Titel...

Neben dem „primitiven“ sequentiellen Suchen (engl.: sequential search), das auch hier angewendet werden kann, gibt es *zwei weitere mögliche Suchstrategien*:

1. Es kann ein Index angelegt werden, der neben allen Schlüsseln noch die Adressen der zugehörigen Datensätze anführt (indizierte Organisation, indexsequentielle Organisation; engl.: indexed organization), oder
2. es kann aus dem Schlüssel die Adresse des Datensatzes berechnet werden (gestreute Organisation, Hash-Verfahren; engl.: hash organization).

Wodurch unterscheiden sich die beiden Beispiele?

Im ersten Fall sucht der Bibliothekar nach einer Karteikarte über einen *numerischen Schlüssel*. Dieser numerische Schlüssel entspricht der *relativen Adresse* des gesuchten Datensatzes. Dieser Datensatz kann also ohne weiteres Suchverfahren sofort gelesen oder geschrieben werden.

Im zweiten Fall ist die Adresse des Datensatzes *nicht bekannt*; der Datensatz muss gesucht werden.

Der Zugriff über einen *numerischen Schlüssel* auf relative Dateien ist in der Praxis *ehn selten* und soll nicht weiter behandelt werden. In der betrieblichen Informationsverarbeitung ist der *Zugriff über alphanumerische Schlüssel von weitaus größerer Bedeutung*. Dieser (sowie selbstverständlich auch der Zugriff über rein numerische Schlüssel) wird durch die nachfolgend behandelten Verfahren der indizierten und gestreuten Organisation ermöglicht.

Wie schon im Abschnitt 9.1.3 erwähnt, werden die Attribute als *Schlüssel* bezeichnet, die dadurch ausgezeichnet sind, daß über sie *direkt zugriffen* werden kann. Bei gleichzeitig identifizierenden Attributen spricht man von „*Primärschlüsseln*“. Bei Attributen, die keine identifizierende Eigenschaft haben, aber einen direkten Zugriff erlauben, verwendet man den Begriff „*Sekundärschlüssel*“.

An dieser Stelle muß darauf hingewiesen werden, daß die Verwaltung der Daten, also auch der Zugriff über Schlüssel auf einen Datensatz, Funktionen beinhaltet, die von der *Systemsoftware* übernommen werden. Jedoch ist es für jeden, der sich auch nur entfernt mit Programmierung beschäftigt, unumgänglich, sich mit dieser Problematik auseinanderzusetzen.

a) Indizierte Organisation

INDEXDATEI

INDEXDATEI	RA	INR	Autor	Titel
Date C.J.	4	1	4711	Hansen H.R.
Elson M.	6	2	3812	Maurer H.
Hansen H.R.	1	3	1029	Wirth N.
Maurer G.	2	4	1744	Date C.J.
Müller G.	7	5	1222	Wedekind H.
Wedeckind H.	5	6	1313	Elson M.
Wirth N.	3	7	4712	Müller G.

HAUPTDATEI

HAUPTDATEI	RA	INR	Autor	Titel
Date C.J.	1	4711	Hansen H.R.	Wirtschaftsinformatik I ...
Elson M.	2	3812	Maurer H.	Datenstrukturen und ...
Hansen H.R.	3	1029	Wirth N.	Algorithmen und Datenstrukturen ...
Maurer G.	4	1744	Date C.J.	An Introduction to Database Systems ...
Müller G.	5	1222	Wedekind H.	Datenbanksysteme ...
Wedeckind H.	6	1313	Elson M.	Data Structures ...
Wirth N.	7	4712	Müller G.	Informationsstrukturierung in ...

b) Gestreute Organisation

HAUPTDATEI

HAUPTDATEI	RA	INR	Autor	Titel
Date C.J.	1	4711	Hansen H.R.	Wirtschaftsinformatik I ...
Elson M.	2	3812	Maurer H.	Datenstrukturen und ...
Hansen H.R.	3	1029	Wirth N.	Algorithmen und Datenstrukturen ...
Müller G.	4	1744	Date C.J.	An Introduction to Database Systems ...
Wedeckind H.	5	1222	Wedekind H.	Datenbanksysteme ...
Elson M.	6	1313	Müller G.	Data Structures ...
Wirth N.	7	4712		Informationsstrukturierung in ...

Hash-Funktion

Abkürzungen: AV = Adreßverweis; INR = Inventarnummer; RA = relative Adresse

Abb. 14.1/3: Gegenüberstellung (a) der Organisation mit Index und (b) der gestreuten Organisation

Anmerkungen:

Bei beiden Strategien ist die Inventarnummer (zum Beispiel 4711, 3812...) im Gegensatz zur relativ adressierten Datei Bestandteil der Datenstruktur und daher der Datensätze. Sie muß hier keinesfalls der relativen Adresse entsprechen. Somit muß diese auch nicht forthlaufend vergeben werden. So ist es möglich, zum Beispiel den Aufbewahrungsort des Buches in der Inventarnummer genauer zu spezifizieren (1xxx könnte für die erste Buchstelllage stehen, 2xxx für die zweite und so fort).

zu a) Der Adreßverweis stellt die Verbindung zwischen Indexdatei und Hauptdatei dar. Wird im Index ein Schlüssel gefunden, kann über diesen Adreßverweis auf den zugehörigen Satz der Hauptdatei zugegriffen werden.

zu b) Bei der gestreuten Organisation wird der Suchschlüssel als Eingabeparameter für den Algorithmus der Hash-Funktion verwendet, der die relative Adresse des diesem Schlüssel gehörenden Datensatzes liefert.

14.1.1 Indizierte Organisation

Unter einem **Index** (engl.: index) versteht man in der Informationsverarbeitung eine Hilfsdatei, deren Datensätze neben den Schlüsseln der Hauptdatei die Adressen der zu diesen Schlüsseln gehörenden Datensätze beinhalten (= Adreßverweise).

Abb. 14.1.1/1: Index- und Hauptdatei

HAUPTDATEI					
Schlüssel	AV	RA	INR	Autor	Titel
Date C.J.	4	1	4711	Hansen H.R.	Wirtschaftsinformatik ...
Eison M.	6	2	3812	Maurer H.	Datenstrukturen und ...
Hansen H.R.	1	3	1029	Wirth N.	Algorithmen und Datenstrukturen ...
Maurer 4.	2	4	1744	Date C.J.	An introduction to Database Systems ...
Müller G.	7	5	1222	Weeckind H.	Datenbanksysteme ...
Weeckind H.	5	6	1313	Eison M.	Data Structures ...
Wirth N	3	7	4712	Müller G.	Informationsstrukturierung in ...

Abkürzungen: AV = Adreßverweis; INR = Inventarnummer; RA = relative Adresse

Abb. 14.1.1/2: Index- und Hauptdatei

Der *einfachste Fall* der Dateiorganisation mit einem Index ist der, daß die Schlüssel in der Reihenfolge, in der sie erfaßt werden, in den Index eingetragen werden.

Das Ergebnis ist ein *unsortierter Index*. Bei jedem Suchvorgang muß der gesamte Index durchgelesen werden, da das System nicht „wissen“ kann, an welcher Stelle in der Indexdatei der gesuchte Schlüssel aufzufinden ist.

Das Ziel jeder Dateiorganisation ist – wie schon weiter oben erwähnt – ein möglichst schneller Zugriff mit möglichst wenig Aufwand. Daher werden Sie jetzt vielleicht fragen: „Warum überhaupt ein Index?“ Sie wissen bereits, daß bei jedem Schreib- und Lesezugriff ein konstant großer Datenteil gelesen oder geschrieben werden kann. Wenn die Schlüssel einer Datei in einen Index ausgelagert werden, können bei *jedem Zugriff mehrere Schlüssel eingelesen* werden, und die Dauer des Suchprozesses wird dadurch verkürzt.

An einem konkreten Beispiel soll die kürzere Suchzeit demonstriert werden. Ein Datensatz der Hauptdatei hat genau die Länge eines Sektors auf einer Diskette (256 Zeichen). Der alphanumerische Schlüssel ist vier Zeichen lang. Die Datei besteht aus 1.000 Datensätzen, daher kann die Addressierung über vier Zeichen erfolgen. Die Datensätze der Indexdatei haben folgenden Aufbau:

Schlüssel 4 Zeichen
Adresse 4 Zeichen

Summe 8 Zeichen

Die Dauer des Suchvorganges wird hauptsächlich von der Anzahl der Diskettenzugriffe zum Auslesen eines Sektors bestimmt (Zugriffszeiten im Arbeitsspeicher sind für unser Beispiel vernachlässigbar). Sowohl die Hauptdatei, als auch die Indexdatei sind unsortiert.

Wie lange dauert das Auffinden eines Datensatzes,

- a) wenn in der Hauptdatei gesucht wird?
- b) wenn zuerst in der Indexdatei gesucht wird?

Lösung:

- a) Alle Datensätze der Hauptdatei müssen gelesen werden, das entspricht also 1.000 Zugriffen (ein Datensatz entspricht ja genau einem Sektor).
- b) In einem Sektor mit 256 Zeichen passen maximal 32 Indexeinträge mit je acht Zeichen ($256/8 = 32$). Das bedeutet, daß sämtliche Indexeinträge für 1.000 Datensätze in 32 Sektoren Platz haben.

1.000 Datensätze / 32 Eintragen je Sektor = 31,25 Sektoren
Mit dem Auslesen des gesuchten Datensatzes aus der Hauptdatei sind nur 33 Zugriffe notwendig.

Die Zugriffszeit kann also durch die Indizierung erheblich reduziert werden (in unserem Beispiel um den Faktor 30).

Obiges Beispiel soll nur die Grundidee der indizierten Organisation beschreiben. Es muß aber darauf hingewiesen werden, daß dies eine triviale Form ist, die in der Praxis nirgendwo angewendet wird. Es gibt wesentlich schnellere Zugriffsverfahren über einen Index. Diese *Verbesserung* kann dadurch bewirkt werden, daß die *Indexeintragungen* nicht einfach in der Reihenfolge ihrer Erfassung, sondern in *einer bestimmten Sortierung* eingebracht werden. Wie diese Indexorganisation aussiehen kann, wird in den nächsten Abschnitten gezeigt.

Übungsaufgabe Nr. 300 im Arbeitsbuch

14.1.1.1 Indizierte Organisation mit physisch sortiertem Index

In einem *physisch sortierten Datenbestand* entspricht die Reihenfolge der Speicherung der Sortierreihenfolge.

In diesem Abschnitt werden die *Suchverfahren in solchen physisch sortierten Datenbeständen* erklärt. Auf die Prinzipien der wichtigsten Sortierverfahren, also der Verfahren, die notwendig sind, um einen sortierten Index zu erzeugen, soll hier nicht näher eingegangen werden.

Nehmen wir als anderes Beispiel das Telefonteilnehmerverzeichnis. In größeren Städten werden mehrere Telefonbücher geführt. So wird etwa in Wien für die Buchstaben A bis H, I bis Q und R bis Z jeweils ein Teilband verwendet. Man könnte nun den Suchvorgang in den drei Teilbänden als 3-Weg-Suchen betrachten; jeder beginnt in jenem Telefonbuch zu suchen, das den gesuchten Namen und die gesuchte Telefonnummer beinhaltet.

Die Qualität des *m-Weg-Suchens* ist stark von der gewählten Blockgröße abhängig. Im Grenzfall, bei Blockgröße 1, entspricht das m-Weg-Suchen dem sequentiellen Verfahren.

Allgemein kann gesagt werden, daß mit der Hilfe des *binären Suchens* in größeren Datenbeständen nach den wenigsten Vergleichen das gesuchte Element aufgefunden wird.

Wie erwähnt sind obige Suchverfahren nur dann anzuwenden, wenn der Datenbestand physisch sortiert ist, das heißt wenn die Schlüssel der Datensätze in der Indexdatei eine auf- oder absteigende Folge bilden.

Was passiert, wenn ein *neuer Datensatz in eine Datei aufgenommen wird*? In der Regel wird für diesen Fall auf dem Speichermedium *für den Index ein Überlaufbereich* angelegt, in den die Indexeintragungen der hinzukommenden Sätze in der Reihenfolge ihrer Erfassung eingetragen werden. Wird der gesuchte Schlüssel im sortierten Bereich nicht gefunden, so wird im unsortierten Überlaufbereich sequentiell weitergesucht. Die Dauer des gesamten Suchvorganges wird somit von der Größe des Überlaufbereiches wesentlich beeinflußt. Es ist unumgänglich, daß der *Index von Zeit zu Zeit reorganisiert* wird, das heißt, daß der Indexdatenbestand neu sortiert wird, daß ein neuer Überlaufbereich angelegt wird usw.

Übungsaufgabe Nr. 302 im Arbeitsbuch

→

14.1.1.2 Indizierte Organisation mit logisch sortiertem Index

Wir haben festgestellt, daß das Einfügen in physisch sortierte Datenbestände mit dem Problem des Reorganisierens verbunden ist. Wind es selten reorganisiert, verlängern sich die Zugriffszeiten durch das langsame Suchen im Überlaufbereich. Wird oft reorganisiert, benötigen die Reorganisationsläufe erhebliche Rechenzeiten. Es gibt aber auch Organisationsformen, bei denen Reorganisationen nicht mehr notwendig sind.

Ketten

Bei einer **Kette** (engl.: chain) ist in jedem Datensatz die Adresse (*Zeiger*; engl.: pointer) des in der Sortierreihenfolge nachfolgenden Satzes gespeichert. In diesem Fall entspricht der physische Nachfolger normalerweise nicht dem logisch nachfolgenden Element.

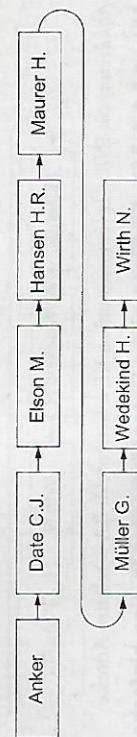
Der Zeiger, der auf den ersten Datensatz zeigt, wird *Anker* genannt.

Der Zeiger des letzten Datenelementes, also jenes Datensatzes, der keinen logischen Nachfolger hat, ist eine *Endemarke*.

Physische Reihenfolge

INDEXDATEI					HAUPTDATEI		
RA	Schlüssel	KF	INR	Autor	INR	Autor	Titel
1	Hansen H.R.	2	4711	Hansen H.R.	Wirtschaftsinformatik I ...		
2	Maurer H.	7	3812	Maurer H.	Datenstrukturen und ...		
3	Wirth N.	-	1029	Wirth N.	Algorithmen und Datenstrukturen ...		
4	Date C.J.	6	1744	Date C.J.	Ein Introduction to Database Systems ...		
5	Wedekind H.	3	1222	Wedekind H.	Datenbanksysteme ...		
6	Elson M.	1	1313	Elson M.	Data Structures ...		
7	Müller G.	5	4712	Müller G.	Informationsstrukturierung in ...		

Logische Reihenfolge



Abkürzungen: INR = Inventarnummer; KF = Kettenfeld; RA = relative Adresse

Abb. 14.1.1.2/1: Einfache Kette

Anmerkung:

Bei dieser Lösung entspricht die relative Adresse (RA) jeder Indexeintragung der relativen Adresse des dazugehörigen Datensatzes der Hauptdatei. Die Zahl im Kettenfeld (KF) ist die relative Adresse der logisch nachfolgenden Indexeintragung, der Strich („-“) ist eine Endemarke.

Der Anker wurde in diesem Fall durch ein Feld mit dem Inhalt „4“ (= relative Adresse der alphabetisch kleinsten Indexeintragung) realisiert werden.

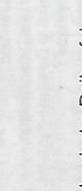
Wird ein *neues Element hinzugefügt*, so wird es physisch an das Ende des Datenbestandes geschrieben. Der Zeiger des nächstkleineren Elementes (Vorgänger; engl.: predecessor) wird auf das neu eingefügte gerichtet,

das neue Element erhält als Zeiger die Adresse des nächstgrößeren Elements (Nachfolger; engl.: successor).

Wird ein Datensatz *gelöscht*, so wird nur der Zeiger des Vorgängerdatensatzes verändert, und der Speicherplatz des gelöschten Datensatzes wird überschreibbar gemacht.

Physische Reihenfolge

INDEXDATEI		HAUPTDATEI		
RA	Schlüssel	KF	INR	Autor
1	Hansen H.R.	8	4711	Hansen H.R.
2	Maurer H.	7	3812	Maurer H.
3	Wirth N.	-	1029	Wirth N.
4	Date C.J.	6	1744	Date C.J.
5	Wedekind H.	3	2222	Wedekind H.
6	Eison M.	1	3313	Eison M.
7	Müller G.	5	4112	Müller G.
8	Knuth D.E.	2	1000	Knuth D.E.



Abkürzungen: INR = Inventarnummer; KF = Kettenfeld; RA = relative Adresse

Abb. 14.1.1.2/2: Einfügen eines Datenelementes in eine einfache Kette

Übungsaufgabe Nr. 303 im Arbeitsbuch

Der wesentliche *Vorteil von Ketten* liegt im einfachen Änderungsdienst. Dem steht jedoch der *Nachteil* gegenüber, daß bei dieser Organisationsform auch auf direkt adressierbaren Speichermedien nur sequentielles Suchen möglich ist. Die gekettete Organisationsform findet *vor allem im Arbeitsspeicher* Anwendung, wo infolge der hohen Zugriffs geschwindigkeit die Nachteile des sequentiellen Suchens weniger ins Gewicht fallen. Bei externen Speichern muß damit gerechnet werden, daß jedes Kettungselement in einem anderen Sektor beziehungsweise Block liegen kann; die Folge ist ein entsprechend langer Suchvorgang.

Baumstrukturen

Baumstrukturierte Dateiorganisationsformen gewinnen immer mehr an Bedeutung. Die in der Literatur vorgeschlagenen Verfahren sind überaus zahlreich. Es können hier in der Folge *nur die wichtigsten grundlegenden Konzepte* dargelegt werden.

Eine Baumstruktur (engl.: tree) besteht aus einer Menge von Knoten (engl.: vertices) (= eigentliche Information) und Kanten (engl.: edges) (= Adressinformation). Jede Kante zeigt auf einen Knoten. Die Baumstruktur weist drei *Eigenschaften* auf:

1. Es gibt genau einen Knoten, der keinen Vorgänger hat; dieser Knoten ist die Wurzel (engl.: root) dieser Struktur.
2. Jeder Knoten, außer der Wurzel, hat genau einen unmittelbaren Vorgänger.
3. Zu jedem Nichtwurzelknoten gibt es genau einen Weg von der Wurzel zu diesem Knoten.

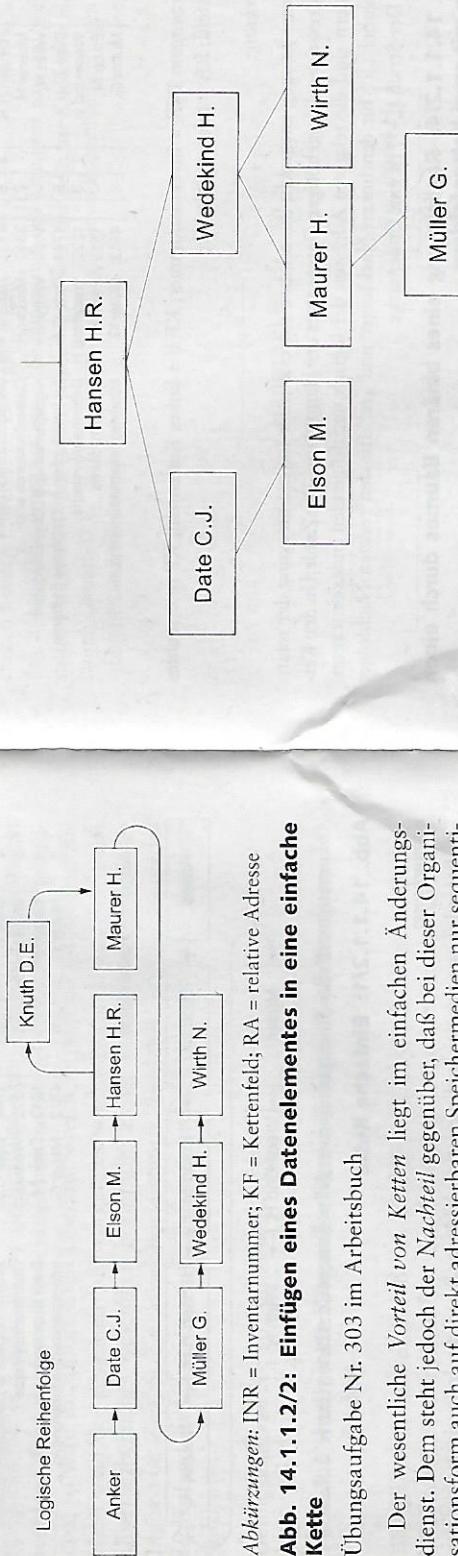


Abb. 14.1.1.2/3: Baumstruktur

Wie Sie aus Abb. 14.1.1.2/3 entnehmen können, werden Baumstrukturen gewöhnlich verkehrt herum gezeichnet. Wenn Sie diese Seite um 180 Grad drehen, dann zeigt die Wurzel wirklich nach unten. Ehe wir auf Baumorganisationen näher eingehen, sind noch einige *Begriffe* zu erklären.