Data Structures and Programming Techniques

HERMAN A. MAURER

Data Structures and Programming Techniques

HERMAN A. MAURER
University of Karlsruhe

Translated by
Camille C. Price
Stephen F. Austin State University

Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632

Library of Congress Cataloging in Publication Data

Maurer, Hermann A 1941–
Data structures and programming techniques.

Translation of Datenstrukturen und Programmierverfahren.
Bibliography: P.
Includes index.
1. Electronic digital computers—Programming.
2. Data structures (Computer science) I. Title.
QA76.6 M38313 001.6'42 76-22758
ISBN 0-13-197038-0

A translation of Dr. Maurer's
Datenstrukturen und Programmierverfahren

The sole authorized English translation of
the original German edition published in
the series *Leitfäden der Angewandten*
Mathematik und Mechanik, edited by Dr. Gortler.

Englewood Cliffs, New Jersey 07632

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

Preface

This book is a translation from the German of a book which was developed from a course conducted by Professor Dr. H. Maurer at the University of Karlsruhe. The lecture notes of the course were thoroughly revised and expanded, in collaboration with H. W. Six, into the final form which was published as *Datenstrukturen und Programmierverfahren.*

In this book, techniques will be presented which allow data to be organized in ways suited to problem solving using a digital computer. Such techniques can contribute to the design of efficient, well-arranged programs and therefore ought to be thoroughly familiar to every computer scientist.

The book is self-contained and, except for an acquaintance with some programming language, no specific knowledge of mathematics or computer science is presupposed.

A few remarks are in order on the notational conventions followed in the book. The notations introduced in Sections 1.2 and 1.3 and used consistently thereafter permit a uniform and precise description of the subject matter. The presentation is, however, sufficiently straight-forward so that the book can be used by the casual reader without requiring a careful study of the notational scheme.

Program segments are used throughout the book to provide a description of the algorithms and to illustrate the manipulation of data structures. The use of an assembler language would allow immediate access to the individual memory locations, however because of the unavailability of a widely-known standard assembler language, and for reasons of convenience, a high-level language is used instead. A very simple subset of the PL/I language—actually only a small extension of the Fortran language—is used so that a reader who is only slightly familiar with programming languages can easily follow the programs. The author wishes to thank Miss S. Reiniger for testing the PL/I programs.

Nacogdoches, Texas Camille C. Price

Contents

Introduction

The efficiency, clarity, and complexity of a program for the solution of a given problem depends substantially on the organization imposed on the data to be used, i.e., on the selected data structure. It is the purpose of this book to present the most important data structures, to explain under what conditions which data structures can be used most effectively, and to indicate which programming techniques are most desirable for working with a particular data structure.

In the study of data structures, it is necessary not only to define the concept of a data structure itself but also to state how the data structure can be stored and manipulated. With this purpose in mind, several considerations are appropriate. First, using the general definition of the concept of a data structure, a method will be given for representing a data structure graphically. Then a model of computer memory will be introduced, along with a way of graphically representing the memory locations. With this foundation established, methods of storing data structures can be presented. The actual descriptions of algorithms for manipulating data structures stored in a computer are given using PL/I programs in which the memory locations in the computer memory model are realized using declarations of PL/I variables.

The general foundations referred to are the subject of Chapter 1. Chapter 2 is dedicated to common and simple data structures known as *lists*. In Chapter 3 the particularly important tree structures are emphasized, a topic often treated too lightly. Then in Chapter 4 two types of complex data structures are introduced.

1 A Model for the Manipulation of Data Structures

Section 1.1 Mathematical Preliminaries

Definition 1.1.1 (Sets and Set Operations) If a set M consists of the n elements $a_1, a_2, \ldots, a_n$, where $n \geqslant 0$, then the set is said to be *finite* and is written $M = \{a_1, a_2, \ldots, a_n\}$. If all the elements of the set M have property P, then M may be written as $\{x \mid x \text{ has property P}\}$. The set containing no elements is called the *empty* set and is denoted by $\varnothing$. A set that is not finite is said to be *infinite*.

If M is a set, and a is an element of M, then we write $a \in M$; if a is not an element of M, we indicate this by $a \notin M$. If M and N are sets, then M is a *subset* of N, written $M \subseteq N$ or $N \supseteq M$, if every element of M is also an element of N. The set M is called a *proper* subset of N, written $M \subset N$ or $N \supset M$, if $M \subseteq N$ and there is at least one element in N that is not an element of M. Two sets M and N are said to be equal, written $M = N$, if $M \subseteq N$ and $N \subseteq M$.

The *union*, *intersection*, and *difference* of two sets M and N, written $M \cup N$, $M \cap N$, and $M - N$, respectively, are defined as follows:

$$M \cup N = \{x \mid x \in M \text{ or } x \in N \text{ or both}\}$$

$$M \cap N = \{x \mid x \in M \text{ and } x \in N\}$$

$$M - N = \{x \mid x \in M \text{ and } x \notin N\}.$$

Two sets are said to be *disjoint* if their intersection is empty. If M is a set, then the sequence $M_1, M_2, \ldots, M_n$ of pairwise disjoint sets is called a *partitioning* of M if $M = M_1 \cup M_2 \cup \cdots \cup M_n$.

Definition 1.1.2 (Tuples and Sequences) An n-tuple A of elements $a_1, a_2, \ldots, a_n$, where $n \geqslant 0$, is a sequence of elements of length n and is written either as $A = (a_1, a_2, \ldots, a_n)$ or as $A = a_1, a_2, \ldots, a_n$. Suppose $A = (a_1, a_2, \ldots, a_n)$ is an n-

tuple; then if $b = a_i$, we call b the *ith component* of A, where i is the position of b in A. We write this symbolically as $\pi_A b = i$ or simply $\pi b = i$.

The notational conventions for sets are often applied to sequences. In particular, we speak of an empty sequence (a null-tuple) $\varnothing$; we write $a \in A$ or $a \notin A$, depending on whether a is or is not in the sequence A; and we say two sequences A and B are disjoint if $\{x \mid x \in A\} \cap \{x \mid x \in B\} = \varnothing$. A sequence B is a subsequence of a sequence A if B is formed by the removal of n elements from A, where $n \geqslant 0$. (Thus, 3, 4, 2 is a subsequence of 6, 3, 0, 12, 4, 2, 7 but not of 6, 4, 0, 0, 12, 3, 7, 2). Two sequences A and B are *equal*, written $A = B$, if A is a subsequence of B and vice versa.

Definition 1.1.3 (Cartesian Products and Relations) The Cartesian product of two sets M and N, written $M \times N$, is defined to be a set of ordered pairs as follows:

$$M \times N = \{(x, y) \mid x \in M \text{ and } y \in N\}.$$

If M and N are sets, then every subset of $M \times N$ is called a *relation* on $M \times N$ (or on M if $M = N$).

Let r be a relation on M. If $(a, b) \in r$, then a is called the *predecessor* of b, and b is the *successor* of a, relative to r. To specify that $(a, b) \in r$, we write a r b; and if $(a, b) \notin r$, we write $a \not r b$.

The *domain* and *range* of a relation r are written symbolically domain (r) and range (r) and are defined as:

$$\text{domain } (r) = \{x \mid (x, y) \in r\}$$

and

$$\text{range } (r) = \{y \mid (x, y) \in r\}.$$

The *inverse* of a relation r, written r^{-1}, is defined by

$$r^{-1} = \{(y, x) \mid (x, y) \in r\}.$$

A relation r on M is called *reflexive* if, for every $a \in M$, $(a, a) \in r$. The relation is called *antireflexive* if $(a, a) \in r$ does not hold for any $a \in M$. A relation is *symmetric* if $(a, b) \in r$ whenever $(b, a) \in r$; a relation is *transitive* if $(a, c) \in r$ whenever $(a, b) \in r$ and $(b, c) \in r$.

Definition 1.1.4 (Basis of a Transitive Relation) Let r be a transitive relation on M, and $b \subseteq r$. The relation b is called a *basis* of r and r is called the *transitive hull* of b if the following holds:

$$r = \{(x, y) \mid \text{there are elements } x_0, x_1, x_2, \ldots, x_n \text{ in M, where } n \geqslant 1, \text{ such that (a) } x_0 = x, \text{ (b) } x_n = y, \text{ and (c) } (x_{i-1}, x_i) \in b \text{ for } i = 1, 2, \ldots, n\}.$$

Definition 1.1.5 (Equivalence Relations and Equivalence Classes) A relation r on M is called an *equivalence relation* if r is reflexive, symmetric, and transitive. If r is an equivalence relation and $(a, b) \in r$, then a and b are said to be *equivalent*.

If M is a set with equivalance relation r, then M is the union of pairwise disjoint sets. These sets are called *equivalence classes.*

Definition 1.1.6 (Partial Ordering and Topological Sorting) A relation r on M is called a *partial ordering* if r is transitive and antireflexive. If $A = a_1, a_2, \ldots, a_n$ is a sequence of elements of M and if $(a_i, a_j) \in r$ for $i < j$, then A is *topologically sorted relative to r.*

Note: It should be observed that in every finite set with partial ordering r there exists at least one element having no predecessor and at least one element having no successor.

Definition 1.1.7 (Total Ordering and Sorting) A relation r on M is a *total ordering* if it is transitive and if conditions a and b below are satisfied:

(a) If $(a, b) \in r$ and $(b, a) \in r$, then $a = b$.
(b) For any two elements a and b, either $(a, b) \in r$ or $(b, a) \in r$, or both.

A sequence $A = a_1, a_2, \ldots, a_n$ of elements from M is *sorted* or *ordered* (relative to r) if $(a_i, a_{i+1}) \in r$ for $1 \leqslant i \leqslant n - 1$.

Definition 1.1.8 (Minimum and Maximum) If M is a set of numbers, then if there exists a smallest number in M, it is called the *minimum* of M and is denoted min (M). Likewise max (M) denotes the *maximum* of M and $\max(M) = -\min\{-x \mid x \in M\}$.

Definition 1.1.9 (Function) A relation f on $M \times N$ is called a *function* from M to N, written symbolically $f: M \longrightarrow N$, if for every $a \in M$ there is exactly one $b \in N$ such that $(a, b) \in f$. Instead of $(a, b) \in f$ we often write $f(a) = b$ or simply $fa = b$. A function $f : M \longrightarrow N$ is called *one-to-one* if whenever $a \neq b$, it follows that $f(a) \neq f(b)$.

Exercises

1.1.1 Let r be the relation:

$$r = \{(1, 2), (2, 3), (3, 5), (3, 4), (1, 3), (1, 4), (2, 4), (2, 5), (1, 5)\}.$$

(a) Show that r is transitive.
(b) Is r an equivalence relation?
(c) Determine a basis b on r with four elements.

1.1.2 Let r be a transitive relation with basis b:

$$b = \{(1, 4), (2, 5), (5, 8), (0, 3), (3, 6), (6, 9), (4, 1), (8, 2), (9, 0)\}.$$

(a) Is r an equivalence relation?
(b) Determine the range of r.

1.1.3 Let r be the relation:

$$r = \{(3, 1), (3, 2), (4, 6), (4, 7), (5, 4), (5, 6), (5, 7), (6, 7)\} \text{ on } M = \{1, 2, 3, 4, 5, 6, 7\}$$

(a) Show that r is a partial ordering.
(b) Determine a topologically sorted sequence of elements of M.

Section 1.2 Data Structures and Ways of Representing Data Structures

Computers are used extensively in the processing of collections of data. A data collection may be thought of as a collection of data *elements* or *nodes* that have n-tuples of character strings as values. The data elements of a data collection can be analyzed and altered in various ways using a computer.

Some typical examples of values of nodes are triples of character strings, representing the name, quantity on hand, and identification number of a part of a piece of machinery; pairs of numbers defining the arguments and corresponding function values of a certain function; 4-tuples of character strings indicating the name, address, bank affiliation, and serial number of a contractor; and so on.

It is often of interest to know how the nodes in a data collection are related to one another. For example, where nodes represent parts of a piece of machinery, it may be meaningful to indicate that one part is a constituent part of another or that all parts with certain part numbers are to be handled together when reordering is done. Where nodes define function values, each node may be associated with the node having the next largest function value. If the nodes represent contractors, it may be useful to indicate that certain ones are working in the same city.

Connections between nodes of a data collection may be indicated by using relations. The combination of nodes and relations is called a *data structure*.

Definition 1.2.1 (Data Structure) A *data structure* B is a pair $B = (K, R)$, where K is a finite set of nodes and R is a finite set of relations on K. The value[1] of a node $k \in K$ is denoted by ωk and is an n-tuple, $n \geq 0$, of character strings; $\omega_i k$ denotes the ith component of the (value of the) node k. This is expressed symbolically as $\omega k = (\omega_1 k, \omega_2 k, \ldots, \omega_n k)$.

Definition 1.2.2 (Notational Conventions) Let $B = (K, R)$ be a data structure, with $r \in R$ a relation and k, k′ nodes. If $(k, k') \in r$, then we say that k′ is a successor of k, k is a predecessor of k′, k and k′ are consecutive, and k points to k′ (all relative to r). If there is no node k′ such that $(k, k') \in r$, then k is called an *end node* (relative to r); if there is no node k′ such that $(k', k) \in r$, then k is called a *start node* (relative to r). Otherwise k is an *inner node*. If k is a node and $\omega k = \varnothing$ (i.e., k is a null-tuple), then k is called a *pointer*. A pointer which points to a start node is called a *start pointer*

[1] Where no misunderstanding is likely, no distinction will be made between a node and the value of the node.

and one which points to an end node is called an *end pointer*. A data structure B is said to be *manipulated* if the value of a node is determined or changed or if a node is added to or removed from B.

Definition 1.2.3 (Graphical Representation of Data Structures) Let $B = (K, R)$ be a data structure. A graphical representation of B is obtained as follows. For every node $k \in K$, a figure is drawn: a circle if $\omega k = \varnothing$ and a rectangle containing ωk otherwise. If a node k points to another node k′ (relative to a relation r), then the figures corresponding to k and k′ are connected by a directed line segment. For different relations we choose different types of line segments. Sometimes it is useful to assign names to the figures corresponding to the nodes; such names are written outside the figures.

Example 1.2.1 The data structure $B = (K, R)$ with 10 nodes $K = \{k_1, k_2, \ldots, k_{10}\}$ and two relations $R = \{T, N\}$, where

$$\omega k_1 = (\text{machine}, 40, 612)$$
$$\omega k_2 = (\text{motor}, 2, 802)$$
$$\omega k_3 = (\text{block}, 3, 105)$$
$$\omega k_4 = (\text{filter}, 2, 117)$$
$$\omega k_5 = (\text{tank}, 10, 118)$$
$$\omega k_6 = (\text{brakes}, 60, 230)$$
$$\omega k_7 = (\text{ignition}, 30, 408)$$
$$\omega k_8 = (\text{housing}, 17, 507)$$
$$\omega k_9 = (\text{frame}, 40, 230)$$
$$\omega k_{10} = (\text{plates}, 1, 702)$$
$$T = \{(k_1, k_2), (k_1, k_8), (k_2, k_3), (k_2, k_7), (k_3, k_4), (k_3, k_5), (k_3, k_6), (k_8, k_9), (k_8, k_{10})\}$$
$$N = \{(k_{10}, k_4), (k_4, k_2), (k_2, k_3)\}$$

is displayed graphically in Fig. 1.2.1. Since, for example, $(k_3, k_5) \in T$ but $(k_3, k_8) \notin T$, there is a line, marked with T, from node k_3 to node k_5 but no line from k_3 to k_8.

The given data structure can be thought of as a rough description of a certain *machine*. Each node k corresponds to a particular part: $\omega_1 k$ is understood to be the name, $\omega_2 k$ is the number of parts on hand, and $\omega_3 k$ represents the part identification number. The relation T explains the composition of the piece of equipment, namely that it consists of two main components: the *motor* and the *housing*, the latter consisting of the *frame* and the *plates*, and so on. The relation N applies to the nodes k, where $\omega_2 k \leqslant 3$, that is, the parts with small quantities on hand. Note that $\omega_2 k \leqslant \omega_2 k'$, where k is a predecessor of k′ relative to N.

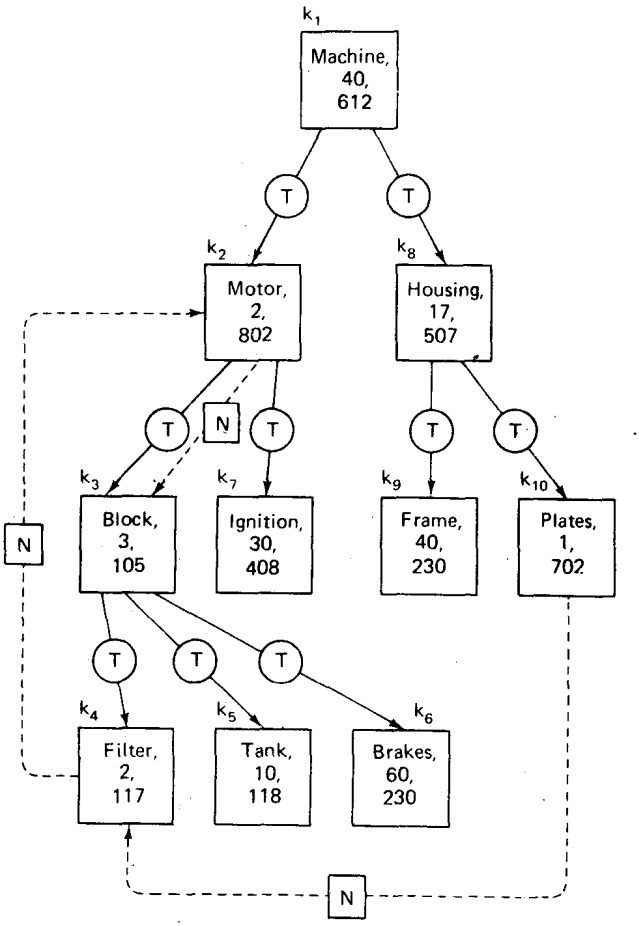

Figure 1.2.1

Figure 1.2.2 differs from Fig. 1.2.1 in the addition of a start pointer st (relative to T) and a start pointer sn (relative to N). This data structure describes not only the nodes $k_1, k_2, \ldots, k_{10}$ but also the nodes st and sn, where $\omega st = \omega sn = \varnothing$; in addition to the relations T and N, it also shows the relations $p_T = \{(st, k_1)\}$ and $p_N = \{(sn, k_{10})\}$.

In actual practice, the description of a piece of equipment such as the one just discussed would necessitate a significantly more complicated data structure; not only would the number of nodes be greater and the value of each node consist of more components, but also additional relations to relate, for example, price and terms of a contract, would be of interest. It is sometimes more convenient to describe a situation using several data structures, each having only a few relations, instead of one

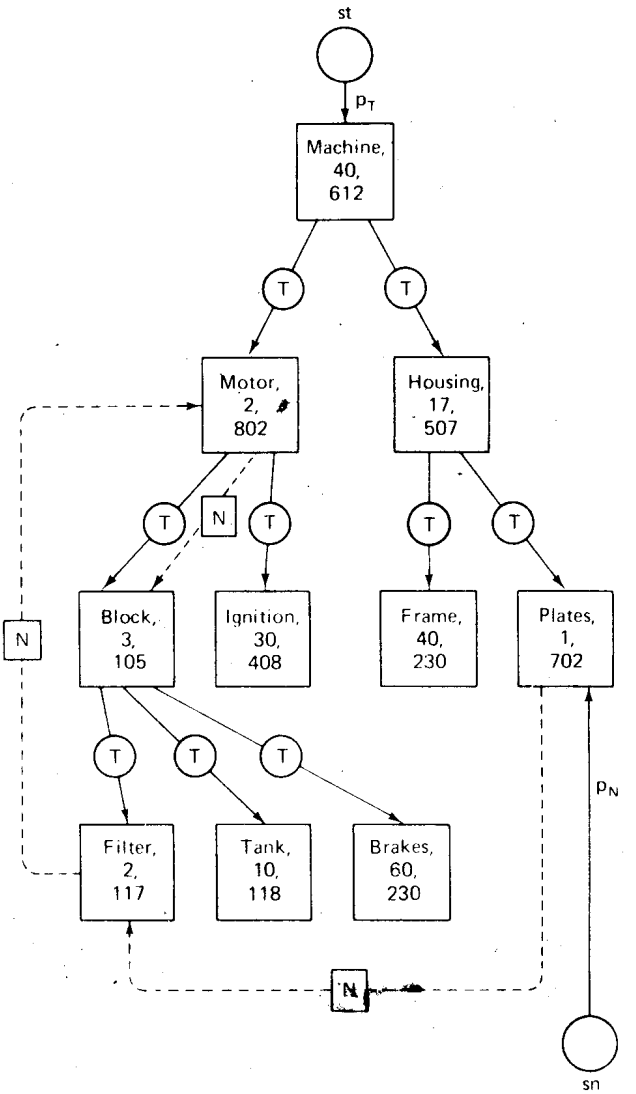

Figure 1.2.1

data structure having many relations. With this in mind, our further investigation of data structures will be restricted to those with a small number of relations.

Remarks

There is no standard definition for the notion of a *data structure*. The definition given here, using relations, is one of several in which relations play an important role.

Exercises

1.2.1 Graphically represent the data structure $B = (K, R)$ with nine nodes k_i $(1 \leqslant i \leqslant 9)$ and three relations $R = \{N, T, Z\}$, where $\omega k_i = i$.

$$N = \{(k_1, k_3), (k_1, k_9), (k_2, k_3), (k_3, k_1), (k_3, k_2), (k_4, k_5), (k_6, k_8)\}$$

$$T = \{(k_4, k_5), (k_3, k_4), (k_7, k_8), (k_1, k_3), (k_8, k_9), (k_5, k_7)\}$$

$$Z = \{(k_i, k_j) \mid i + j = 7\}.$$

1.2.2 Determine all start nodes and end nodes relative to relations N, T, and Z in Exercise 1.2.1.

Section 1.3 Methods of Storing Data Structures

When considering the actual storage and manipulation of data structures using computers, it is useful to begin by constructing a theoretical model for the storage of the data structure. Each memory location in such a model may be visualized as consisting of two parts, the *data field* and the *pointer field.* The data field serves to store the value of a node k. A pointer field can be used in the realization of a relation r, that is, to indicate the memory addresses of all those locations containing successors of k (relative to r). An *indicator*, which can be either a 0 or a 1, may be associated with every pointer field to give additional information about the data structure.

Definition 1.3.1 (Memory) A *memory* consists of a finite number q of *memory locations*, numbered from 1 to q. The location of a memory cell z is called the *address* and is denoted symbolically with αz. Conversely, if a memory location with address t is denoted by σt, then $\alpha\sigma t = t$ and $\sigma\alpha z = z$. A memory location z consists of a *data field*, whose value is denoted δz, and $m \geqslant 0$ *pointer fields.* The ith pointer field of a location z defines a *value*, denoted $\rho_i z$, and an indicator, denoted $\tau_i z$. Frequently the indicator is 0 and need not be explicitly expressed.

The value[2] of a pointer field $\rho_i z$ is a t-tuple, $t \geqslant 1$, of whole numbers. The jth component of $\rho_i z$ is denoted by $\rho_{ij} z$. The value of a data field δz is an n-tuple, $n \geqslant 0$, of character strings, where the jth component is denoted by $\delta_j z$. Unless otherwise specifically stated, the value of a pointer field $\rho_i z$ consists of only one component, and $\rho_{i1} = \rho_i$. If only one pointer field ρ_1 is being considered, then we simply write ρ.

Definition 1.3.2 (Storing a Data Structure) A data structure $B = (K, R)$ is *stored* by uniquely associating every $k \in K$ with a memory location z, where $\delta z = \omega k$. The memory location z associated with a node k is denoted by ξk, and $\xi k = z$.

Definition 1.3.3 (Notational Conventions) Two memory locations z and z′ are *consecutive* if $\alpha z' = 1 + \alpha z$. A memory location z is to the left of z′ and z′ is to the

[2]Where no misunderstanding is likely, no distinction will be made between a pointer field and its value or between a pointer component and its value.

right of z if $\alpha z < \alpha z'$. A set M of consecutive memory locations is called a *storage domain*. The *first* location z in a storage domain M, $\alpha z = \min \{\alpha z'' \mid z'' \in M\}$, is called the *left end* of the storage domain; the *last* location z' of the storage domain, $\alpha z' = \max \{z'' \mid z'' \in M\}$, is the *right end* of the storage domain. A memory location z is said to *point* to a memory location z' (with respect to the ith pointer field or the data field) if the address $\alpha z'$ of z' appears in the ith pointer field or, respectively, the data field of z. If B = (K, R) is a stored data structure and r is a relation, then the nodes K are said to be stored *in* the memory locations $\xi K = \{\xi k \mid k \in K\}$, and the terms *node* and *memory location* may be used interchangeably as long as no misunderstanding is possible, For example, the *address* of a node k, symbolically αk, is the address of the memory location corresponding to the node k; the value of the ith pointer field of a node k is written symbolically as either $\rho_i k$ or $\rho_i \xi k$. Start locations, end locations, and inner locations (relative to a relation r) correspond to start nodes, end nodes, and inner nodes, respectively.

Definition 1.3.2 is only concerned with storing the values of nodes. We must also consider the realization of the relations; this is done in one of two ways: *sequentially* or *linked.*

Definition 1.3.4 (Sequential Realization of a Relation) Let B = (K, R) be a stored data structure. A relation r is said to be realized *sequentially* (with step size s) if, for every pair of nodes $(k, k') \in r$, $\alpha k' = \alpha k + s$ holds. In sequential realization with step size s, consecutive nodes are stored in locations whose addresses differ from one another by s. The step size is usually 1 and, in that case, need not be expressed explicitly.

Definition 1.3.5 (Linked Realization of a Relation) Let B = (K, R) be a stored data structure. A relation r is realized as a *linked* structure (relative to the ith pointer field) if, for every pair of nodes $(k, k') \in r$, $\alpha k' \in \rho_i k$ holds. If $\alpha k' = \rho_i k$ for every pair of nodes $(k, k') \in r$, then the linking is called *linear.* In the linked realization of a relation, the ith pointer field of a node k gives the addresses of all successors of k.

If a relation r is sequentially realized, then we say that the corresponding nodes are sequentially stored. If a relation r is realized as a linked structure, then we say that the corresponding nodes are linked in the memory.

Definition 1.3.6 (Graphical Representation of Memory Locations) Let Z be a set of memory locations. A graphical representation of Z may be obtained as follows: For every $z \in Z$ with m pointer fields, a rectangle is drawn that is divided from left to right into m + 2 segments. The first segment contains αz; the second contains δz; and the remaining m segments contain the values of the pointer fields $\rho_1 z$, $\rho_2 z, \ldots, \rho_m z$. If an indicator $\tau_i z$ is not equal to zero, then a star is attached to the corresponding pointer field. If, in a rectangular segment G of a memory location z, the address $\alpha z'$ of another memory location appears, then G can be connected to the first rectangular segment of memory location z' with a directed line.

For simplification, some of the entries in the rectangular segments corresponding

to αz, δz, or $\rho_1 z, \ldots, \rho_m z$ will occasionally be omitted. In some cases, it will be appropriate to supply names to the rectangles or rectangular segments corresponding to the memory locations. These names will be written outside the rectangles or rectangular segments.

Example 1.3.1 A possible method of storing the data structure $B = (K, R)$ with five nodes $K = \{k_1, k_2, k_3, k_4, k_5\}$ and the relation $r = \{(k_1, k_2), (k_2, k_3), (k_3, k_4), (k_4, k_5)\}$, $\omega k_1 = \text{THAT}$, $\omega k_2 = \text{THE}$, $\omega k_3 = \text{THIS}$, $\omega k_4 = \text{US}$, $\omega k_5 = \text{WE}$, is illustrated in Fig. 1.3.1. The relation r is realized as a linked structure since $\rho k_1 = \alpha k_2$, $\rho k_2 = \alpha k_3$, $\rho k_3 = \alpha k_4$, $\rho k_4 = \alpha k_5$. Note that ρk_5 is not used and is set to 0.

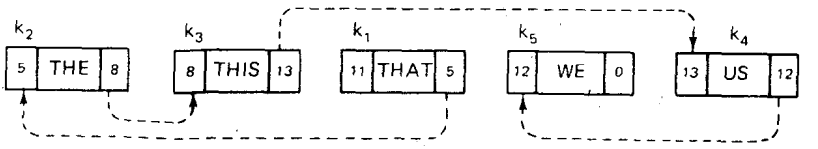

Figure 1.3.1

The words $\omega k_1, \omega k_2, \omega k_3, \omega k_4, \omega k_5$ are alphabetically ordered, in the sense that each memory location z points to that memory location z′ which contains the word which is next in alphabetical order.

Example 1.3.2 Three possible methods of storing the data structure in Fig. 1.2.1 are presented in Fig. 1.3.2, 1.3.3, and 1.3.4. In the method shown in Fig. 1.3.2, neither of the relations T and N is realized. In Fig. 1.3.3, both relations T and N are shown

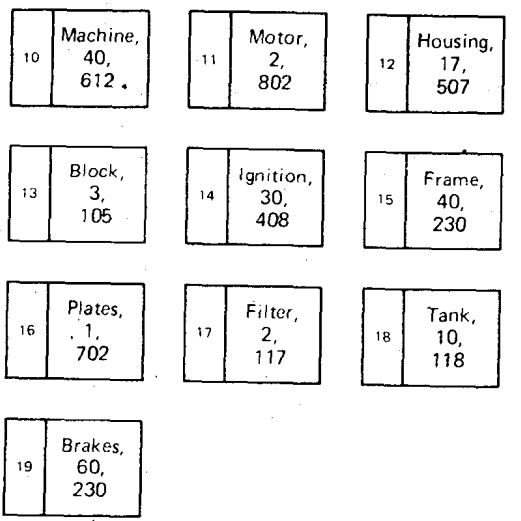

Figure 1.3.2