

IBM**Data Processing Techniques**

Techniques for Processing Pointer Lists and Lists of Lists in PL/I

This manual illustrates usage of PL/I list-processing facilities for processing pointer lists and lists of lists. Pointer lists consist of based variable structures that contain pointers which address data plus pointers that link the structures. Lists of lists contain pointers that address other lists.

The information in this manual assumes knowledge of *Introduction to the List Processing Facilities of PL/I* (GF20-0015) and *Techniques for Processing Data Lists in PL/I* (GF20-0018). The audience for this manual is assumed to be the experienced programmer.

Illustrative programs were processed by the PL/I (F) Compiler (Version 5) under control of the IBM System/360 Operating System (Release 18.6).

First Edition (August 1971)

Copies of this and other IBM publications can be obtained through IBM branch offices.

A form has been provided at the back of this publication for readers' comments. If this form has been removed, address comments to: IBM Corporation, Technical Publications Department, 1133 Westchester Avenue, White Plains, New York 10604.

©Copyright International Business Machines Corporation 1971

Preface

This manual is the third in a series about processing lists with PL/I. It assumes knowledge of the two preceding manuals, *Introduction to the List Processing Facilities of PL/I* (GF20-0015) and *Techniques for Processing Data Lists in PL/I* (GF20-0018).

This manual builds upon the preceding manuals to extend the concept of processing simple and complex data lists to include techniques for processing pointer lists and lists of lists. Function and subroutine procedures used to manipulate the lists are illustrated. Use of suitable inline coding may be preferred for applications.

The illustrative programs compiled and executed. Rea-

sonable care has been exercised to minimize error. Clarity of presentation has been emphasized rather than efficient programming or computer utilization techniques.

The advanced nature of this manual requires the reader to be an experienced programmer who has studied the companion texts mentioned above and who is skilled in the use of subroutines and functions. References to particular implementations of PL/I are held to a minimum, but information on the F-level facilities for processing lists appears in *IBM System/360: PL/I Reference Manual* (GC28-8201) and *IBM System/360 Operating System: PL/I (F) Programmer's Guide* (GC28-6594).

Note: Version 5 of the PL/I (F) Compiler under control of the IBM System/360 Operating System (Release 18.6) produced the program printouts shown in this manual.

Contents

	<i>Page</i>		<i>Page</i>
Introduction	1	PROCESSING LISTS OF LISTS	22
DATA LISTS	1	Creating a List of Available Storage	
POINTER LISTS	1	Components	22
LISTS OF LISTS	1	Manipulating Component Elements in a	
		List of Lists	24
		<i>Obtaining the Address of a List Component</i>	24
		<i>Obtaining the Values of Elements in</i>	
		List Components	27
		<i>Assigning Values to Elements of</i>	
		List Components	31
		<i>Comparing Data Values of List</i>	
		Components	33
Chapter 1. Pointer Lists	3	Manipulating Top Level of a List of Lists	34
ORGANIZING POINTER LISTS	3	Counting Number of Values at Top Level	
Separating Storage for Data Items		of a List of Lists	34
and List Components	3	<i>Inserting Values into Top Level of a</i>	
Sharing Data Items among Pointer Lists	4	List of Lists	36
Storing Mixed Data Types in Pointer Lists	4	<i>Obtaining Values and their Type Codes from</i>	
Freeing Data Storage	5	Top Level of a List of Lists	39
PROCESSING POINTER LISTS	5	<i>Combining Lists of Lists at Top Level</i>	42
USING POINTER LISTS	10	<i>Copying Top Level of a List of Lists in</i>	
Multiple Sorting of Records	10	Reverse Order	46
Multiple Searching of Records	12	Manipulating All Levels of a List of Lists	48
REVIEW OF POINTER LISTS	14	<i>Obtaining First and Last Data Values in a</i>	
SUMMARY	14	List of Lists	48
		<i>Counting Data Values in a List of Lists</i>	52
		<i>Deleting List Components from a</i>	
		List of Lists	53
		<i>Copying Lists of Lists</i>	56
		<i>Testing Lists of Lists</i>	59
		<i>Replacing Data Values in a List of Lists</i>	63
Chapter 2. Lists of Lists	15	USING LISTS OF LISTS	65
ORGANIZING LISTS OF LISTS	15	Sorting with a Binary Tree	65
Component Elements for Lists of Lists	15	Indexing Catalog Cards	70
Permissible Arrangements of List Components	15		
Null Lists of Lists	17	REVIEW OF LISTS OF LISTS	75
Sharing Data Items among Lists of Lists	18		
Sharing List Components among Lists of Lists	20	SUMMARY	75
Parenthetic Representation of Lists of Lists	20		
Circular Lists of Lists	21	Index	76

Introduction

This introduction defines the organization of data lists, pointer lists, and lists of lists. Diagrammatic representation of the three types of list organization is shown. This manual illustrates techniques for processing pointer lists and lists of lists.

DATA LISTS

A *data list* is made up of allocations of based variable structures containing data plus pointer elements that link the structures:

```

DECLARE
  1 DATA_LIST_COMPONENT BASED(P),
  2 DATA CHARACTER (80),
  2 LINK POINTER,
  (P, SAVE, HEAD_POINTER) POINTER;
ALLOCATE DATA_LIST_COMPONENT SET(P);
HEAD_POINTER, SAVE = P;
ALLOCATE DATA_LIST_COMPONENT SET(P);
SAVE->LINK = P;
SAVE = P;
ALLOCATE DATA_LIST_COMPONENT SET(P);
SAVE->LINK = P;
P->LINK = NULL;
  
```

The data list organization resulting from such code is represented in Figure I.1.

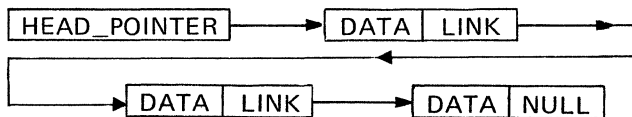


Figure I.1. Data list organization

A data list has two significant limitations: (1) all data items in the list generally must have the same attributes, and (2) the same data item cannot be shared by two or more lists at the same time; a distinct copy of the item must appear in each list, thereby reducing conservation of storage.

POINTER LISTS

The cited limitations of data lists can be avoided by replacing the data items in list components with pointer variables

that specify the locations of data items outside the list. Such a pointer list component can be specified as follows:

```

DECLARE
  1 POINTER_LIST_COMPONENT BASED(P),
  2 DATA_PTR POINTER,
  2 PTR_LINK POINTER,
  (P_HEAD, P) POINTER,
  DATA CHARACTER(80);
  
```

Linked allocations of the type of list component associated with such a declaration are represented in Figure I.2. Such lists are called *pointer lists* because they consist of linked pointers. They retain the advantages of list organization while allowing the same data item to be shared (pointed to) by different lists and permitting the data items associated with a list to have different attributes.

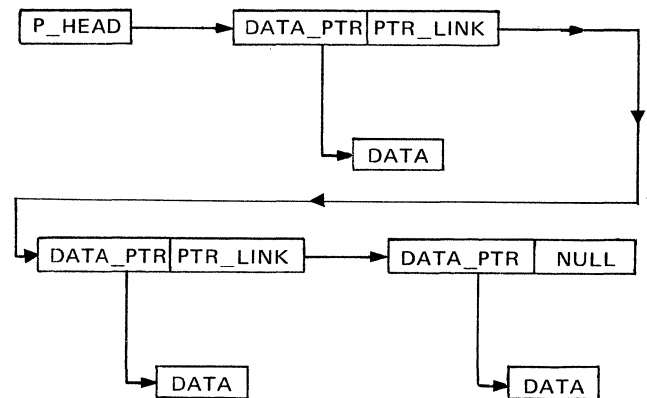


Figure I.2. Pointer list

It is possible to allow a data list to contain data items with different attributes. However, such a list must be processed on an individual basis. Pointer lists, on the other hand, permit general rather than specific processing techniques to be developed for all lists and still allow the data items associated with a list to possess a variety of attributes.

Chapter 1 of this manual discusses pointer lists.

LISTS OF LISTS

The flexibility of a pointer list can be extended to organize lists in higher-level lists called *lists of lists*. Each component in a list of lists can contain three elements:

1. A pointer variable that specifies the location of the next component in the list

- 2. Another pointer variable that specifies the location of the data item or the sublist associated with the component
- 3. A type code that indicates whether a data item (code 'D') or a sublist (code 'L') is associated with the component.

The elements can be specified as follows:

```

DECLARE
  1 LIST_OF_LISTS_COMPONENT BASED(P),
  2 CODE CHARACTER(1),
  2 SUB_PTR POINTER,
  2 LIST_LINK POINTER,

```

(L_HEAD, P) POINTER,
 DATA CHARACTER(80);

Linked allocations of this type of list component are represented in Figure I.3.

Each sublist can contain other sublists to an arbitrary depth, and the number of components permitted in each sublist is also arbitrary. This type of organization retains the advantages of pointer lists and also frees the programmer from having to know the exact number of lists that will be required during a particular run of a program. New lists can be accommodated by treating them as sublists within a master list.

Chapter 2 of this manual discusses lists of lists.

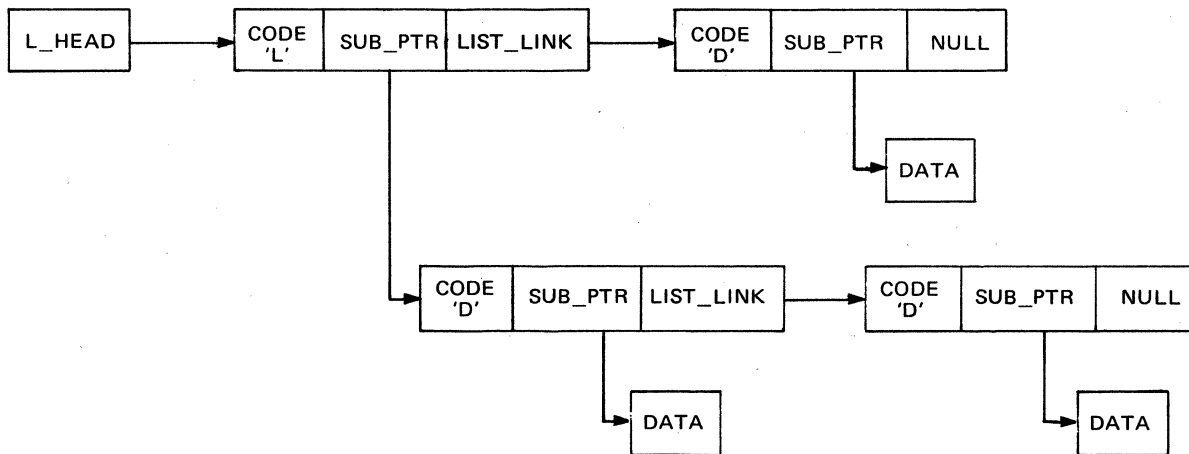


Figure I.3. List of lists

Chapter 1. Pointer Lists

An essential characteristic of a data list is that its data items appear within the body of the list. As a result, a data item must be duplicated if it is to be a member of two different data lists. Duplication of a small data item, such as a single character, does not require much storage. However, duplication of a large data item, such as a long string, or an array or structure with many elements, may lead to excessive use of storage.

A way of avoiding duplicate storage is to store the address of a data item rather than the data item itself in a list. Then storage need be allocated for only one copy of the item.

The type of list produced by this arrangement is called a *pointer list* to distinguish it from a data list. This chapter shows how pointer lists may be organized and how they permit more efficient use of computer storage and program execution time.

ORGANIZING POINTER LISTS

Figure 1.1 illustrates the organization of a pointer list. Each list component contains two pointer elements: DATA and LINK. The DATA pointer contains the address of the data item associated with the component. LINK points to the next list component.

Separating Storage for Data Items and List Components

The data items associated with a pointer list can be of any storage class and data type and can be located anywhere within a program. They can even be intermixed with their associated list components in the same storage area. A less complicated arrangement would involve separate areas for list components and data items. Figure 1.2 shows how a

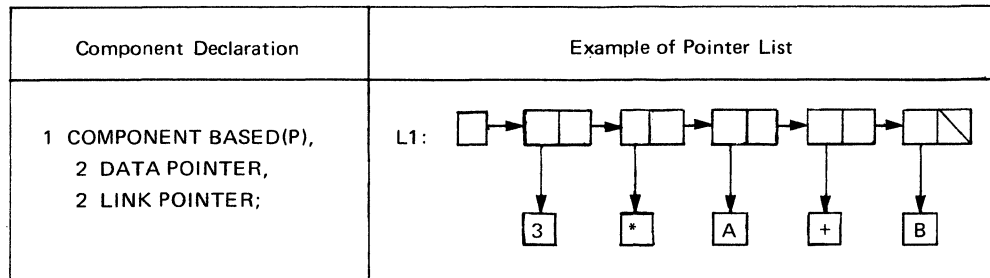


Figure 1.1. Example of a pointer list

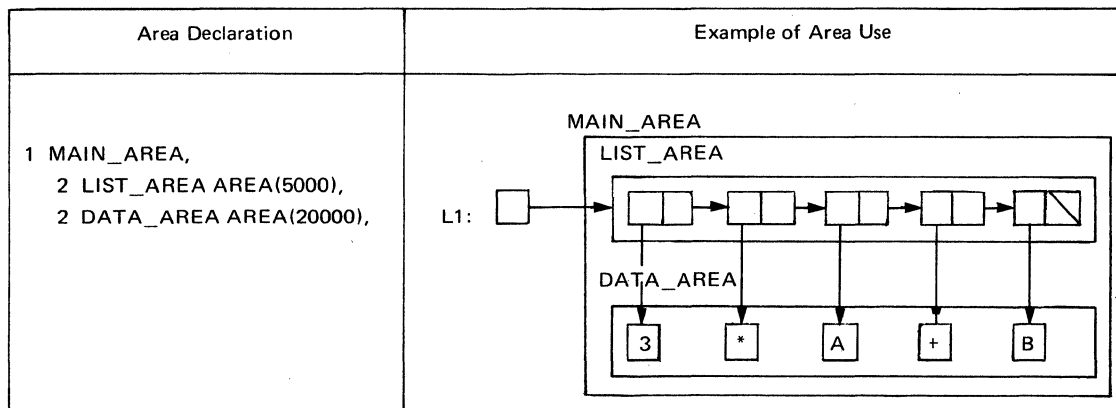


Figure 1.2. Subdividing an area for list storage and data storage

structure organization can be used to divide an area into separate list storage and separate data storage.

Sharing Data Items Among Pointer Lists

The illustration in Figure 1.3 shows how two data lists (L1 and L2) may share data items. Both lists contain the same first two data items, but storage is required for only one copy of each item.

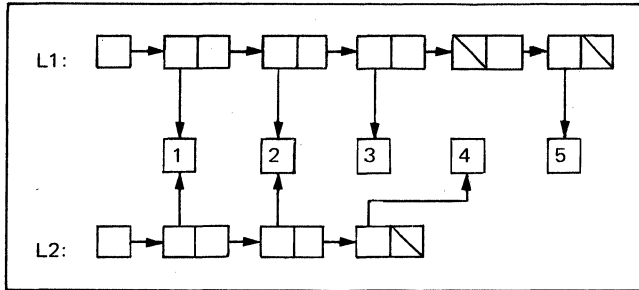


Figure 1.3. Two pointer lists with data items in common

The same data item may appear on an arbitrary number of pointer lists and may also appear an arbitrary number of times on the same list.

Observe that the fourth component of L1 contains a null data pointer, which allows a data item to be removed from a pointer list without requiring a corresponding deletion of the component. This use of a null data pointer avoids the need to link the list component to the list of available storage components when it is known that the list will use the component again.

Also note that the size of L1 is five, even though its fourth position contains a null data pointer. As a result, a null data item is considered to be a possible member of a pointer list.

Because the illustrations for pointer lists can become complicated, a more compact representation is often desirable. Figure 1.4 contains an abbreviated representation of pointer lists.

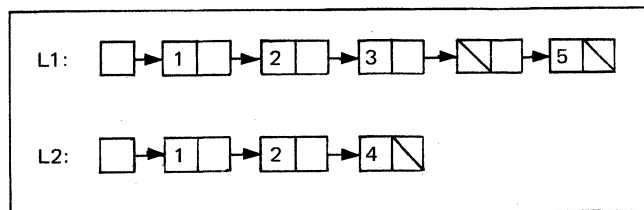


Figure 1.4. Abbreviated representation of pointer lists

Storing Mixed Data Types in Pointer Lists

The techniques used for organizing data lists in *Techniques for Processing Data Lists in PL/I* (GF20-0018) do not permit data lists to contain mixed data types. Such flexibility would require continual allocating and freeing of component storage on an individual basis and would eliminate the efficiency obtained from a list of available storage components.

With pointer lists, however, mixed data types are possible without a loss in efficient storage handling. Figure 1.5 shows a pointer list that contains four data items. The first element in each item represents a type code that distinguishes the item. The first item is a four-position array; the second, a single character; the third, a three-element structure; and the fourth, a single character. A type code would not be necessary if the items always appeared in a predetermined pattern.

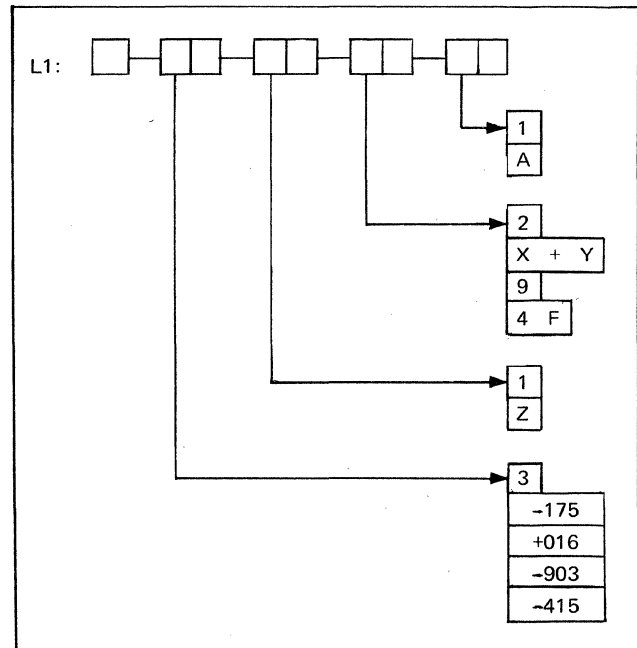


Figure 1.5. A pointer list with data items of mixed type; the first element of each item serves as a type code

Since data items do not appear within the body of a pointer list, the components of the list can have the same structure. It is possible, therefore, to create a list of available storage components for pointer lists that contain data items of mixed type.

Deletion of a data item from a pointer list can return the associated list component to the list of available storage components without destroying the data item. The data item can still be a member of another list, as illustrated in Figure 1.6.

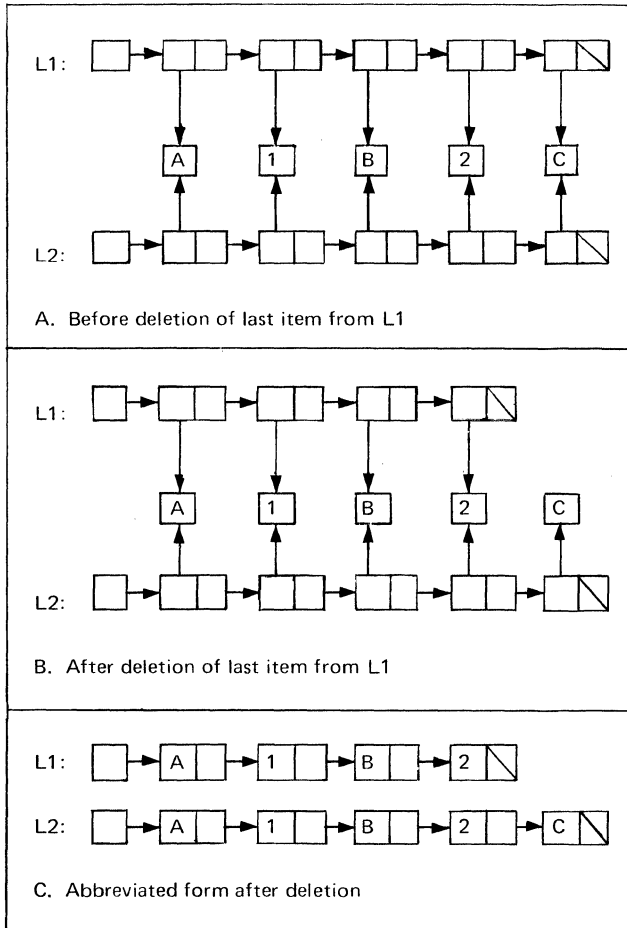


Figure 1.6. Deletion of an item from a pointer list

Freeing Data Storage

A count can be attached to each data item to specify the number of lists that contain the item. As the item is inserted into or deleted from a list, the count can be adjusted appropriately. A zero count would indicate that the item belonged to no list and that its storage could be freed.

PROCESSING POINTER LISTS

The techniques for processing pointer lists resemble those for processing data lists, except that the addresses of data items and not the data items themselves are manipulated within pointer lists. Insertion, deletion, and retrieval of a data item associated with a pointer list always involve the address of the item.

This section presents elementary subroutines and functions (Figures 1.7 through 1.16) for processing simple pointer lists that possess the linear organization developed

earlier in this chapter. Elementary procedures are developed first and used in turn to create higher-level procedures.

Because of the similarity between the techniques of this chapter and those in *Techniques for Processing Data Lists in PL/I*, fewer procedures are developed here. The range of development is restricted to those procedures needed for the examples in the next section, "Using Pointer Lists". More extensive methods for processing pointer lists, including recursive techniques, appear in Chapter 2, which discusses lists of lists.

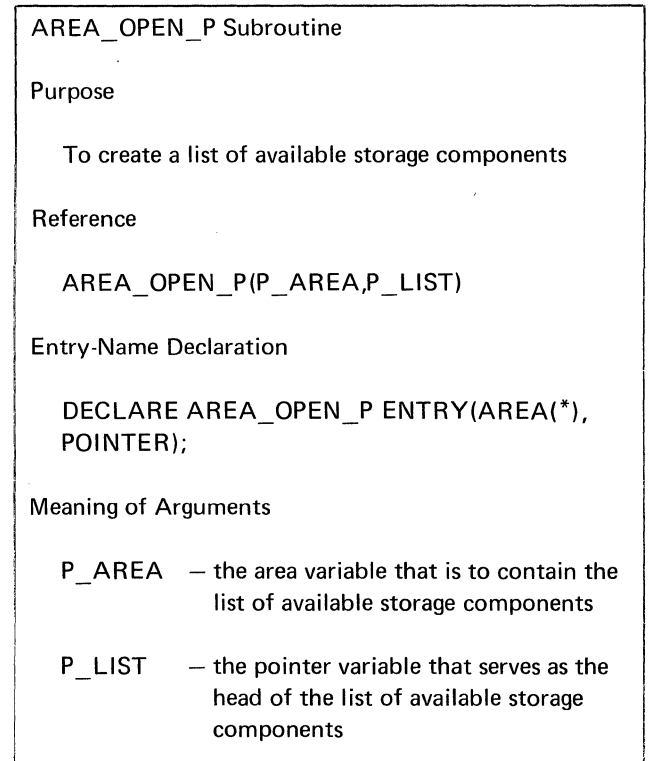


Figure 1.7A. Description of the AREA_OPEN_P subroutine for creating a list of available storage compartments

```
AREA_OPEN_P:
  PROCEDURE (P_AREA, P_LIST);
  DECLARE
    P_AREA AREA(*),
    (P_LIST, T) POINTER,
    1 P_COMP BASED(P),
    2 DATA POINTER,
    2 LINK POINTER;
  ON AREA BEGIN;
  IF
    P->=NULL
  THEN
    P->LINK = NULL;
  GO TO
    END_AREA_OPEN_P;
END;
P = NULL;
ALLOCATE P_COMP IN(P_AREA)
SET(P);
```

```

P_LIST = P;
L:
  T = P;
  ALLOCATE P_COMP IN(P_AREA)
  SET(P);
  T->LINK = P;
  GO TO
  L;
END_AREA_OPEN_P:
END
  AREA_OPEN_P;

```

Figure 1.7B. The AREA_OPEN_P subroutine

```

DO
  I = 1 BY 1;
  IF
    (ADDRESS->LINK = NULL) & (I=N)
  THEN
    RETURN (NULL);
  IF
    I = N
  THEN
    RETURN(ADDRESS);
    ADDRESS = ADDRESS->LINK;
END;
END
  ADDRESS_N_P;

```

Figure 1.8B. The ADDRESS_N_P function

ADDRESS_N_P Function

Purpose

To obtain the address of the nth component in a pointer list

Reference

ADDRESS_N_P(P_LIST,N)

Entry-Name Declaration

```

DECLARE ADDRESS_N_P ENTRY (POINTER,
                          FIXED DECIMAL(5))
  RETURNS(POINTER);

```

Meaning of Arguments

P_LIST — the pointer variable that is the head of the list to be examined

N — a fixed-point decimal integer value that specifies the component whose address is to be obtained; N has a maximum size of five digits

Figure 1.8A. Description of the ADDRESS_N_P function for obtaining the address of the nth component in a pointer list

```

ADDRESS_N_P:
  PROCEDURE (P_LIST, N)
  RETURNS (POINTER);
DECLARE
  P_LIST POINTER,
  (N,I) FIXED DECIMAL(5),
  1 P_COMP BASED(ADDRESS),
  2 DATA POINTER,
  2 LINK POINTER;
  IF
    (P_LIST = NULL) | (N < 1)
  THEN
    RETURN (NULL);
  ADDRESS = P_LIST;

```

GET_LINK_P Function

Purpose

To obtain the address of the next component in a pointer list

Reference

GET_LINK_P(ADDRESS)

Entry-Name Declaration

```

DECLARE GET_LINK_P ENTRY(POINTER)
  RETURNS(POINTER);

```

Meaning of Argument

ADDRESS — a pointer value that specifies the address of a list component

Figure 1.9A. Description of the GET_LINK_P function for obtaining the address of the next component in a pointer list

```

GET_LINK_P:
  PROCEDURE (ADDRESS)
  RETURNS (POINTER);
DECLARE
  1 P_COMP BASED(ADDRESS),
  2 DATA POINTER,
  2 LINK POINTER;
  IF
    ADDRESS = NULL
  THEN
    RETURN (NULL);
    RETURN(ADDRESS->LINK);
END
  GET_LINK_P;

```

Figure 1.9B. The GET_LINK_P function

GET_DATA_P Function

Purpose

To obtain the value of the data pointer in a component of a pointer list

Reference

GET_DATA_P(ADDRESS)

Entry-Name Declaration

```
DECLARE GET_DATA_P ENTRY(POINTER)
        RETURNS(POINTER);
```

Meaning of Argument

ADDRESS — a pointer value that specifies the address of a pointer list component

Figure 1.10A. Description of the GET_DATA_P function for obtaining the value of the data pointer in a component of a pointer list

```
GET_DATA_P:
  PROCEDURE (ADDRESS)
  RETURNS (POINTER);
DECLARE
  1 P_COMP BASED(ADDRESS),
  2 DATA POINTER,
  2 LINK POINTER;
  IF
  ADDRESS = NULL
  THEN
  RETURN (NULL);
  RETURN(ADDRESS->DATA);
END
GET_DATA_P;
```

Figure 1.10B. The GET_DATA_P function

SET_LINK_P Subroutine

Purpose

To assign a value to the link pointer of a component in a pointer list

Reference

SET_LINK_P(ADDRESS,L)

Entry-Name Declaration

```
DECLARE SET_LINK_P ENTRY(POINTER,
POINTER);
```

Meaning of Arguments

ADDRESS — a pointer value that specifies the address of a pointer list component

L — the value to be assigned to the link element of the list component

Figure 1.11A. Description of the SET_LINK_P subroutine for assigning a value to the link pointer of a component in a pointer list

```
SET_LINK_P:
  PROCEDURE(ADDRESS,L);
DECLARE
  L POINTER,
  1 P_COMP BASED(ADDRESS),
  2 DATA POINTER,
  2 LINK POINTER;
  IF
  ADDRESS = NULL
  THEN
  RETURN;
  ADDRESS->LINK = L;
END
SET_LINK_P;
```

Figure 1.11B. The SET_LINK_P subroutine

SET_DATA_P Subroutine

Purpose

To assign a value to the data pointer of a component in a pointer list

Reference

SET_DATA_P(ADDRESS, D)

Entry-Name Declaration

```
DECLARE SET_DATA_P ENTRY(POINTER,  
POINTER);
```

Meaning of Arguments

ADDRESS — a pointer value that specifies the address of a list component

D — the pointer value to be assigned to the data pointer of the list component

Figure 1.12A. Description of the SET_DATA_P subroutine for assigning a value to the data pointer of a component in a pointer list

```
SET_DATA_P:  
  PROCEDURE(ADDRESS,D);  
  DECLARE  
    D POINTER,  
    1 P_COMP BASED(ADDRESS),  
    2 DATA POINTER,  
    2 LINK POINTER;  
    IF  
      ADDRESS = NULL  
    THEN  
      RETURN;  
      ADDRESS->DATA = D;  
  END  
  SET_DATA_P;
```

Figure 1.12B. The SET_DATA_P subroutine

SIZE_P Function

Purpose

To obtain the number of data pointers (null and non-null) in a pointer list

Reference

SIZE_P(P_LIST)

Entry-Name Declaration

```
DECLARE SIZE_P ENTRY(POINTER)  
  RETURNS(FIXED  
  DECIMAL(5));
```

Meaning of Argument

P_LIST — the pointer variable that is the head of the list to be examined

Figure 1.13A. Description of the SIZE_P function for obtaining the number of data pointers in a pointer list

```
SIZE_P:  
  PROCEDURE (P_LIST)  
  RETURNS (FIXED DECIMAL(5));  
  DECLARE  
    (P_LIST, ADDRESS) POINTER,  
    N FIXED DECIMAL(5);  
    ADDRESS = P_LIST;  
  DO  
    N = 0 BY 1;  
    IF  
      ADDRESS = NULL  
    THEN  
      RETURN(N);  
      ADDRESS = GET_LINK_P(ADDRESS);  
  END;  
  END  
  SIZE_P;
```

Figure 1.13B. The SIZE_P function

INSERT_ND_P Subroutine

Purpose

To insert a data pointer into the nth position of a pointer list

Reference

INSERT_ND_P(P_LIST,N,D)

Entry-Name Declaration

```
DECLARE INSERT_ND_P ENTRY(POINTER,  
FIXED DECIMAL(5), POINTER);
```

Meaning of Arguments

P_LIST — the pointer variable that is the head of the list to be processed

N — the position in the list where the data pointer is to be inserted

D — the pointer value to be inserted

Figure 1.14A. Description of the INSERT_ND_P subroutine for inserting a data pointer into the nth position of a pointer list

```
INSERT_ND_P:  
PROCEDURE(P_LIST, N, D);  
DECLARE  
  N FIXED DECIMAL(5),  
  (P,Q) POINTER,  
  (P_LIST, D, ADDRESS1, ADDRESS2,  
  AVAIL_P EXTERNAL) POINTER;  
  /* IF LIST OF AVAILABLE STORAGE  
  COMPONENTS IS EMPTY, THEN PRINT  
  MESSAGE AND RETURN. */  
  IF AVAIL_P = NULL THEN DO;  
  PUT  
  LIST('LIST OF AVAILABLE STORAGE IS  
  EMPTY');  
  RETURN; END;  
  /* ASSIGN DATA ITEM TO FIRST  
  COMPONENT IN LIST OF AVAILABLE  
  STORAGE. */  
  CALL SET_DATA_P(AVAIL_P, D);  
  /* IF P_LIST IS NULL OR N<2, INSERT  
  FIRST COMPONENT OF AVAIL_P INTO  
  FIRST POSITION OF LIST AND RETURN*/  
  IF  
  (P_LIST = NULL) | (N < 2)  
  THEN DO;  
  ADDRESS1 = P_LIST; P_LIST = AVAIL_P;  
  AVAIL_P = ADDRESS_N_P(AVAIL_P, 2);  
  CALL SET_LINK_P(P_LIST, ADDRESS1);  
  RETURN; END;  
  IF N > SIZE_P(P_LIST)  
  THEN DO;
```

```
P = P_LIST;  
DO WHILE(P != NULL);  
Q = P; P = Q->LINK;  
END;  
P, Q->LINK = AVAIL_P;  
AVAIL_P = ADDRESS_N_P(AVAIL_P, 2);  
P->LINK = NULL;  
RETURN;  
END;  
/* OTHERWISE OBTAIN THE ADDRESS OF  
THE N-TH COMPONENT OF P_LIST. */  
ADDRESS2 = ADDRESS_N_P(P_LIST, N);  
ADDRESS1 = ADDRESS_N_P(P_LIST, N-1);  
/* INSERT FIRST COMPONENT OF AVAIL_P  
INTO THE N-TH POSITION OF P_LIST. */  
CALL SET_LINK_P(ADDRESS1, AVAIL_P);  
ADDRESS1 = AVAIL_P;  
AVAIL_P = ADDRESS_N_P(AVAIL_P, 2);  
CALL SET_LINK_P(ADDRESS1, ADDRESS2);  
END INSERT_ND_P;
```

Figure 1.14B. The INSERT_ND_P subroutine

GET_ND_P Function

Purpose

To get the value of the data pointer in the nth position of a pointer list

Reference

GET_ND_P(P_LIST,N)

Entry-Name Declaration

```
DECLARE GET_ND_P ENTRY(POINTER, FIXED  
DECIMAL(5))  
RETURNS(POINTER);
```

Meaning of Arguments

P_LIST — the pointer variable that is the head of the list to be processed

N — the position of the data pointer whose value is to be obtained

Figure 1.15A. Description of the GET_ND_P function for getting the value of the data pointer in the nth position of a pointer list

```

GET_ND_P:
  PROCEDURE (P_LIST, N)
  RETURNS (POINTER);
  DECLARE
    P_LIST POINTER,
    N FIXED DECIMAL(5);
  RETURN(GET_DATA_P
    (ADDRESS_N_P(P_LIST, N)));
END
  GET_ND_P;

```

Figure 1.15B. The GET_ND_P function

DELETE_ND_P Subroutine	
Purpose	
To delete the data pointer in the nth position of a pointer list	
Reference	
DELETE_ND_P(P_LIST,N)	
Entry-Name Declaration	
<pre> DECLARE DELETE_ND_P ENTRY(POINTER, FIXED DECIMAL(5)); </pre>	
Meaning of Arguments	
P_LIST	— the pointer variable that is the head of the list to be processed
N	— the position of the data pointer to be deleted

Figure 1.16A. Description of the DELETE_ND_P subroutine for deleting the data pointer in the nth position of a pointer list

```

DELETE_ND_P:
  PROCEDURE (P_LIST, N);
  DECLARE
    N FIXED DECIMAL(5),
    (P_LIST, ADDRESS1, ADDRESS2, ADDRESS3,
    AVAIL_P EXTERNAL) POINTER;
  /* IF P_LIST IS EMPTY OR N IS LESS
  THAN 1, THEN RETURN. */
  IF
    (P_LIST = NULL) | (N < 1)
  THEN
    RETURN;
  /* DELETE FIRST COMPONENT IF N
  EQUALS 1. */
  IF
    N = 1
  THEN

```

```

DO;
  ADDRESS2 = P_LIST;
  P_LIST = ADDRESS_N_P(P_LIST, 2);
  GO TO
  L;
END;
  /* OBTAIN N-TH COMPONENT. */
  ADDRESS2 = ADDRESS_N_P(P_LIST,N);
  IF
    ADDRESS2 = NULL
  THEN
    RETURN;
  ADDRESS1 = ADDRESS_N_P(P_LIST,N-1);
  ADDRESS3 = ADDRESS_N_P(P_LIST,N+1);
  /* DELETE N-TH COMPONENT. */
  CALL SET_LINK_P(ADDRESS1,ADDRESS3);
  /* INSERT DELETED COMPONENT INTO
  LIST OF AVAILABLE STORAGE
  COMPONENTS. */
  L:
  ADDRESS1 = AVAIL_P;
  AVAIL_P = ADDRESS2;
  CALL SET_LINK_P(AVAIL_P,ADDRESS1);
END
  DELETE_ND_P;

```

Figure 1.16B. The DELETE_ND_P subroutine

USING POINTER LISTS

Pointer lists possess the same advantages as data lists in providing efficient control over varying storage requirements. As with data lists, pointer lists need not reserve dormant storage in anticipation of maximum requirements; storage not needed by one list can be used by another.

Pointer lists also provide two additional benefits not obtained from data lists. They permit a data item to be a member of two or more lists at the same time and also eliminate unnecessary data movement. Both benefits are obtained by manipulating the addresses of data items rather than the items themselves.

The following discussions demonstrate these advantages by two examples. The first example shows how multiple sorts may be performed efficiently on the records of a file by manipulating the addresses of the records. The second example illustrates how different arrangements of the same records on separate pointer lists permit efficient searching of the records for different key values.

Multiple Sorting of Records

Figure 1.17 presents the M_SORT program, which shows how a sort can be made more efficient by avoiding unnecessary data movement. The program obtains records from the standard system-input file (SYSIN) and prints the records in two different sorted arrangements on the standard system-output file (SYSPRINT).

```

M_SORT:
  PROCEDURE;
  DECLARE
    (I,J,SIZE1,SIZE2)
    FIXED DECIMAL(5),
    (AVAIL_P EXTERNAL, AUTHOR_LIST,
    TITLE_LIST, P1, P2) POINTER,
    1 MAIN_AREA,
    2 LIST_AREA AREA,
    2 DATA_AREA AREA,
    1 CARD,
    2 FIELD1 CHARACTER(15),
    2 FIELD2 CHARACTER(25),
    2 FIELD3 CHARACTER(10),
    2 FIELD4 CHARACTER(30),
    1 DOCUMENT BASED(P1),
    2 AUTHOR CHARACTER(15),
    2 TITLE CHARACTER(25),
    2 SUBJECT CHARACTER(10),
    2 DESCRIPTORS CHARACTER(30);
    /* WHEN DATA_AREA IS EXHAUSTED OR
    ALL DOCUMENT CARDS HAVE BEEN READ,
    GO TO PRINT_AUTHOR_LIST. */
    ON AREA
  GO TO
  PRINT_AUTHOR_LIST;
  ON ENDFILE (SYSIN)
  GO TO
  PRINT_AUTHOR_LIST;
  /* INITIALIZE. */
  SIZE1,SIZE2 = 0;
  AUTHOR_LIST,TITLE_LIST = NULL;
  /* FORM LIST OF AVAILABLE STORAGE
  COMPONENTS IN LIST_AREA. */
  CALL AREA_OPEN_P(LIST_AREA,AVAIL_P);
  /* GET DOCUMENT CARDS, AND ASSIGN
  THEM TO STORAGE ALLOCATED IN
  DATA_AREA. ALSO FORM A POINTER
  LIST OF DOCUMENT CARDS SORTED ON
  AUTHOR. */
#AUTHOR:
  DO WHILE
    (18);
    /* DO WHILE(18) IS TERMINATED */
    /* BY EOF OR AREA CONDITION */
  GET
    EDIT(CARD)(A(15),A(25),A(10),A(30));
    ALLOCATE DOCUMENT IN(DATA_AREA)
    SET(P1);
    P1->DOCUMENT = CARD;
    /* FIND INSERTION POINT IN
    AUTHOR_LIST. */
  DO
    I = 1 TO SIZE1 BY 1;
    P2 = GET_ND_P(AUTHOR_LIST,I);
    IF P2 = NULL
      THEN GO TO INSERT_AUTHOR;
  IF
    P1->AUTHOR<P2->AUTHOR
  THEN
    GO TO
    INSERT_AUTHOR;
  END;
  /* INSERT ADDRESS OF DOCUMENT IN
  AUTHOR_LIST. */
  INSERT_AUTHOR:
    CALL INSERT_ND_P(AUTHOR_LIST,I,P1);
    SIZE1 = SIZE1+1;
  END_AUTHOR:
  END
  #AUTHOR;
  /* PRINT AUTHOR_LIST. */
  PRINT_AUTHOR_LIST:
    PUT
      PAGE;
    PUT
      LIST('AUTHOR FILE');
    PUT
      SKIP(2);
  DO
    I = 1 TO SIZE1 BY 1;
    P1 = GET_ND_P(AUTHOR_LIST,I);
    PUT
      EDIT(P1->DOCUMENT)(A);
    PUT
      SKIP;
  END;
  /* SORT DOCUMENT CARDS ON TITLE. */
  #TITLE:
  DO
    J = 1 TO SIZE1 BY 1;
    /* GET AND DELETE ADDRESS OF FIRST
    DOCUMENT FROM AUTHOR_LIST. */
    P1 = GET_ND_P(AUTHOR_LIST,1);
    CALL DELETE_ND_P(AUTHOR_LIST,1);
    /* FIND INSERTION POINT IN
    TITLE_LIST. */
  DO
    I = 1 TO SIZE2 BY 1;
    P2 = GET_ND_P(TITLE_LIST,I);
    IF P2 = NULL
      THEN GO TO INSERT_TITLE;
  IF
    P1->TITLE<P2->TITLE
  THEN
    GO TO
    INSERT_TITLE;
  END;
  /* INSERT ADDRESS OF DOCUMENT IN
  TITLE_LIST. */
  INSERT_TITLE:
    CALL INSERT_ND_P(TITLE_LIST, I,P1);
    SIZE2 = SIZE2 + 1;
  END_TITLE:
  END
  #TITLE;
  /* PRINT TITLE_LIST. */
  PRINT_TITLE_LIST:
    PUT
      PAGE;
    PUT
      LIST('TITLE FILE');
    PUT
      SKIP(2);
  DO
    I = 1 TO SIZE1 BY 1;
    P1 = GET_ND_P(TITLE_LIST,I);
    PUT
      EDIT(P1->DOCUMENT)(A);
    PUT
      SKIP;
  END;
  END
  M_SORT;

```

Figure 1.17. The M_SORT procedure

Each record describes a document and contains four fields: AUTHOR, TITLE, SUBJECT, and DESCRIPTORS. The records are printed in sort order: first on AUTHOR, then on TITLE.

As document cards are read, storage is allocated in DATA_AREA, and the address of each card is stored in the pointer list AUTHOR_LIST, which is arranged in ascending sequence on AUTHOR. LIST_AREA contains all list components.

After AUTHOR_LIST is used to print the document cards in sort order, successive data addresses are removed from the list and inserted into the pointer list TITLE_LIST, which is arranged in ascending sequence on TITLE. This list is used in the second printing of the sorted document cards.

Note that during both sorts the document cards remain at their original storage locations within DATA_AREA; only the addresses of the cards are rearranged within both lists. As a result, less data is moved, and a more efficient sort is obtained.

When the number of document cards exceeds the storage capacity of DATA_AREA, only those cards that can be stored in the area are sorted.

Multiple Searching of Records

Since the same data item may be referred to simultaneously by two different pointer lists, it is possible to maintain more than one sort arrangement of a single set of data items. Multiple arrangements of this type avoid data duplication and permit faster searching of items on different keys.

Figure 1.18A contains the SEARCH program, which arranges a set of records on two different keys and searches the records for specified values of the keys. Each record describes a document and contains four fields: AUTHOR, TITLE, SUBJECT, and DESCRIPTORS. The records are read from the standard system-input file (SYSIN) and stored at locations allocated within DATA_AREA. The addresses of the records are stored in two pointer lists: AUTHOR_LIST and TITLE_LIST. AUTHOR_LIST is arranged in ascending sequence on AUTHOR and TITLE_LIST, on TITLE (as shown in Figure 1.18B). LIST_AREA provides all storage for list components.

```
SEARCH:
PROCEDURE ;
DECLARE
1 MAIN_AREA,
2 LIST_AREA AREA,
2 DATA_AREA AREA,
(AVAIL_P EXTERNAL, AUTHOR_LIST,
TITLE_LIST, P1, P2) POINTER,
(SIZE, 1) FIXED DECIMAL(5),
SEARCH_CARD CHARACTER(80),
TITLE_ITEM CHARACTER(25),
AUTHOR_ITEM CHARACTER(15),
1 CARD,
```

```
2 FIELD1 CHARACTER(15),
2 FIELD2 CHARACTER(25),
2 FIELD3 CHARACTER(10),
2 FIELD4 CHARACTER(30),
1 DOCUMENT BASED(P1),
2 AUTHOR CHARACTER(15),
2 TITLE CHARACTER(25),
2 SUBJECT CHARACTER(10),
2 DESCRIPTORS CHARACTER(30);
/* WHEN DATA AREA CANNOT HOLD ALL
DOCUMENT CARDS, OR ALL SEARCH
CARDS HAVE BEEN PROCESSED, TERMINATE
PROGRAM. */
ON AREA
GO TO
END_SEARCH;
ON ENDFILE (SYSIN)
GO TO
END_SEARCH;
/* INITIALIZE. */
SIZE = 0;
AUTHOR_LIST, TITLE_LIST = NULL;
/* FORM LIST OF AVAILABLE STORAGE
COMPONENTS IN LIST_AREA. */
CALL AREA_OPEN_P(LIST_AREA, AVAIL_P);
/* GET DOCUMENT CARDS, AND ASSIGN
THEM TO STORAGE ALLOCATED IN
DATA_AREA. ALSO FORM SORTED AUTHOR
LIST AND TITLE LIST. */
GET_DOCUMENT_CARD:
GET
EDIT(CARD)(A(15), A(25), A(10),
A(30));
IF
SUBSTR(FIELD1, 1, 5) = '*****'
THEN
GO TO
DOCUMENT_SEARCH;
ALLOCATE DOCUMENT IN(DATA_AREA)
SET(P1);
P1->DOCUMENT = CARD;
/* FIND INSERTION POINT IN AUTHOR
LIST. */
DO
I = 1 TO SIZE BY 1;
P2 = GET_ND_P(AUTHOR_LIST, I);
IF P2 = NULL
THEN GO TO INSERT_AUTHOR;
IF
P1->AUTHOR<P2->AUTHOR
THEN
GO TO
INSERT_AUTHOR;
END;
/* INSERT ADDRESS OF DOCUMENT IN
SORTED AUTHOR LIST. */
INSERT_AUTHOR:
CALL INSERT_ND_P(AUTHOR_LIST, I, P1);
/* FIND INSERTION POINT IN TITLE
LIST. */
DO
I = 1 TO SIZE BY 1;
P2 = GET_ND_P(TITLE_LIST, I);
IF P2 = NULL
THEN GO TO INSERT_TITLE;
IF
P1->TITLE<P2->TITLE
THEN
GO TO
INSERT_TITLE;
END;
/* INSERT ADDRESS OF DOCUMENT IN
SORTED TITLE LIST. */
```



```

INSERT_TITLE:
  CALL INSERT_ND_P(TITLE_LIST,I,P1);
  /* INCREASE SIZE BY ONE, AND GET
  NEXT DOCUMENT CARD. */
  SIZE = SIZE + 1;
  GO TO
  GET_DOCUMENT_CARD;
  /* READ SUCCESSIVE SEARCH CARDS,
  AND PRINT CORRESPONDING DOCUMENT
  CARDS. */
DOCUMENT_SEARCH:
  GET
  EDIT(SEARCH_CARD)(A(80));
  PUT
  SKIP(2);
  PUT
  EDIT(SEARCH_CARD)(A(80));
  IF
  SUBSTR(SEARCH_CARD, 1, 1) = 'A'
  THEN
  GO TO
  AUTHOR_SEARCH;
  /* PERFORM TITLE SEARCH. */
TITLE_SEARCH:
  TITLE_ITEM = SUBSTR(SEARCH_CARD,
  2, 25);
  DO
  I = 1 TO SIZE BY 1;
  P1 = GET_ND_P(TITLE_LIST, I);
  IF
  TITLE_ITEM < P1->TITLE
  THEN
  GO TO
  DOCUMENT_SEARCH;
  IF
  TITLE_ITEM = P1->TITLE
  THEN
  PUT
  EDIT(P1->DOCUMENT)
  (COLUMN(1),A(15),A(25),A(10),A(30));
  /* WHEN THIS POINT IS REACHED, GET
  NEXT SEARCH CARD. */
  GO TO
  DOCUMENT_SEARCH;
  /* PERFORM AUTHOR SEARCH. */
AUTHOR_SEARCH:
  AUTHOR_ITEM = SUBSTR(SEARCH_CARD,
  2, 15);
  DO
  I = 1 TO SIZE BY 1;
  P1 = GET_ND_P(AUTHOR_LIST, I);
  IF
  AUTHOR_ITEM < P1->AUTHOR
  THEN
  GO TO
  DOCUMENT_SEARCH;
  IF
  AUTHOR_ITEM = P1->AUTHOR
  THEN
  PUT
  EDIT(P1->DOCUMENT)
  (COLUMN(1),A(15),A(25),A(10),A(30));
  /* WHEN THIS POINT IS REACHED, GET
  NEXT SEARCH CARD. */
  GO TO
  DOCUMENT_SEARCH;
END_SEARCH:
END
SEARCH;

```

Figure 1.18A. The SEARCH procedure

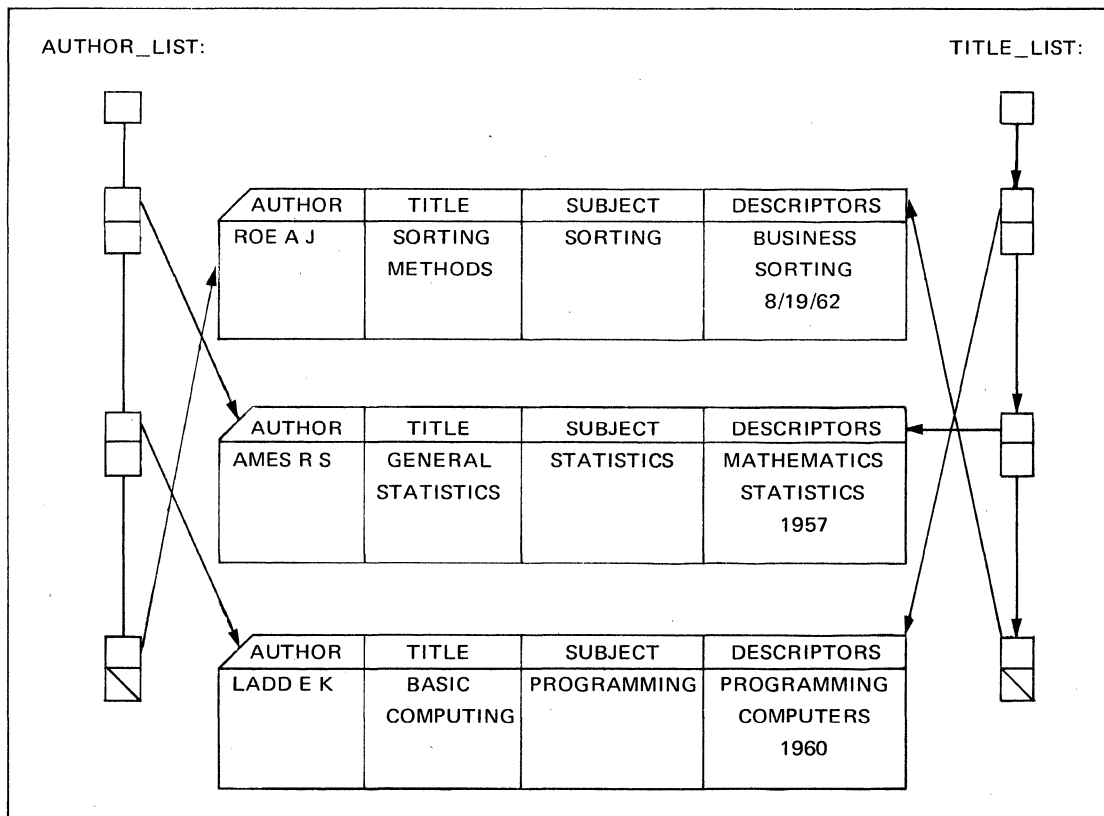


Figure 1.18B. How single copies of document cards are arranged in sort order on two different pointer lists

A trailer card with asterisks in cc 1 through 5 follows the last document card in the input file. If DATA_AREA cannot hold all the document cards, the program is terminated.

The remaining input cards are search cards, which contain two items: a type code and a key value. The type code appears in cc 1 and must be either the letter A or the letter T. Type code A indicates that the key value is the name of an author, which appears left-adjusted in cc 2 through 16 of the search card. The key value associated with type code T is a title, which appears left-adjusted in cc 2 through 26.

As each search card is read, the appropriate pointer list, either AUTHOR_LIST or TITLE_LIST, is searched for the specified AUTHOR value or TITLE value. All document cards that contain the key value are printed along with the search card on the standard system-output file (SYSPRINT).

Associating the document cards with two sorted pointer lists eliminates the need for exhaustive searching through all the document cards. Each search ends when the specified key exceeds the key value in the current document card.

The SEARCH program need not be restricted to two pointer lists; a pointer list can be created for each field in the document cards. In a more elaborate program a pointer list can be created for each key value. For example, a pointer list can be created for all document cards that contain the SUBJECT value "programming". With such a list, no searching is required, since the list contains only those document cards that have "programming" as their SUBJECT value. The term "inverted file" is often used to describe such arrangements.

REVIEW OF POINTER LISTS

This chapter shows how to overcome the main disadvantages of data lists by replacing each data item in a list with a pointer variable that specifies the address of the data item outside the body of the list (see Figure 1.19). The resulting pointer list still retains flexible control over varying storage requirements, but it also eliminates much of the duplication and movement of data produced by data lists and allows mixed data types to be associated with the same list.

SUMMARY

1. Pointer lists resemble data lists, except that the address of a data item rather than the data item itself appears in a pointer list.
2. Pointer lists provide the same advantages as data lists in maintaining efficient control over varying storage requirements. Pointer lists also possess the following additional benefits:
 - a. A data item may be shared by two or more pointer lists, thus avoiding data duplication.
 - b. Transmission of data addresses (rather than data items) to and from pointer lists reduces data movement.
 - c. Different orderings of the same collection of data items may be obtained with separate pointer lists.
 - d. Mixed data types may be associated with the same pointer lists.

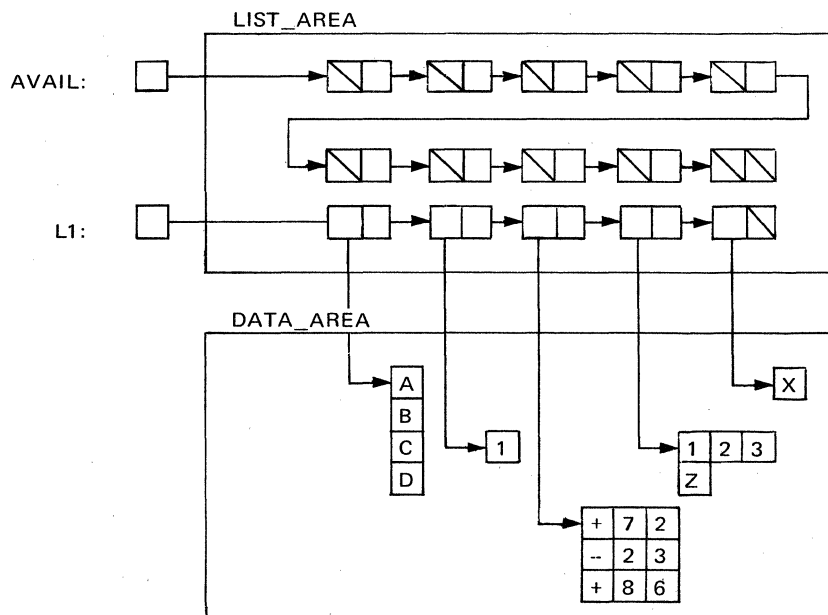


Figure 1.19. Pointer lists

Chapter 2. Lists of Lists

The previous chapter shows how pointer lists may be used to link various combinations of data elements, arrays, and structures. However, pointer lists need not be restricted to these types of items; the lists themselves can also appear as members of a pointer list. The higher-level list formed by this type of linkage is called a *list of lists*. This chapter describes how such lists are constructed and shows how they extend the general flexibility and efficiency of pointer lists.

ORGANIZING LISTS OF LISTS

A pointer list and a list of lists use pointer values to link data items that appear outside the body of the list. Since a list of lists can link other lists as well as data items, some method must be used to determine whether a list component specifies a data item or a sublist.

The following discussions use a type code within each list component to distinguish between a data item and a sublist, and illustrate the effect of this code upon list organization and list-processing techniques.

Component Elements for Lists of Lists

Each component in a list of lists may contain three elements:

1. A type code (TYPE), which is a single-position character string that contains the character 'D' (for data item) or the character 'L' (for list)
2. A value pointer (VALUE), which specifies the address of a data item or a sublist
3. A link pointer (LINK), which specifies the address of the next component in the list

The declaration in A of Figure 2.1 shows how these elements may be combined to form a list component (COMPONENT).

A schematic representation of a component for a list of lists appears in B of Figure 2.1.

The diagram in C of Figure 2.1 shows a list component that specifies a data item (the character *).

In D of Figure 2.1, the component with type code 'L' specifies a sublist. The first component in the sublist specifies a data item (the character *).

Note that the VALUE pointer in a list component with type code 'L' serves as the head pointer of the specified sublist.

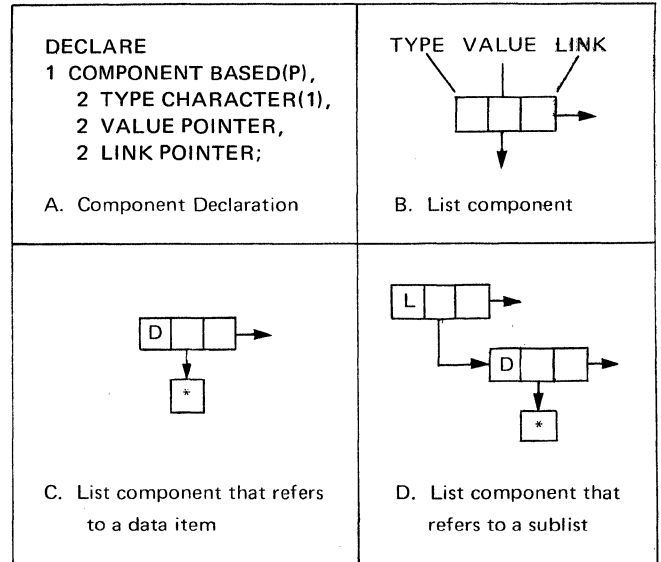


Figure 2.1. Illustrations of list components for lists of lists

Permissible Arrangements of List Components

A list of lists can contain D-components only, L-components only, or any combination of D- and L-components. The number of levels to which sublists may be linked is arbitrary and limited only by available storage.

Figure 2.2 shows a list of lists that contains data items only. Part A displays the full form of the list; storage areas for the data items are shown outside the body of the list. A more compact representation appears in Part B which shows the data items within the body of the list. Since it is

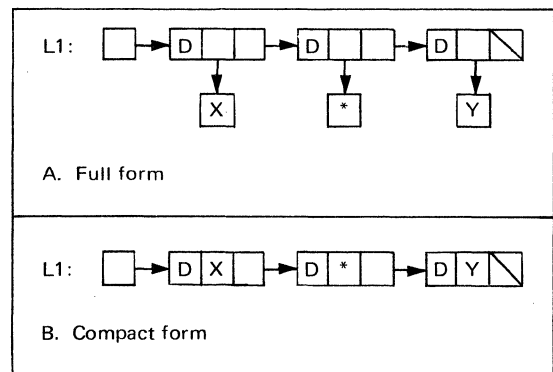


Figure 2.2. A list of lists that contains data items only

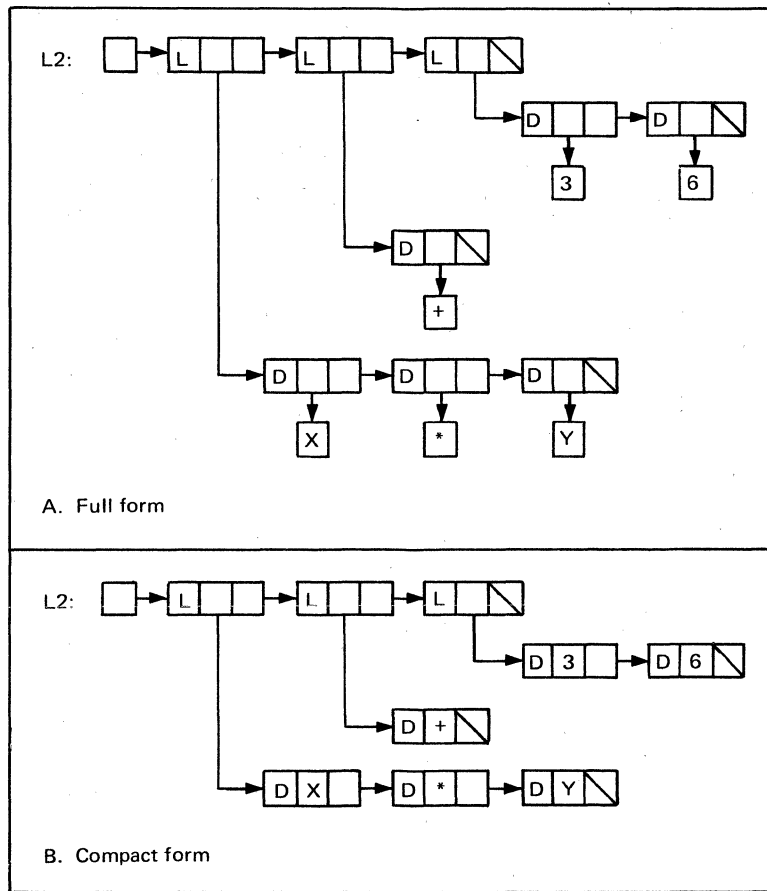


Figure 2.3. A list of lists that contains lists at the top level only

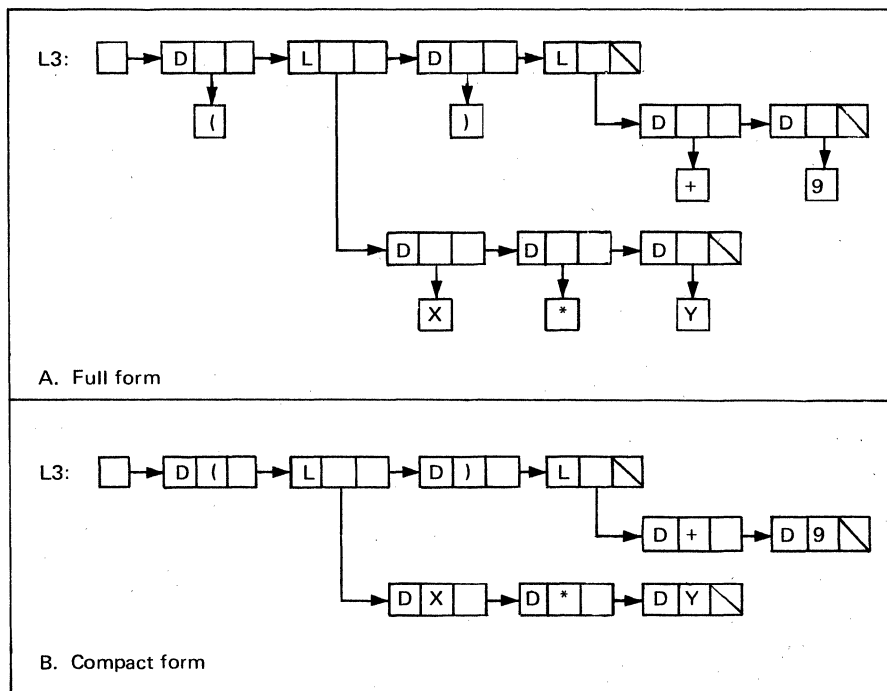


Figure 2.4. A list of lists that contains both data items and lists at the top level

understood that the value element in a list component always contains the address of a data item and not the item itself, the two representations in Figure 2.2 may be considered equivalent.

The use of single-character data items in Figure 2.2 is arbitrary; a list of lists can contain data items of any size and type.

Figure 2.3 shows how three sublists may be linked to form a higher level list. The linking is done so that the resulting lists of lists contains the sublists at the top level.

A combination of data items and lists at the top level of a list of lists appears in Figure 2.4, and Figure 2.5 presents a list of lists that contains data items and lists at multiple levels.

Null Lists of Lists

As with data lists and pointer lists, a null address value for the head pointer of a list of lists creates a null list (A in Figure 2.6). Note the distinction between a list of lists with a null head and a list of lists that contains a null data item (B in Figure 2.6). A list with a null head has zero size, but a list of lists that contains a null data item requires a list component for the item and, therefore, has a size of one.

Observe further the difference between a list of lists that contains a null data item (B in Figure 2.6) and a list of lists that contains a null list (C in Figure 2.6). Both lists have a size of one, but the first contains a D-component, and the second, an L-component.

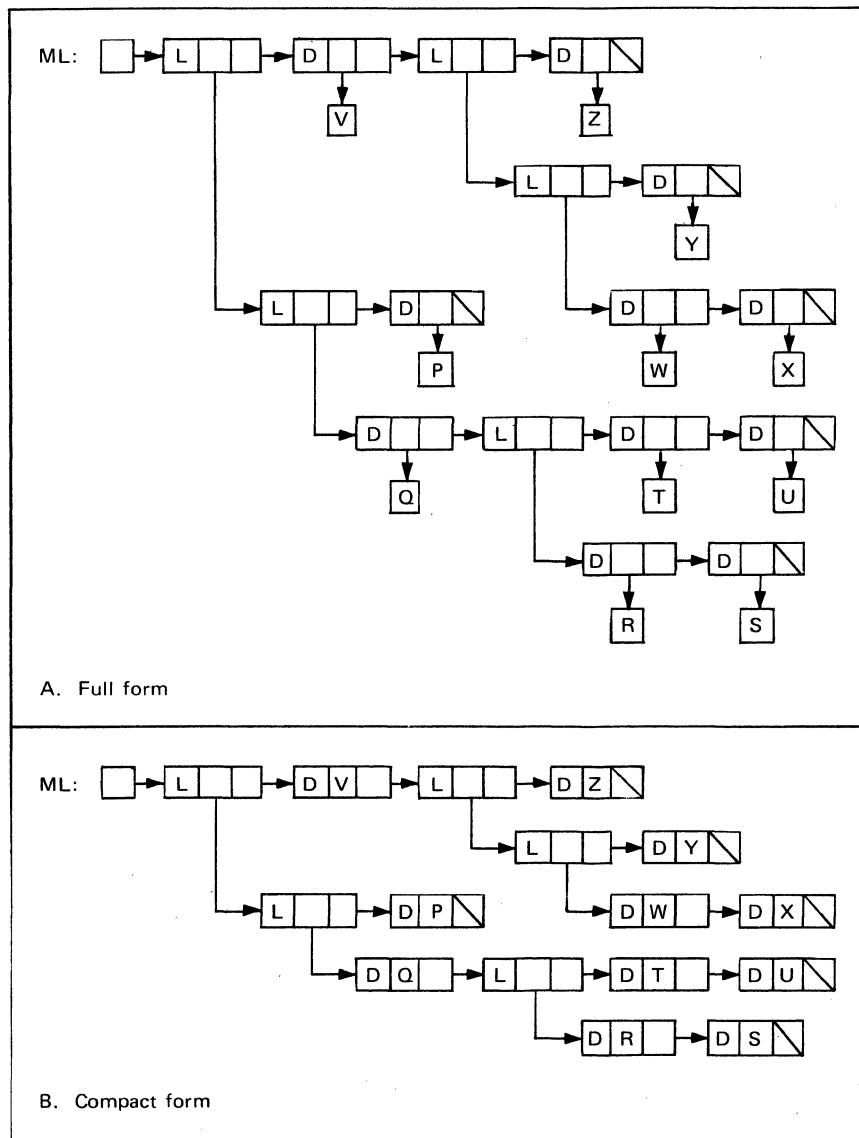


Figure 2.5. A list of lists that contains data items and lists at multiple levels

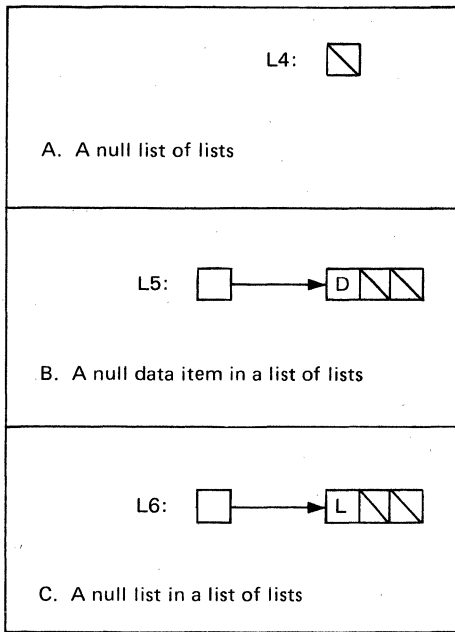


Figure 2.6. Examples of null lists of lists

For the purposes of this book, the three lists of lists in Figure 2.6 are considered to be null lists because they contain null value pointers. As a result, a null list of lists can have a size greater than zero. Such a size represents the number of components in the null list.

Sharing Data Items Among Lists of Lists

The ability of two or more pointer lists to share the same data item is retained by lists of lists. Figure 2.7 shows how two lists of lists can contain the same data items without requiring duplicate storage for the items. Although the compact forms of both lists may seem to indicate duplication of the data items, remember that the value element of a list component contains the address of an item and not the item itself.

Sharing of data items among lists of lists permits direct access to various subsets of items in a collection. The list of lists `L_ARRAY` in Figure 2.8, for example, contains eight sublists—`ELEMENTS`, `ROW1`, `ROW2`, `COL1`, `COL2`,

`DIAG1`, and `DIAG2`—which contain various combinations of the elements in a two-dimensional array. Although each element of the array appears in four sublists, storage for only one copy of each element is required.

A similar application of lists of lists may be used to represent the organization of PL/I structures. As an example, consider the list of lists `L_STRUCTURE` in Figure 2.9. This list contains three sublists, which represent the three minor structures declared at the left of the diagram.

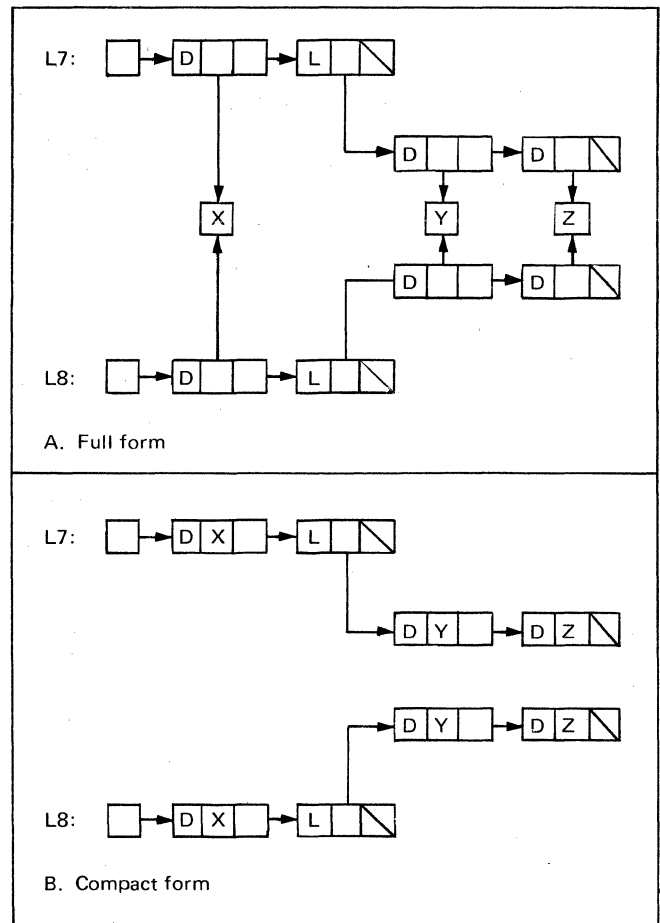


Figure 2.7. Sharing data items among lists of lists

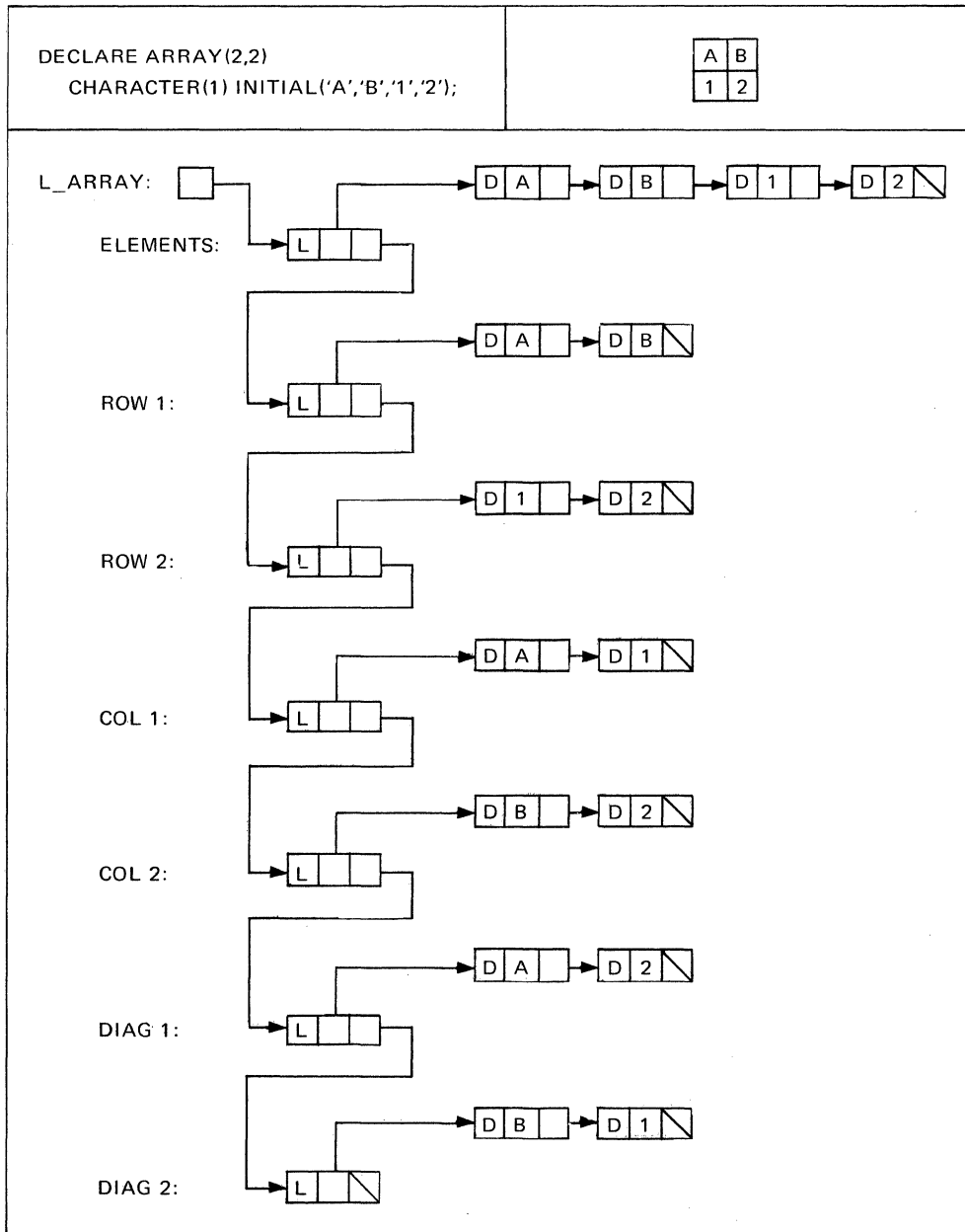


Figure 2.8. A list of lists that contains various sets of elements from the same array

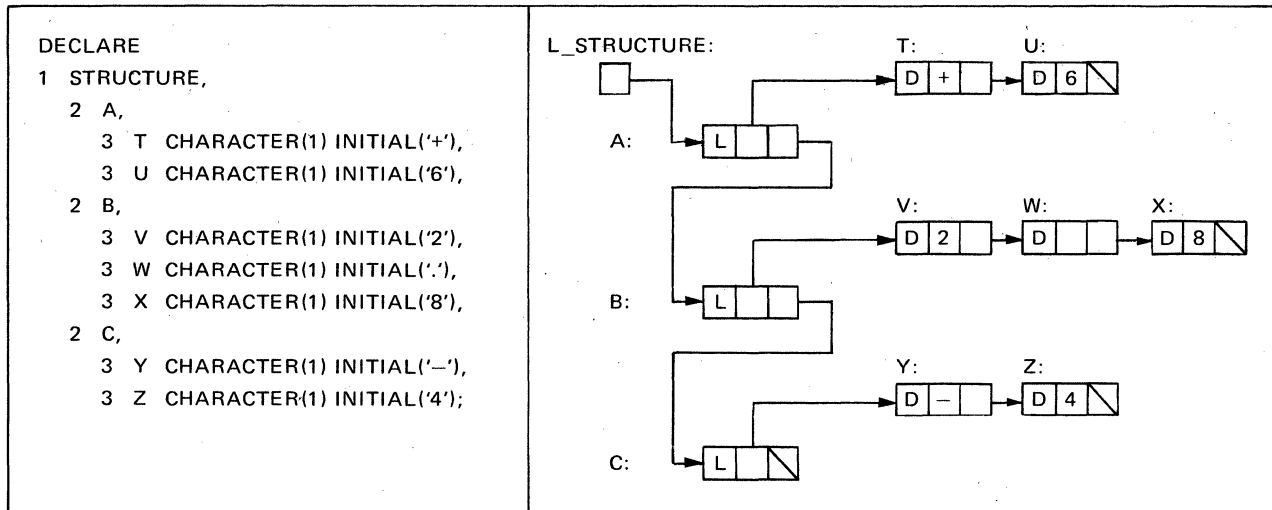


Figure 2.9. A list of lists that represents a PL/I data structure

Sharing List Components Among Lists of Lists

Not only may data items be shared among lists of lists, but list components may be shared as well. Figure 2.10 shows how the components for items X and Y are shared between lists L9 and L10.

This type of sharing eliminates unnecessary duplication of list components as well as data items. Note that the L-component in Figure 2.10 could also be shared between L9 and L10. The organization in Figure 2.10, however, permits the L-component to be deleted from either list without affecting the other list.

Parenthetic Representation of Lists of Lists

A shorter method of showing the organization of a list appears in Figure 2.11, which contains a parenthetic representation of a list of lists. The parenthetic representation contains a sequence of items separated by commas. Parentheses enclose the sequence, and a colon attaches the name of the list to the left parenthesis.

The list presented in Figure 2.11 contains no sublists. Should sublists appear in the list, they are also enclosed in parentheses, as shown in Figure 2.12. Additional levels of sublists are represented by further nesting of parentheses.

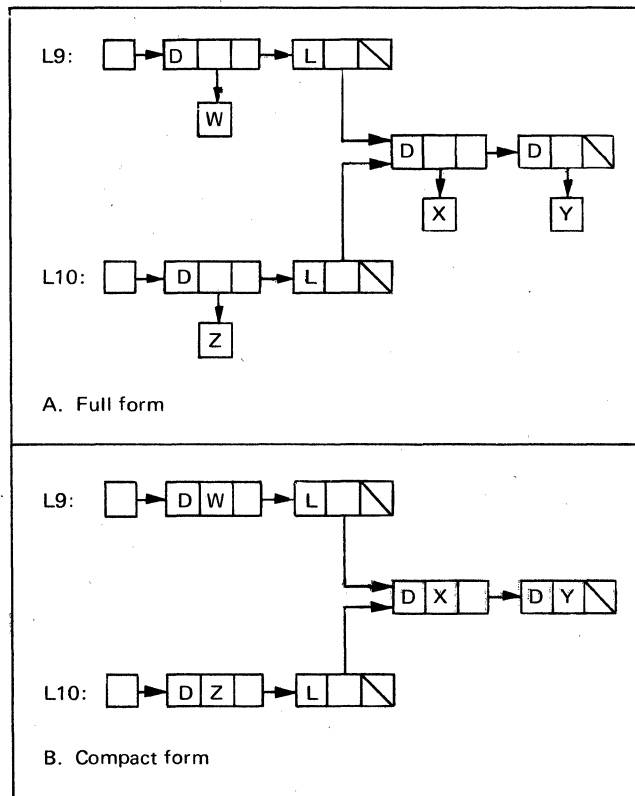


Figure 2.10. Sharing list components among lists of lists

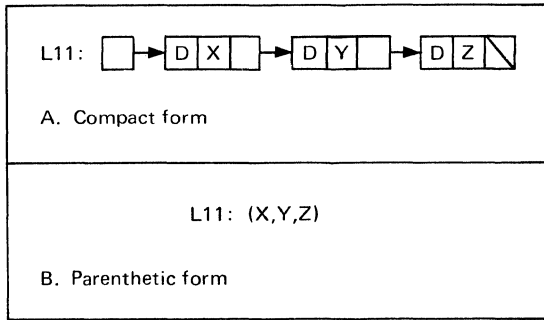


Figure 2.11. Compact and parenthetic representations of a list of lists without sublists

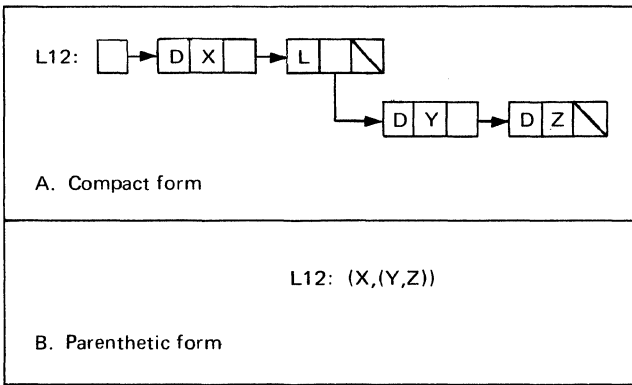


Figure 2.12. Compact and parenthetic representations of a list of lists with a sublist

Circular Lists of Lists

The parenthetic representation of lists is particularly useful in displaying the organization of circular lists. Figure 2.13 contains the parenthetic representation of a circular list of lists that does not contain sublists. An ellipsis (...) indicates the endless cycling of the list, and square brackets ([]) enclose the items that are repeated each cycle.

The parenthetic representation of a circular list of lists that contains a sublist appears in Figure 2.14. Note that square brackets do not denote a sublist but determine the scope of the ellipsis.

The circularities displayed in Figure 2.13 and 2.14 are formed by linking successive list components by means of link pointers. Value pointers can also be used, however, to form circular lists of lists, as shown in Figure 2.15. This type of linkage produces a nested circularity, because the sublist involved links back to a component at a higher level in the list of lists.

Cycling three times through the diagram in Figure 2.15 generates the following list:

L15(X,(Y,Z,(X,(Y,Z,(X,(Y,Z))))))

Such lists are useful in modeling data organizations that have a recursive structure.

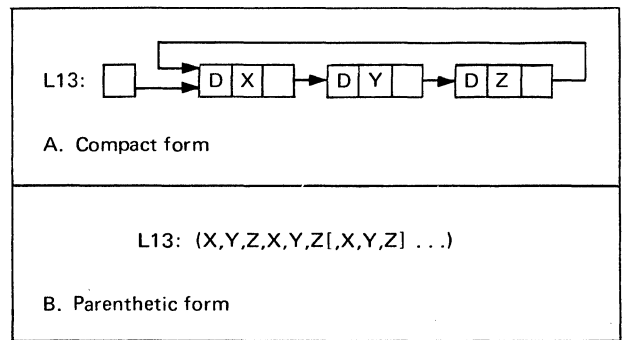


Figure 2.13. A circular list of lists without sublists

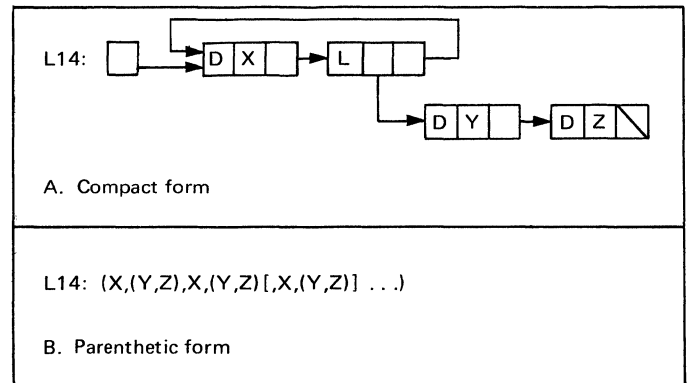


Figure 2.14. A circular list of lists with a sublist

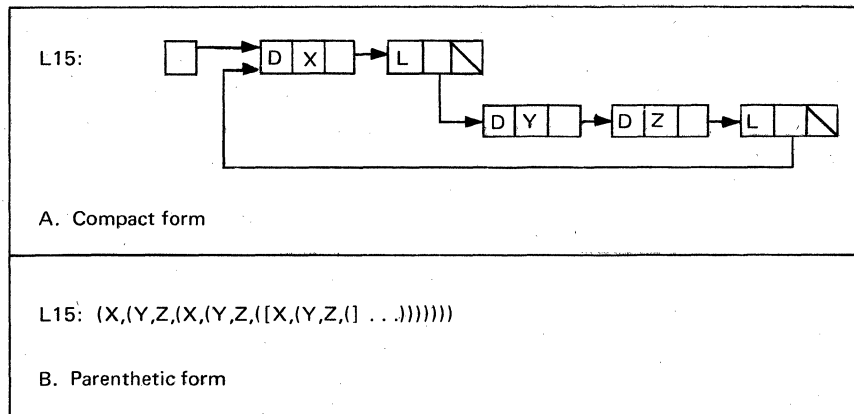


Figure 2.15. A list of lists with a nested circular sublist

PROCESSING LISTS OF LISTS

The following discussions develop subroutines and functions for processing lists of lists. The organization of these procedures resembles the organization used in Chapter 1: elementary procedures are developed first and used in turn to create higher-level procedures.

The procedures are not designed to process lists that contain circularities. Such lists would place many of the procedures—particularly recursive procedures—into endless loops. Circular lists are handled more conveniently on an individual basis.

This section organizes the procedures into four categories:

1. Creating a list of available storage components
2. Manipulating component elements in a list of lists
3. Manipulating the top level of a list of lists
4. Manipulating all levels of a list of lists

No attempt is made to develop an exhaustive collection of procedures; instead, the emphasis is on general methods.

Many of the techniques used in this section have been influenced by the list-processing language LISP, which is in an interpretive programming language developed at the Massachusetts Institute of Technology.*

Creating a List of Available Storage Components

Figure 2.16A, 2.16B, and 2.16C present the AREA_OPEN subroutine for creating a list of available storage components. The subroutine requires two arguments:

1. An area variable throughout which list components are to be allocated
2. A pointer variable that serves as the head of the list of available storage components

The area argument passed to AREA_OPEN can be of any storage class and is not restricted to a particular size, but storage for the area must have been allocated before the subroutine is invoked.

The AREA_OPEN procedure can be used to establish a list of available storage components named LIST. The components can be organized into a list of lists specifying data items. Another invocation of AREA_OPEN can establish a list of available storage components named AVAIL. These latter components relate to insertion and deletion of list components in the list named LIST. The following code is pertinent:

```
DECLARE
  (AREA1, AREA2) AREA,
  (LIST, AVAIL EXTERNAL) POINTER;
CALL AREA_OPEN(AREA1, LIST);
CALL AREA_OPEN(AREA2, AVAIL);
```

A list component deleted from the list named LIST can be inserted into the list named AVAIL. Conversely, a list component can be deleted from the list named AVAIL as needed for insertion into the list named LIST.

This subroutine resembles the similarly named procedure in Chapter 1, except that it creates a list of available storage components for lists of lists.

*Berkeley, Edmund C., and Bobrow, D. G. (editors) *The Programming Language LISP: Its Operation and Applications*. Cambridge, Massachusetts: The M.I.T. Press, 1966 (2nd printing).

AREA_OPEN Subroutine	
Purpose	The LIST argument is assumed to be null upon entry to AREA_OPEN.
To create a list of available storage components	
Reference	Other Programmer-Defined Procedures Required
AREA_OPEN(AREA, LIST)	None
Entry-Name Declaration	Method
DECLARE AREA_OPEN ENTRY(AREA(*), POINTER);	Storage for list components is allocated with the following based structure:
Meaning of Arguments	1 COMPONENT BASED(P),
AREA — the area variable that is to contain the list of available storage components	2 TYPE CHARACTER(1),
LIST — the pointer variable that serves as the head of the list of available storage components	2 VALUE POINTER,
Remarks	2 LINK POINTER,
Storage must have been allocated for the AREA argument before AREA_OPEN is invoked.	Components are allocated throughout AREA until the AREA ON-condition occurs. The LIST argument contains the address of the first component. The LINK element of each component contains the address of the next component. The LINK element of the last component has a null value.

Figure 2.16A. Description of the AREA_OPEN subroutine for creating a list of available storage components

```

AREA_OPEN:
PROCEDURE(AREA, LIST);
DECLARE
  P POINTER,
  AREA AREA(*),
  (LIST, T) POINTER,
  1 COMPONENT BASED(P),
  2 TYPE CHARACTER(1),
  2 VALUE POINTER,
  2 LINK POINTER;

  /* WHEN ALL STORAGE HAS BEEN
  ALLOCATED IN AREA, SET LINK POINTER
  OF LAST COMPONENT, IF ANY, TO NULL
  AND RETURN. */
  ON AREA
BEGIN;
  IF
    P->NULL
  THEN
    P->LINK = NULL;
  GO TO
    END_AREA_OPEN;

  END;
  /* ALLOCATE FIRST COMPONENT IN
  AREA, AND ASSIGN COMPONENT
  ADDRESS TO POINTER PARAMETER CALLED
  LIST. */
  P = NULL;
  ALLOCATE COMPONENT IN(AREA)
  SET(P);
  LIST = P;

  /* CONTINUE ALLOCATING COMPONENTS IN
  AREA UNTIL ALL STORAGE HAS BEEN
  ALLOCATED. LINK EACH COMPONENT TO
  THE PREVIOUSLY ALLOCATED
  COMPONENT. */
  L:
    T = P;
    ALLOCATE COMPONENT IN(AREA)
    SET(P);
    T->LINK = P;
  GO TO
    L;
END_AREA_OPEN:
END
AREA_OPEN;

```

Figure 2.16B. The AREA_OPEN subroutine

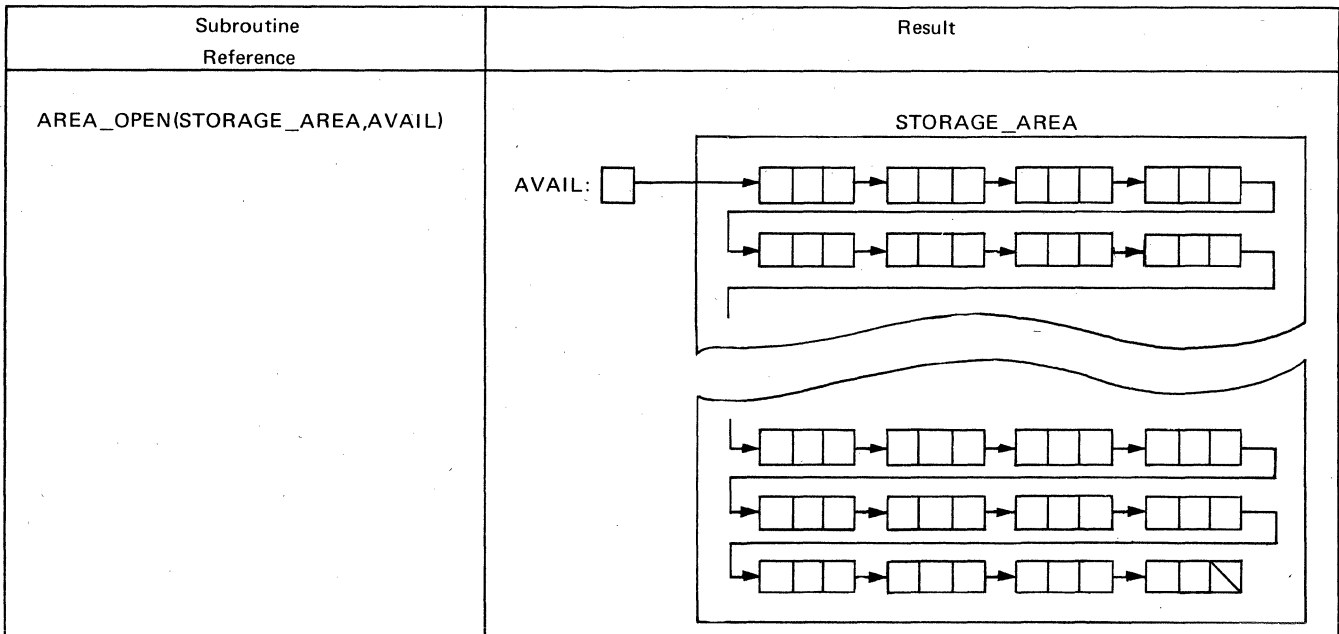


Figure 2.16C. An example of a reference to the AREA_OPEN subroutine

Manipulating Component Elements in a List of Lists

The following discussions develop subroutines and functions for:

1. Obtaining the address of a list component
2. Obtaining the values of elements in list components
3. Assigning values to the elements of list components
4. Comparing the data values of list components

These procedures eliminate the syntactic details associated with PL/I pointer qualification and allow the programmer to view and process lists of lists in a more application-oriented manner.

Obtaining the Address of a List Component

The following discussions develop two function procedures for obtaining the address of a specified list component:

1. ADDRESS_NVT, which obtains the address of the component that contains the nth value at the top level of a list of lists
2. ADDRESS_LVT, which obtains the address of the component that contains the last value at the top level of a list of lists

Later discussions show how to obtain the address of a component that is not at the top level of a list.

ADDRESS_NVT Function

Figures 2.17A, 2.17B, and 2.17C present the ADDRESS_NVT function procedure. This function requires two arguments:

1. A pointer variable that forms the head of the list being processed

2. An integer that indicates the sequential position (first, second, third, etc.) of a value at the top level of the list

The function returns the address of the component that contains the specified value.

ADDRESS_NVT Function	N	— a fixed-point decimal integer value that specifies the component whose address is to be obtained; N has a maximum size of five digits
Purpose		
To obtain the address of the nth component at the top level of a list of lists	Remarks	
Reference		A null pointer value is returned when LIST is null, N is less than one, or N is greater than the number of components at the top level of LIST.
ADDRESS_NVT(LIST, N)		
Entry-Name Declaration		Other Programmer-Defined Procedures Required
DECLARE ADDRESS_NVT ENTRY(POINTER, FIXED DECIMAL(5)) RETURNS(POINTER);		None
	Method	
Meaning of Arguments		The function proceeds through the top level of LIST until the (n-1)th component is reached. The link pointer of this component contains the address of the nth component.
LIST		— the pointer variable that is the head of the list to be examined

Figure 2.17A. Description of the ADDRESS_NVT function for obtaining the address of the nth component at the top level of a list of lists

```

ADDRESS_NVT:
  PROCEDURE(LIST,N)
  RETURNS (POINTER);
  DECLARE
    LIST POINTER,
    (N,I) FIXED DECIMAL(5),
    1 COMPONENT BASED(ADDRESS),
    2 TYPE CHARACTER(1),
    2 VALUE POINTER,
    2 LINK POINTER;
  IF
    (LIST = NULL)|(N<1)
  THEN
    RETURN (NULL);
    ADDRESS = LIST;
  DO
    I = 1 BY 1;
  IF
    (ADDRESS->LINK = NULL) & (I≠N)
  THEN
    RETURN (NULL);
  IF
    I = N
  THEN
    RETURN(ADDRESS);
    ADDRESS = ADDRESS->LINK;
  END;
  END
  ADDRESS_NVT;
  
```

Figure 2.17B. The ADDRESS_NVT function

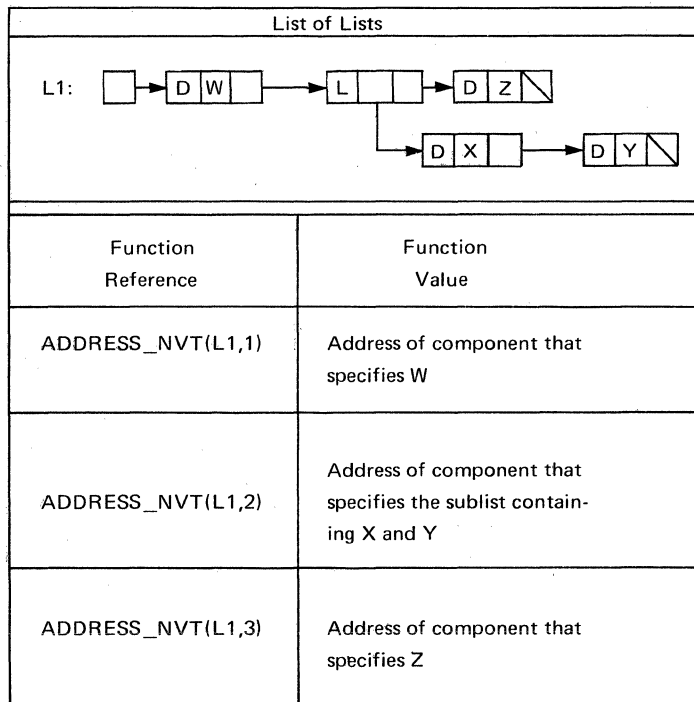


Figure 2.17C. Examples of references to the ADDRESS_NVT function

ADDRESS_LVT Function

Many list-processing operations involve the last value at the top level of a list of lists. It is convenient, therefore, to have a function that specifically obtains the address of the last component at the top level.

Figures 2.18A, 2.18B, and 2.18C present the ADDRESS_LVT function procedure, which returns the address of the last component at the top level of a list. The function requires one argument: the pointer variable that forms the head of the list being processed.

ADDRESS_LVT Function	LIST — the pointer variable that is the head of the list to be examined
Purpose	Remarks
To obtain the address of the last component at the top level of a list of lists	A null pointer value is returned when LIST is null.
Reference	Other Programmer-Defined Procedures Required
ADDRESS_LVT(LIST)	None
Entry-Name Declaration	Method
<pre> DECLARE ADDRESS_LVT ENTRY(POINTER) RETURNS(POINTER); </pre>	The function proceeds through the top level of LIST until the last component is reached. The link pointer in the next-to-last component contains the address of the last component.
Meaning of Argument	

Figure 2.18A. Description of the ADDRESS_LVT function for obtaining the address of the last component at the top level of a list of lists

```

ADDRESS_LVT:
  PROCEDURE(LIST)
  RETURNS (POINTER);
  DECLARE
    ADDRESS POINTER,
    (LIST, SAVE, ADDRESS1) POINTER,
    1 COMPONENT BASED(ADDRESS),
    2 TYPE CHARACTER(1),
    2 VALUE POINTER,
    2 LINK POINTER;
    ADDRESS, ADDRESS1 = LIST;
  DO
    WHILE(ADDRESS1≠NULL);
    SAVE = ADDRESS1;
    ADDRESS1 = ADDRESS1->LINK;
    ADDRESS = SAVE;
  END;
  RETURN(ADDRESS);
END
ADDRESS_LVT;

```

Figure 2.18B. The ADDRESS_LVT function

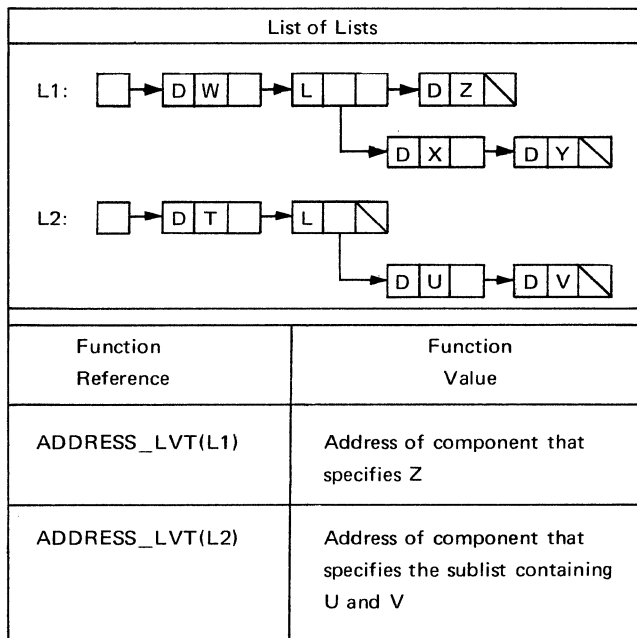


Figure 2.18C. Examples of references to the ADDRESS_LVT function

Obtaining the Values of Elements in List Components

Many list-processing operations examine the values of the elements in list components. The following discussions develop three function procedures for obtaining these values:

1. GET_LINK, which obtains the value of the link pointer in a specified list component
2. GET_VALUE, which obtains the value of the value pointer in a specified list component
3. GET_TYPE, which obtains the value of the type code in a specified list component

GET_LINK Function

Figures 2.19A, 2.19B, and 2.19C present the GET_LINK function, which uses the address of a list component as its argument. The function returns the value of the link pointer in the specified list component. The effect of this function is to obtain the address of the next list component.

GET_VALUE Function

Figures 2.20A, 2.20B, and 2.20C present the GET_VALUE function, which uses the address of a list component as its argument. The function returns the value of the value pointer in the specified list component.

GET_TYPE Function

Figures 2.21A, 2.21B, and 2.21C present the GET_TYPE function, which uses the address of a list component as its argument. The function returns the value of the type code in the specified list component.

GET_LINK Function	
Purpose	ADDRESS – a pointer value that specifies the address of a component in a list of lists
To obtain the address of the next component at the top level of a list of lists	Remarks
Reference	The function assumes that ADDRESS represents a valid address of a component in a list of lists. If ADDRESS is null, a null pointer value is returned.
GET_LINK(ADDRESS)	
Entry-Name Declaration	Method
DECLARE GET_LINK ENTRY(POINTER) RETURNS(POINTER);	The function returns the address contained in the link pointer of the component specified by ADDRESS .
Meaning of Argument	

Figure 2.19A. Description of the GET_LINK function for obtaining the address of the next component at the top level of a list of lists

```

GET_LINK:
  PROCEDURE (ADDRESS)
  RETURNS (POINTER);
DECLARE
  ADDRESS POINTER,
  1 COMPONENT BASED(ADDRESS),
  2 TYPE CHARACTER(1),
  2 VALUE POINTER,
  IF ADDRESS = NULL
  THEN
  RETURN (NULL);
  RETURN(ADDRESS->LINK);
  END
GET_LINK;

```

Figure 2.19B. The GET_LINK function

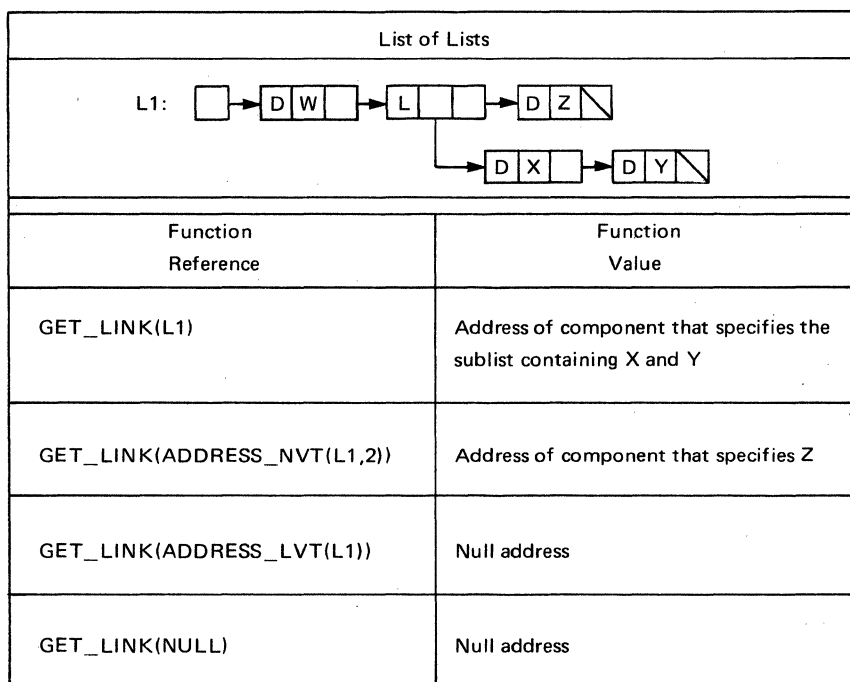


Figure 2.19C. Examples of references to the GET_LINK function

GET_VALUE Function	ADDRESS — a pointer value that specifies the address of a component in a list of lists
Purpose	To obtain the address of the value associated with a component in a list of lists
Reference	GET_VALUE(ADDRESS)
Entry-Name Declaration	DECLARE GET_VALUE ENTRY(POINTER) RETURNS(POINTER);
Meaning of Argument	The function returns the address value of the value pointer in the component specified by ADDRESS.
Remarks	The function assumes that ADDRESS represents a valid address of a component in a list of lists. If ADDRESS is null, a null pointer value is returned.
Other Programmer-Defined Procedures Required	None
Method	

Figure 2.20A. Description of the GET_VALUE function for obtaining the address of the value associated with a component in a list of lists

```

GET_VALUE:                               2 LINK POINTER;
    PROCEDURE (ADDRESS)                    IF
    RETURNS (POINTER);                       ADDRESS = NULL
    DECLARE                                 THEN
    ADDRESS POINTER,                          RETURN (NULL);
    1 COMPONENT BASED(ADDRESS),              RETURN(ADDRESS->VALUE);
    2 TYPE CHARACTER(1),                      END
    2 VALUE POINTER,                           GET_VALUE;

```

Figure 2.20B. The GET_VALUE function

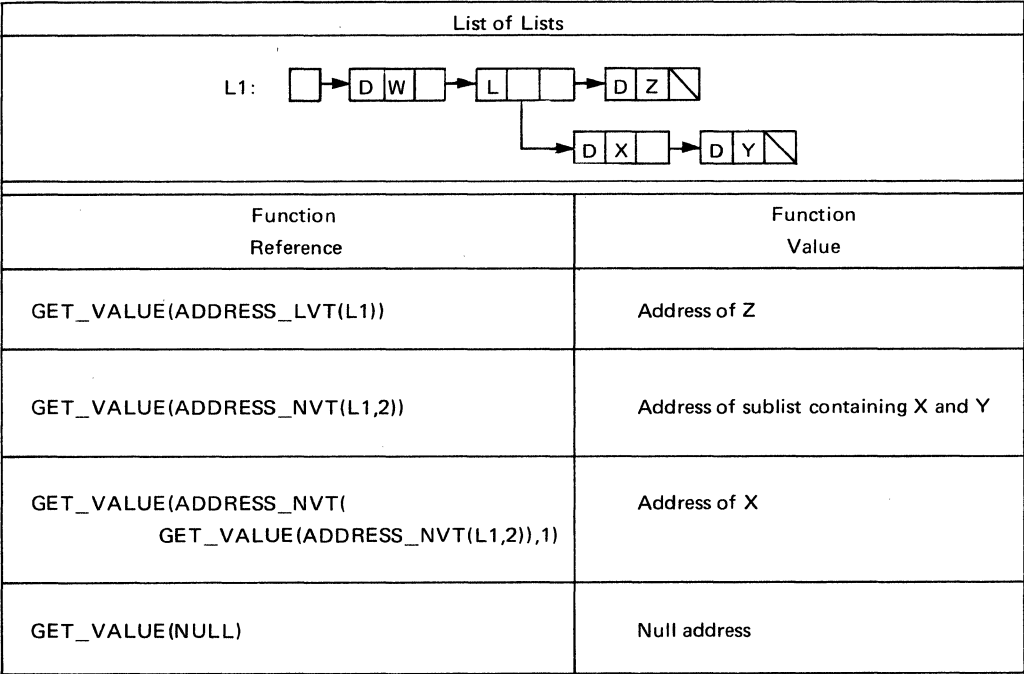


Figure 2.20C. Examples of references to the GET_VALUE function

GET_TYPE Function	ADDRESS — a pointer value that specifies the address of a component in a list of lists
Purpose	
To obtain the type code from a component in a list of lists	Remarks
Reference	The function assumes that ADDRESS represents a valid address of a component in a list of lists. If ADDRESS is null, type code 'D' is returned.
GET_TYPE(ADDRESS)	
Entry-Name Declaration	Other Programmer-Defined Procedures Required
DECLARE GET_TYPE ENTRY(POINTER)	None
RETURNS(CHARACTER(1));	Method
Meaning of Argument	The function returns the value of the type element in the specified component. The value is a single alphameric character.

Figure 2.21A. Description of the GET_TYPE function for obtaining the type code from a component in a list of lists

```

GET_TYPE:
    PROCEDURE(ADDRESS)
    RETURNS (CHARACTER(1));
DECLARE
    ADDRESS POINTER,
    1 COMPONENT BASED(ADDRESS),
    2 TYPE CHARACTER(1),
    2 VALUE POINTER,
                                2 LINK POINTER;
                                IF
                                ADDRESS = NULL
                                THEN
                                RETURN('D');
                                RETURN(ADDRESS->TYPE);
                                END
                                GET_TYPE;

```

Figure 2.21B. The GET_TYPE function

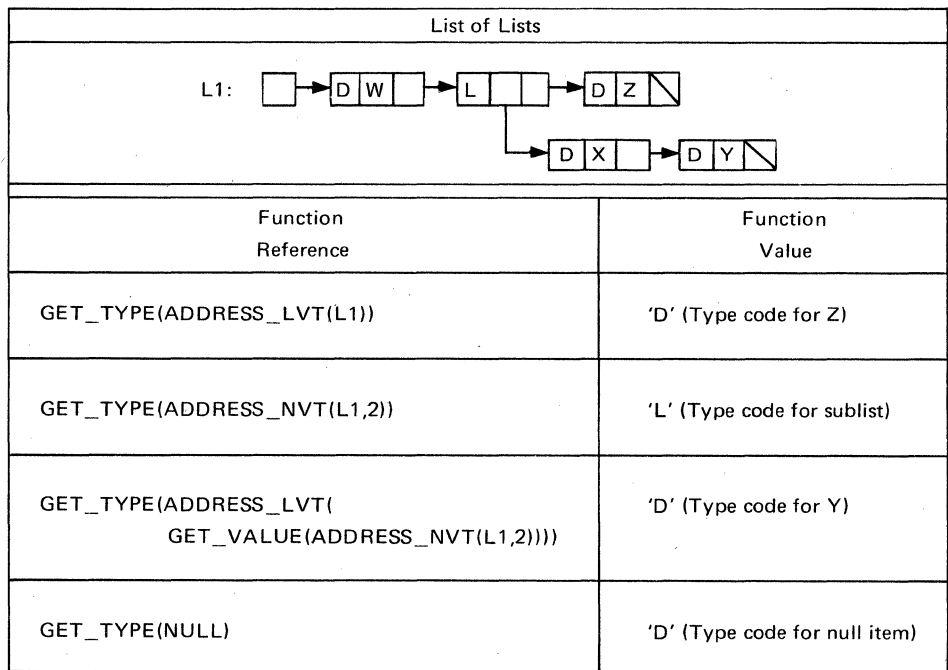


Figure 2.21C. Examples of references to the GET_TYPE function

Assigning Values to Elements of List Components

The link pointers and value pointers of list components must be changed when items are inserted into or deleted from lists of lists. The following discussions develop two subroutines that perform such changes:

1. SET_LINK, which assigns a value to the line pointer in a list component

2. SET_VALUE, which assigns values to the value pointer and type code in a list component

SET_LINK Subroutine

Figures 2.22A, 2.22 B, and 2.22C, present the SET_LINK subroutine, which requires two arguments:

1. Address of a list component
2. Value to be assigned to the link pointer of the specified list component

SET_LINK Subroutine	L	– the value to be assigned to the link pointer of the list component
Purpose		
To assign a value to the link pointer of a component in a list of lists	Remarks	
Reference		The subroutine assumes that ADDRESS represents a valid address of a component in a list of lists. If ADDRESS is null, no assignment is made.
SET_LINK(ADDRESS, L)		
Entry-Name Declaration	Other Programmer-Defined Procedures Required	
DECLARE SET_LINK ENTRY(POINTER, POINTER);	None	
Meaning of Arguments	Method	
ADDRESS – a pointer value that specifies the address of a component in a list of lists		The pointer value of L is assigned to the link pointer of the specified component.

Figure 2.22A. Description of the SET_LINK subroutine for assigning a value to the link pointer of a component in a list of lists

```

SET_LINK:
  PROCEDURE (ADDRESS, L);
  DECLARE
    (ADDRESS, L) POINTER,
    1 COMPONENT BASED(ADDRESS),
    2 TYPE CHARACTER(1),
    2 VALUE POINTER,
    2 LINK POINTER;
  IF
    ADDRESS = NULL
  THEN
    RETURN;
    ADDRESS->LINK = L;
  END
  SET_LINK;
  
```

Figure 2.22B. The SET_LINK subroutine

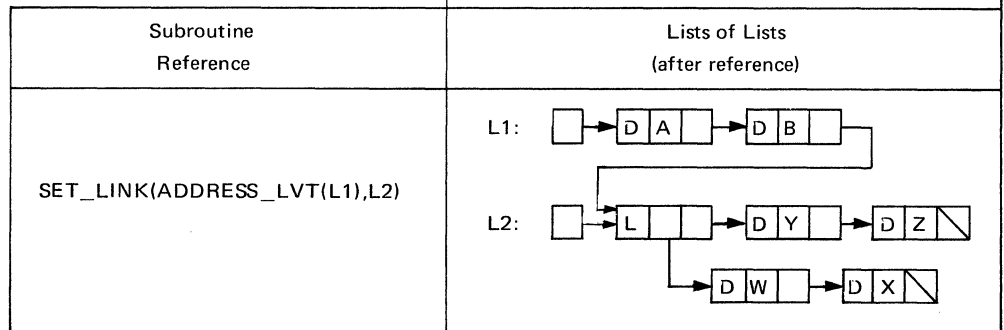
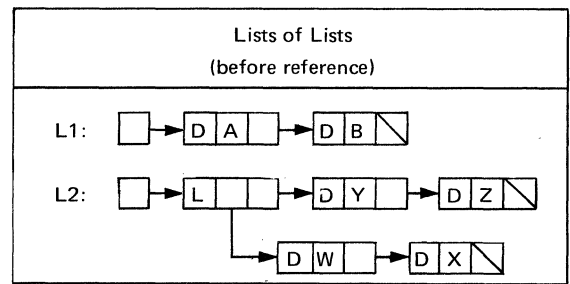


Figure 2.22C. Example of a reference to the SET_LINK subroutine

SET_VALUE Subroutine

Figures 2.23A, 2.23B, and 2.23C present the SET_VALUE subroutine, which requires three arguments:

1. Address of a list component

2. Value to be assigned to the value pointer of the specified list component

3. Value to be assigned to the type code of the specified list component

SET_VALUE Subroutine	V	– a pointer value that specifies the address of the value (data item or sublist) associated with the list component
Purpose	T	– the type code to be assigned to the type element in the list component
To assign an address to the value pointer of a component in a list of lists and also to assign the associated type code	Remarks	The subroutine assumes that ADDRESS represents a valid address of a component in a list of lists. If ADDRESS is null, no assignment is made.
Reference	Other Programmer-Defined Procedures Required	None
SET_VALUE(ADDRESS, V, T)	Method	The pointer value of V is assigned to the value pointer of the specified component. The value of T is converted, if necessary, to a character string, and the leftmost character of the string is assumed to be the type code.
Entry-Name Declaration		
DECLARE SET_VALUE ENTRY(POINTER, POINTER, CHARACTER(1));		
Meaning of Arguments		
ADDRESS – a pointer value that specifies the address of a component in a list of lists		

Figure 2.23A. Description of the SET_VALUE subroutine for assigning an address to the value pointer of a component in a list of lists

```

SET_VALUE:
  PROCEDURE (ADDRESS,V,T);
  DECLARE
    (ADDRESS, V) POINTER,
    T CHARACTER(1),
    1 COMPONENT BASED(ADDRESS),
    IF ADDRESS = NULL THEN
      RETURN;
    2 TYPE CHARACTER(1),
    2 VALUE POINTER,
    2 LINK POINTER;
    IF (T='D') & (T='L') THEN
      RETURN;
      ADDRESS->TYPE = T;
      ADDRESS->VALUE = V;
    END
  SET_VALUE;
  
```

Figure 2.23B. The SET_VALUE subroutine

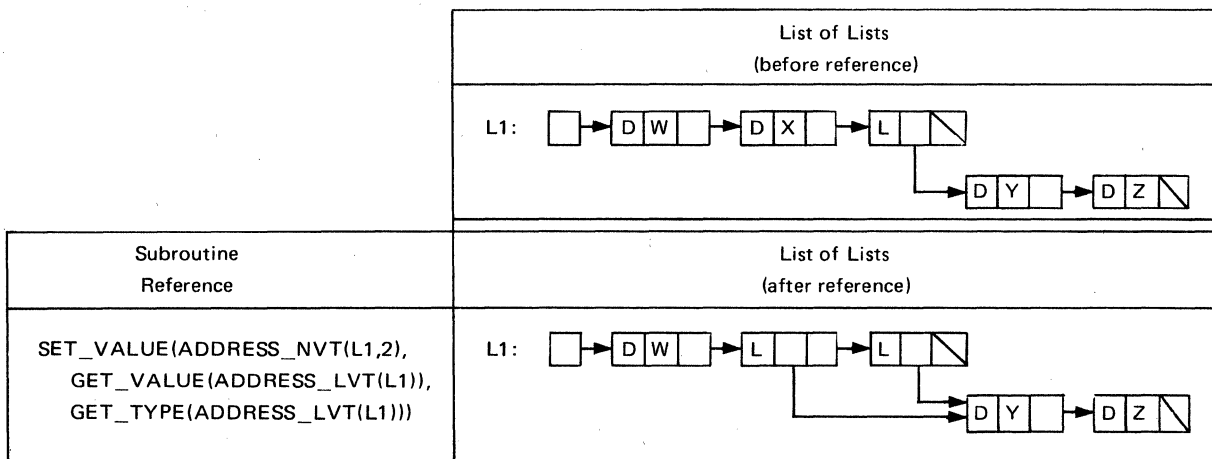


Figure 2.23C. Example of a reference to the SET_VALUE subroutine

Comparing Data Values of List Components

To test the equality of lists of lists, it is often necessary to compare the data values of two list components. The following discussion develops the EQUAL_D function for such comparisons.

EQUAL_D Function

Figures 2.24A, 2.24B, and 2.24C present the EQUAL_D function, which requires two arguments:

1. The first list component involved in the test
2. The second list component involved in the test

The function returns '1'B when the list components contain the same data value, and '0'B when they do not.

EQUAL_D Function	ADDRESS2 — a pointer value that specifies the address of the second list component involved in the test
Purpose	
To test the equality of the data values associated with two list components	Remarks
Reference	The function assumes that ADDRESS1 and ADDRESS2 specify valid addresses of components in lists of lists. When both ADDRESS1 and ADDRESS2 are null, equality is assumed.
EQUAL_D(ADDRESS1, ADDRESS2)	
Entry-Name Declaration	Other Programmer-Defined Procedures Required
DECLARE EQUAL_D ENTRY(POINTER, POINTER) RETURNS(BIT(1));	None
Meaning of Arguments	Method
ADDRESS1 — a pointer value that specifies the address of the first list component involved in the test	For equality, both list components must have type code 'D', and both must have the same value pointer. The function returns '1'B when equality occurs and '0'B when inequality occurs.

Figure 2.24A. Description of the EQUAL_D function for testing the equality of the data values associated with two list components

```

EQUAL_D:
    PROCEDURE(ADDRESS1, ADDRESS2)
    RETURNS (BIT(1));
DECLARE
    (ADDRESS1, ADDRESS2) POINTER;
    IF
        (GET_TYPE(ADDRESS1)= 'D')
        & (GET_TYPE(ADDRESS2)= 'D')
        & (GET_VALUE(ADDRESS1) =
            GET_VALUE(ADDRESS2))
    THEN
        RETURN('1'B);
        RETURN('0'B);
    END
    EQUAL_D;

```

Figure 2.24B. The EQUAL_D function

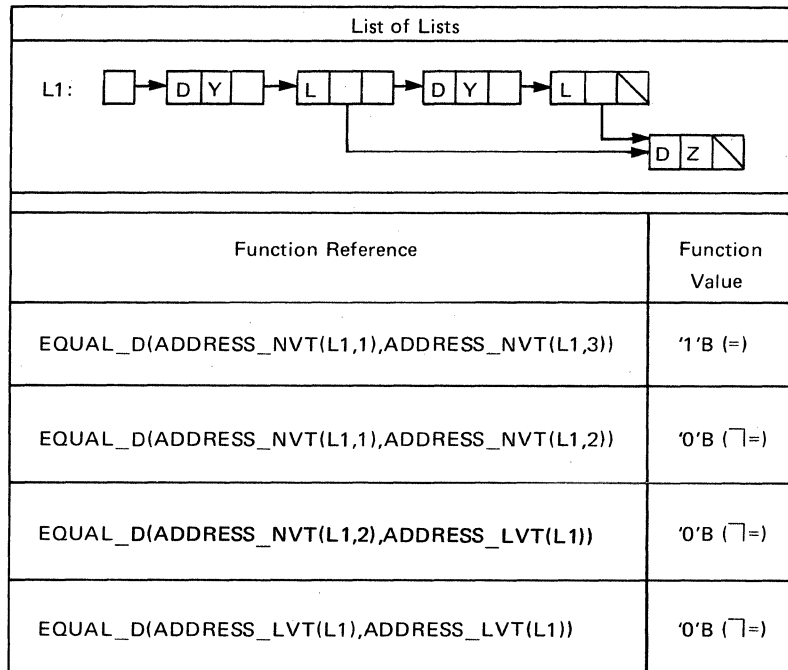


Figure 2.24C. Examples of references to the EQUAL_D function

Manipulating Top Level of a List of Lists

All items in a list of lists may be processed by proceeding through the list in stages. With this approach, the first stage is restricted to the data items and sublists situated at the top level of the list. The second stage then deals with the top level of each sublist encountered during the first stage. Subsequent stages, in general, are concerned only with the sublists of the previous stage. When all sublists have been treated in this manner, processing for the entire list is complete.

The following discussions develop subroutines and functions for manipulating the top level of a list of lists. These procedures are concerned with the following operations:

1. Counting the number of values at the top level of a list of lists
2. Inserting values into the top level of a list of lists

3. Obtaining values and their type codes from the top level of a list of lists
4. Combining lists of lists at the top level
5. Copying the top level of a list of lists in reverse order

Counting Number of Values at Top Level of a List of Lists

The size of a list of lists depends in part upon the number of values at the top level of the list. The following discussion develops the SIZE_TOP function, which counts the values at the top level of a list.

SIZE_TOP Function

Figures 2.25A, 2.25B, and 2.25C present the SIZE_TOP function, which requires the name of a list as its only argument. The function returns a count of the data items and sublists at the top level of the specified list.

SIZE_TOP Function

Purpose

To count the number of values at the top level of a list of lists

Reference

SIZE_TOP(LIST)

Entry-Name Declaration

```

DECLARE SIZE_TOP ENTRY(POINTER)
        RETURNS(FIXED
        DECIMAL(5));

```

Meaning of Argument

LIST — the pointer variable that is the head of the list to be examined

Remarks

The maximum size is 99999. If LIST is null, a zero size is returned.

Other Programmer-Defined Procedures Required

GET_LINK

Method

The function proceeds through the top level of LIST, counting the number of list components, until a null link pointer is encountered. The top level of LIST can contain any combination of data (D) values and list (L) values.

Figure 2.25A. Description of the SIZE_TOP function for counting the number of values at the top level of a list of lists

```

SIZE_TOP:
PROCEDURE (LIST)
RETURNS(FIXED DECIMAL(5));
DECLARE
(LIST,ADDRESS) POINTER,
N FIXED DECIMAL(5);
ADDRESS = LIST;
DO
N = 0 BY 1;
IF
ADDRESS = NULL
THEN
RETURN(N);
ADDRESS = GET_LINK(ADDRESS);
END;
END
SIZE_TOP;

```

Figure 2.25B. The SIZE_TOP function

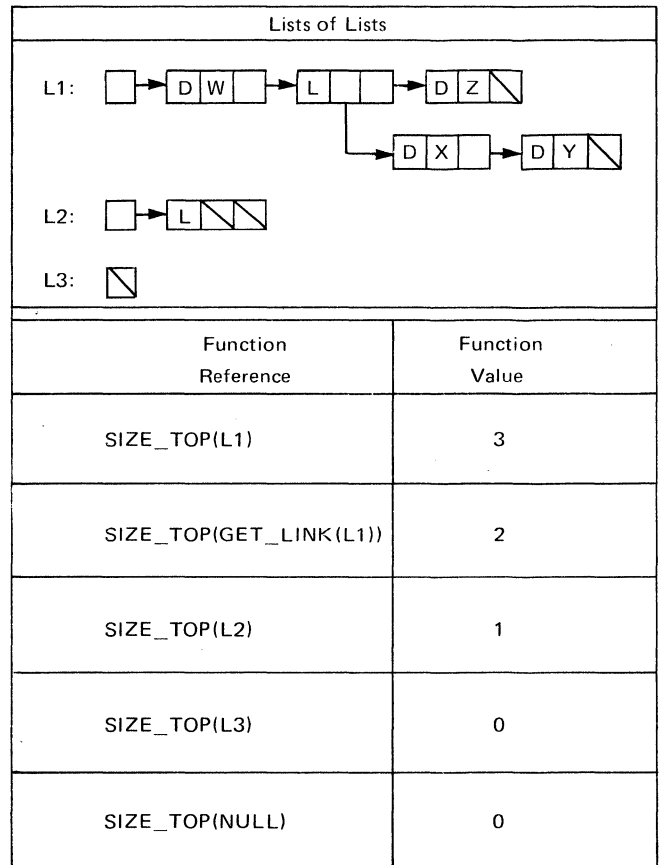


Figure 2.25C. Examples of references to the SIZE_TOP function

Inserting Values Into Top Level of a List of Lists

The creation of a list of lists requires the ability to insert data items and sublists into the list. The following discussions develop two procedures that perform such insertions at the top level of a list of lists:

1. `INSERT_NVT`, which inserts a value and its type code into a specified position at the top level of a list
2. `FORM_BODY`, which forms the body of a new list by extending the front of a given list with a specified value

INSERT_NVT Subroutine

Figures 2.26A, 2.26B, and 2.26C present the `INSERT_NVT` subroutine, which requires four arguments:

1. List in which an insertion is to be made
2. Position of the insertion at the top level of the list
3. Address of the value to be inserted
4. Type code of the value to be inserted

<code>INSERT_NVT</code> Subroutine	V	– the address of the value to be inserted
Purpose	T	– the type code ('D' or 'L') of the value to be inserted
To insert a value into the nth position at the top level of a list of lists		Remarks
Reference		When the list is null or N is less than two, V is inserted into the first position at the top level of the list. When N exceeds the size of the top level of the list, V is inserted into the last position. N cannot have a value greater than 99999. V may be null.
<code>INSERT_NVT(LIST, N, V, T)</code>		
Entry-Name Declaration		Other Programmer-Defined Procedures Required
<code>DECLARE INSERT_NVT ENTRY(POINTER, FIXED DECIMAL(5), POINTER, CHARACTER(1));</code>		<code>SET_VALUE, SET_LINK, ADDRESS_NVT, and ADDRESS_LVT</code>
Meaning of Arguments		Method
LIST – the pointer variable that is the head of the list to be processed		Insertion of a value causes the size of the top level to increase by 1. The value previously at the nth position becomes the (n+1)th value at the top level.
N – the position at the top level of the list where the value is to be inserted		

Figure 2.26A. Description of the `INSERT_NVT` subroutine for inserting a value into the nth position at the top level of a list of lists


```

INSERT_NVT:
  PROCEDURE(LIST,N,V,T);
  DECLARE
    T CHARACTER(1),
    N FIXED DECIMAL(5),
    (LIST,V,ADDRESS1,ADDRESS2,
    AVAIL EXTERNAL) POINTER;
    /* IF LIST OF AVAILABLE STORAGE
    COMPONENTS IS EMPTY, PRINT MESSAGE
    AND RETURN. */
    IF AVAIL = NULL THEN DO;
  PUT
    LIST('LIST OF AVAILABLE STORAGE IS
    EMPTY');
    RETURN; END;
    /* ASSIGN VALUE V AND TYPE T TO
    FIRST COMPONENT IN AVAIL. */
    CALL SET_VALUE(AVAIL,V,T);
    /* IF LIST IS NULL OR N<2, INSERT
    FIRST COMPONENT OF AVAIL INTO
    FIRST POSITION OF LIST, AND
    RETURN. */
  IF
    (LIST = NULL)|(N<2)
    THEN DO;
    ADDRESS1 = LIST; LIST = AVAIL;
    AVAIL = ADDRESS_NVT(AVAIL,2);
    CALL SET_LINK(LIST,ADDRESS1);
    RETURN; END;
    /* OTHERWISE OBTAIN THE ADDRESS OF
    THE N-TH COMPONENT AT THE TOP OF
    LIST. */
    ADDRESS2 = ADDRESS_NVT(LIST,N);
    /* IF N EXCEEDS SIZE OF LIST TOP,
    OBTAIN ADDRESS OF LAST COMPONENT AT
    TOP, ELSE OBTAIN ADDRESS OF (N-1)
    COMPONENT AT TOP..*/
  IF
    ADDRESS2 = NULL
    THEN
      ADDRESS1 = ADDRESS_LVT(LIST);
    ELSE
      ADDRESS1 = ADDRESS_NVT(LIST,N-1);
      /* INSERT FIRST COMPONENT OF AVAIL
      INTO THE N-TH POSITION AT THE TOP
      OF LIST. */
      CALL SET_LINK(ADDRESS1,AVAIL);
      ADDRESS1 = AVAIL;
      AVAIL = ADDRESS_NVT(AVAIL,2);
      CALL SET_LINK(ADDRESS1,ADDRESS2);
  END INSERT_NVT;

```

Figure 2.26B. The INSERT_NVT subroutine

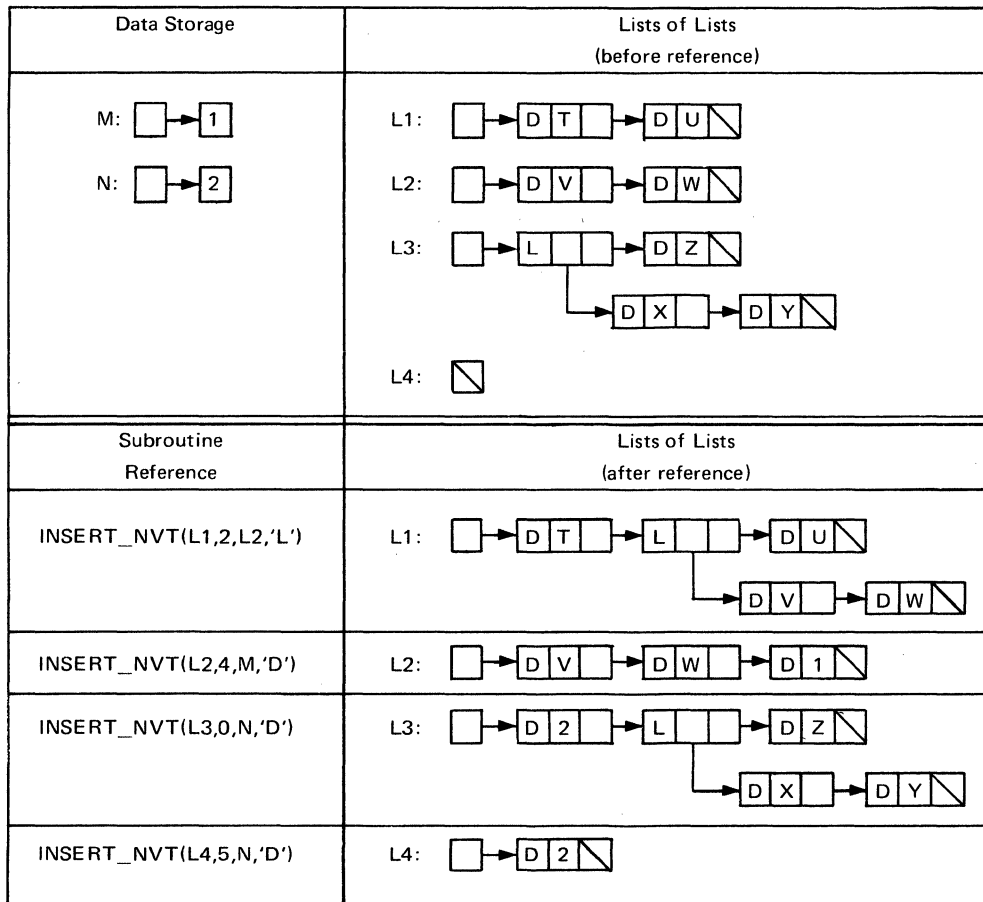


Figure 2.26C. Examples of references to the INSERT_NVT subroutine

FORM_BODY Function

Figures 2.27A, 2.27B, and 2.27C present the FORM_BODY function, which requires three arguments:

1. Address of the value to be inserted at the front of a list
2. Type code of the value to be inserted
3. List in which the insertion is to be made

The function returns the address of the new first component in the list, but the head pointer of the original list does not

receive a new value; the head points to the second component at the top level of the new list. If the head is to point to the first component, the address value of the function must be assigned explicitly to the head.

Because the head of the old list is not modified automatically by FORM_BODY, the function can be thought of as forming only the body of a new list. In fact, the built-in function NULL can be used to specify the list in which insertion is to occur. In this case, no explicit list head is involved, and FORM_BODY generates the body of a new list that contains one list component.

FORM_BODY Function	T	– the type code ('D' or 'L') of the value inserted at the front of the new list
Purpose	LIST	– the pointer variable that is the head of the list to be extended at the front with the new value
To form the body of a new list of lists by extending the front of a given list with a specified value, and also to obtain the address of the first list component in the new list	Remarks	The effect of this function is equivalent to inserting a value into the first position of LIST, except that the pointer value of LIST is not changed; instead, the address of the first list component in the resulting list is returned as the function value.
Reference	Other Programmer-Defined Procedures Required	SET_VALUE, GET_LINK, and SET_LINK
FORM_BODY(V, T, LIST)	Method	V and T are inserted into a new list component, which is linked to the front of the components in LIST. The pointer value of LIST is not changed.
Entry-Name Declaration		
DECLARE FORM_BODY ENTRY(POINTER, CHARACTER(1), POINTER) RETURNS(POINTER);		
Meaning of Arguments		
V		– the address of the value to be inserted at the front of the new list

Figure 2.27A. Description of the FORM_BODY function for forming the body of a new list of lists and returning the address of the first list component

```

FORM_BODY:
  PROCEDURE (V, T, LIST)
    RETURNS (POINTER);
  DECLARE
    T CHARACTER(1),
    (V,LIST,AVAIL EXTERNAL, P) POINTER;
    /* IF LIST OF AVAILABLE STORAGE
    COMPONENTS IS EMPTY, PRINT MESSAGE
    AND RETURN. */
  IF
    AVAIL = NULL
  THEN
  DO;
    PUT
      LIST('LIST OF AVAILABLE STORAGE IS
      EMPTY');
    END;
  RETURN;
  /* LET P POINT TO FIRST COMPONENT
  OF AVAIL LIST, AND LET AVAIL POINT
  TO SECOND COMPONENT OF AVAIL LIST.*/
  P = AVAIL;
  AVAIL = GET_LINK(AVAIL);
  /* SET VALUE POINTER OF P COMPONENT
  EQUAL TO V, AND SET LINK POINTER OF
  P COMPONENT EQUAL TO LIST. */
  CALL SET_VALUE(P,V,T);
  CALL SET_LINK(P,LIST);
  /* RETURN ADDRESS OF P COMPONENT. */
  RETURN(P);
FORM_BODY;

```

Figure 2.27B. The FORM_BODY function





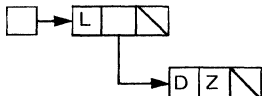
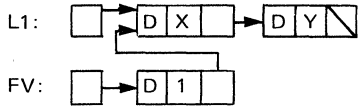
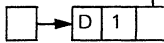
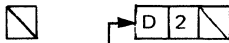

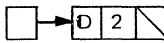
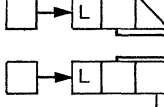

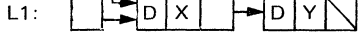

	Data Storage	Lists of Lists (before reference)
	M:  N: 	L1:  L2:  L3: 
Function Reference	Function Value (FV)	Lists of Lists (after reference)
FORM_BODY(M,'D',L1)	Address of component created for '1'	L1:  FV: 
FORM_BODY(N,'D',L2)	Address of component created for '2'	L2:  FV: 
FORM_BODY(N,'D',NULL)	Address of component created for '2'	FV: 
FORM_BODY(GET_VALUE(L3),'L',L1)	Address of component created for duplication of value pointer in first component at top level of L3	L3:  FV:  L1:  FV: 

Figure 2.27C. Examples of references to the FORM_BODY function

Obtaining Values and Their Type Codes from Top Level of a List of Lists

The following discussions develop two functions for obtaining values from the top level of a list of lists:

1. GET_NVT, which gets the nth value at the top level of a list of lists
2. GET_NTT, which gets the nth type code at the top level of a list of lists

GET_NVT Function

Figures 2.28A, 2.28B, and 2.28C present the GET_NVT function, which uses two arguments:

1. Name of a list
2. Position of a value at the top level of the list

The function returns the address value of the value pointer in the specified position at the top level of the list.

GET_NVT Function

Purpose
To obtain the address of the value associated with the nth component at the top level of a list of lists

Reference
GET_NVT(LIST, N)

Entry-Name Declaration

```

DECLARE GET_NVT ENTRY(POINTER, FIXED
DECIMAL(5))
RETURNS(POINTER);
        
```

Meaning of Arguments

LIST – the pointer variable that is the head of the list to be processed

N – the position of the retrieved value at the top level of the list

Remarks
A value of N less than one or greater than the size of the top level of the list causes a null address to be returned.

Other Programmer-Defined Procedures Required
ADDRESS_NVT and GET_VALUE

Method
The following reference obtains the nth value at the top level of the list:


```

GET_VALUE(ADDRESS_NVT(LIST, N))
        
```

The nth value at the top level remains in the list after its equivalent is returned.

```

GET_NVT:
  PROCEDURE (LIST, N)
  RETURNS (POINTER);
  DECLARE
    LIST POINTER,
    N FIXED DECIMAL(5);
  RETURN(GET_VALUE(ADDRESS_NVT
(LIST,N)));
  END
  GET_NVT;
    
```

Figure 2.28B. The GET_NVT function

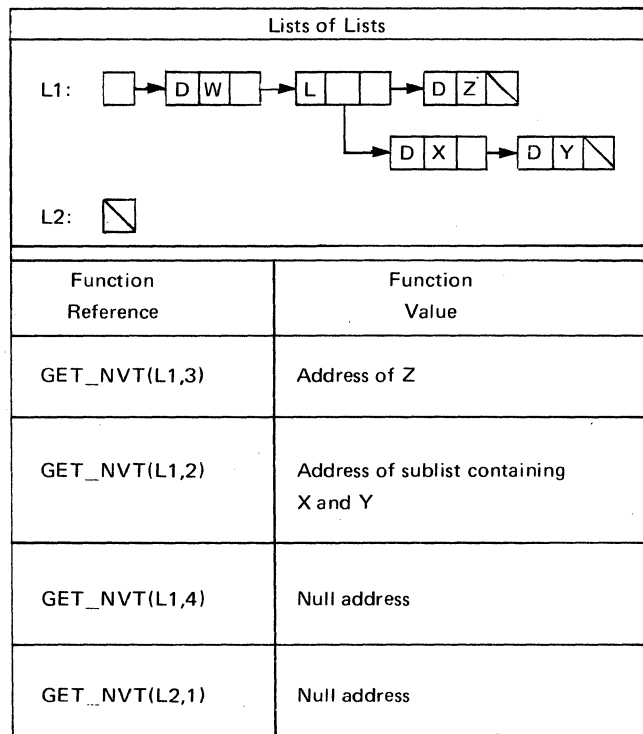


Figure 2.28C. Examples of references to the GET_NVT function

Figure 2.28A. Description of the GET_NVT function for obtaining the nth value at the top level of a list of lists

GET_NTT Function

Figures 2.29A, 2.29B, and 2.29C present the GET_NTT function, which uses two arguments:

1. Name of a list
2. Position of a type code at the top level of the list

The function returns the type code in the specified position at the top level of the list.

GET_NTT Function

Purpose
To obtain the type code ('D' or 'L') of the nth value at the top level of a list of lists

Reference
GET_NTT(LIST, N)

Entry-Name Declaration

```

DECLARE GET_NTT ENTRY(POINTER, FIXED
DECIMAL(5))
RETURNS(CHARACTER
(1));
    
```

Meaning of Arguments

LIST — the pointer variable that is the head of the list to be processed

N — the position of the retrieved type code at the top level of the list

Remarks
A value of N less than one or greater than the size of the top level of the list causes the type code 'D' to be returned.

Other Programmer-Defined Procedures Required
ADDRESS_NVT and GET_TYPE

Method
The following reference obtains the nth type code at the top level of the list:

```

GET_TYPE(ADDRESS_NVT(LIST, N))
    
```

 The nth type code at the top level remains in the list after its equivalent is returned.

```

GET_NTT:
  PROCEDURE (LIST, N)
  RETURNS (CHARACTER(1));
  DECLARE
    LIST POINTER,
    N FIXED DECIMAL(5);
  RETURN(GET_TYPE(ADDRESS_NVT
(LIST,N)));
  END
  GET_NTT;
    
```

Figure 2.29B. The GET_NTT function

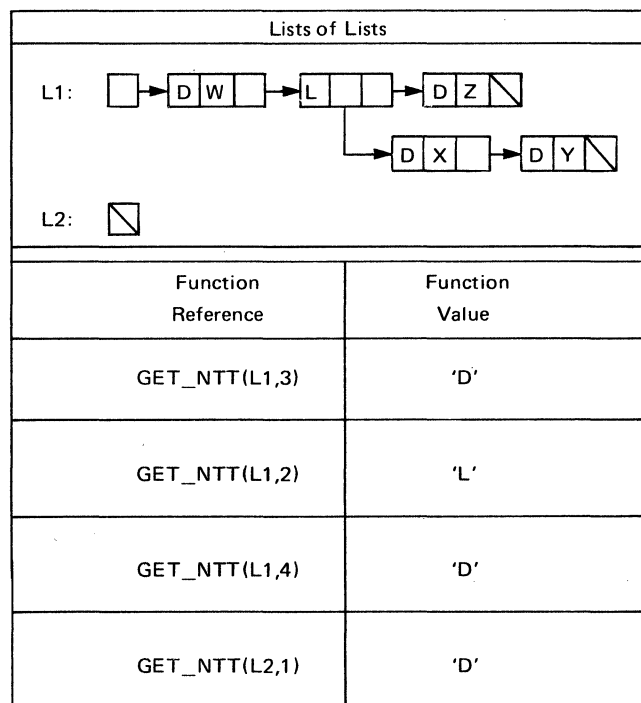


Figure 2.29C. Examples of references to the GET_NTT function

Figure 2.29A. Description of the GET_NTT function for obtaining the nth type code at the top level of a list of lists

Combining Lists of Lists at Top Level

A list can be extended by combining it at the top level with another list. The following discussions develop two procedures for such extensions:

1. LINKL, which links two lists of lists at the top level
2. APPEND, which forms a new list of lists by duplicating the top level of one list and linking the top level of another list behind the duplicate

LINKL Subroutine

Figures 2.30A, 2.30B, and 2.30C present the LINKL subroutine, which requires the names of two lists as its arguments. The subroutine links the last component at the top level of the first list to the first component at the top level of the second list.

LINKL Subroutine	Remarks
Purpose	When LIST1 is null, it becomes equal to LIST2.
To link two lists of lists at the top level	When LIST2 is null, LIST1 does not change.
Reference	Other Programmer-Defined Procedures Required
LINKL(LIST1, LIST2)	ADDRESS_LVT and SET_LINK
Entry-Name Declaration	Method
DECLARE LINKL ENTRY(POINTER, POINTER);	The following reference links the lists:
Meaning of Arguments	SET_LINK(ADDRESS_LVT(LIST1), LIST2)
LIST1 – the first list to be linked	No list components are duplicated, and the pointer values of LIST1 and LIST2 are not changed.
LIST2 – the list to be linked behind LIST1	

Figure 2.30A. Description of the LINKL subroutine for linking two lists of lists at the top level

```
LINKL:
  PROCEDURE(LIST1,LIST2);
  DECLARE
    (LIST1,LIST2) POINTER;
  IF
    LIST1 = NULL
  THEN
    LIST1 = LIST2;
  ELSE
    CALL SET_LINK(ADDRESS_LVT(LIST1),
      LIST2);
  END
  LINKL;
```

Figure 2.30B. The LINKL subroutine

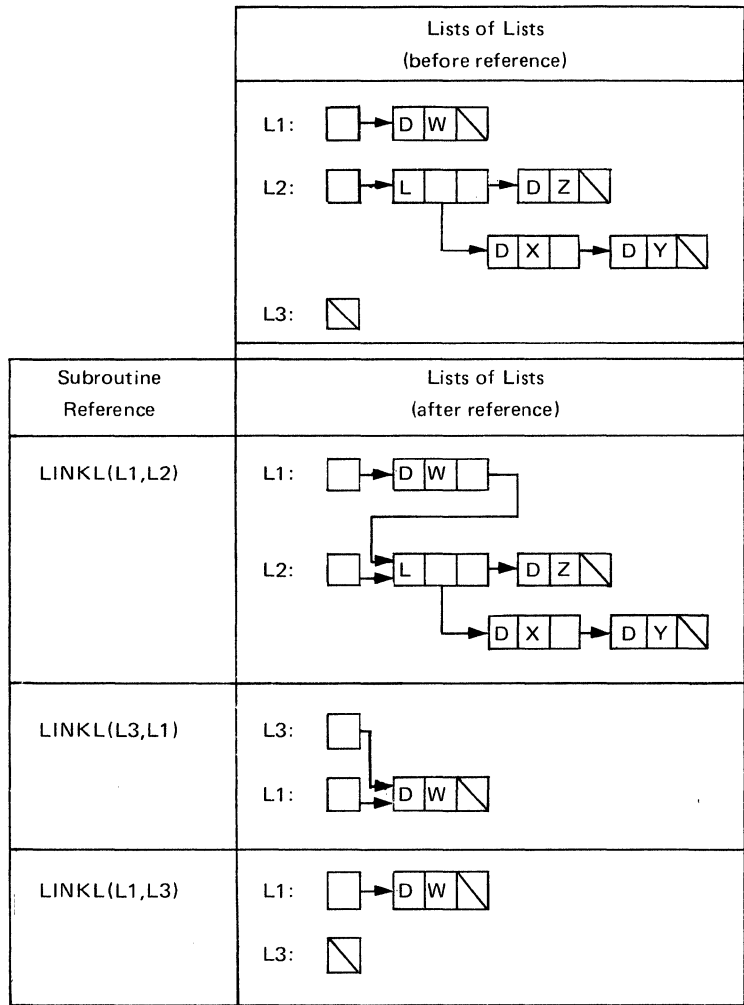


Figure 2.30C. Examples of references to the LINKL subroutine

APPEND Function

Figures 2.31A, 2.31B, and 2.31C present the APPEND function, which uses the names of two lists as its arguments. This function duplicates the top level of the first list and links the top level of the second list behind the duplicate. The function then returns the address of the first component in the new list.

The APPEND function, unlike the previous procedure (LINKL), preserves the first list as an entity.

APPEND Function	LIST2 — the list to be appended behind the duplicate top level
Purpose To form a new list of lists by duplicating the top level of one list and appending the top level of another list behind the duplicate top level, and also to obtain the address of the first list component in the new list.	Remarks When LIST1 is null, the address of the first list component in LIST2 is returned. The components in LIST2 are never duplicated. The address values of pointers LIST1 and LIST2 remain unchanged.
Reference APPEND(LIST1, LIST2)	Other Programmer-Defined Procedures Required FORM_BODY, GET_VALUE, GET_TYPE, and GET_LINK
Entry-Name Declaration DECLARE APPEND ENTRY (POINTER, POINTER) RETURNS (POINTER);	Method The following recursive reference forms the new list:
Meaning of Arguments LIST1 — the list whose top level is to be duplicated	FORM_BODY(GET_VALUE(LIST1), GET_TYPE(LIST1), APPEND (GET_LINK(LIST1), LIST2))

Figure 2.31A. Description of the APPEND function

```
APPEND:
        PROCEDURE (LIST1, LIST2)
          RETURNS (POINTER)
          RECURSIVE;
DECLARE
        (LIST1,LIST2) POINTER;
IF
        LIST1 = NULL
THEN
        RETURN(LIST2);
        RETURN (FORM_BODY(GET_VALUE(LIST1),
        GET_TYPE(LIST1),
        APPEND(GET_LINK(LIST1), LIST2)));
END
        APPEND;
```

Figure 2.31B. The APPEND function

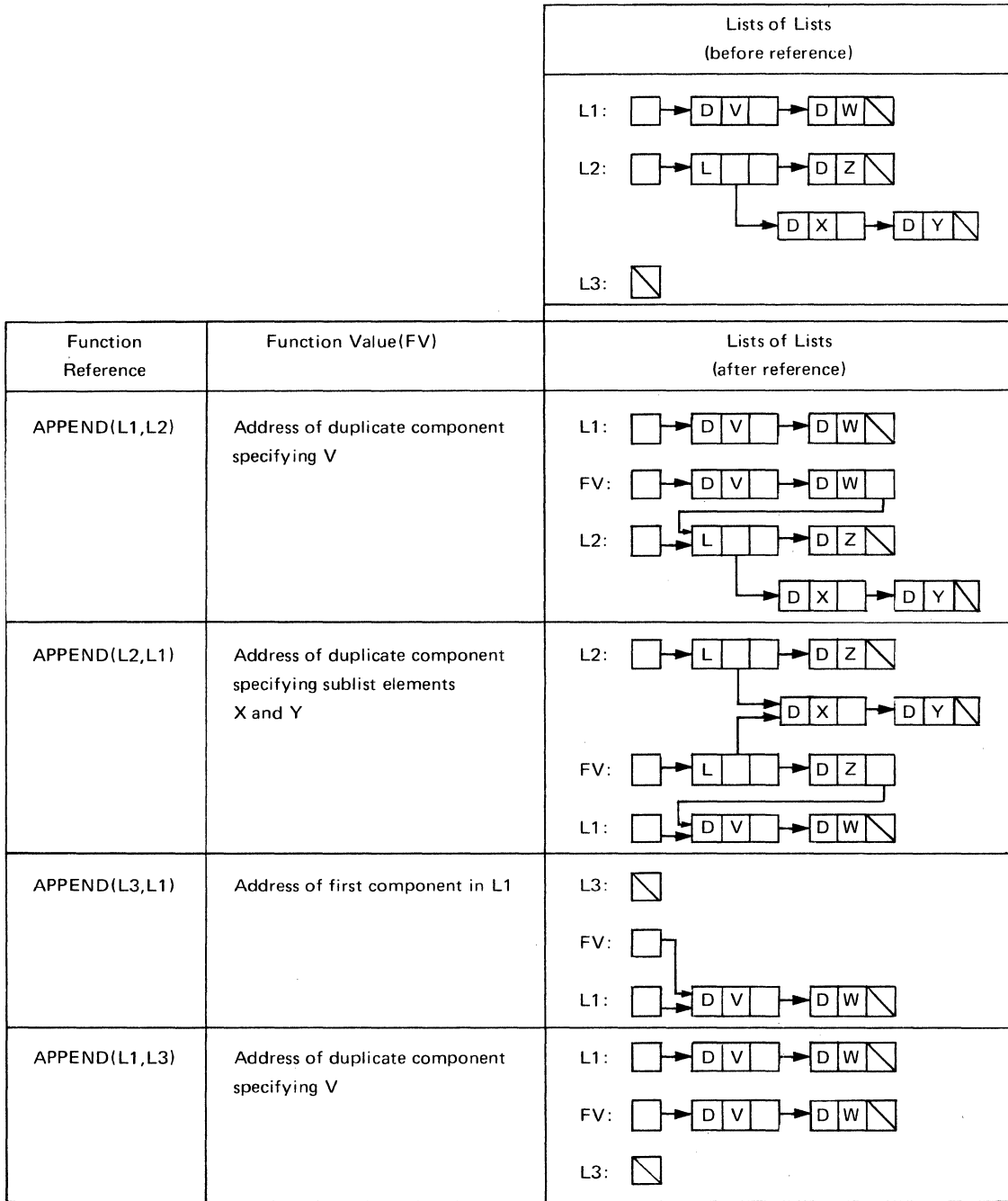


Figure 2.31C. Examples of references to the APPEND function

Copying Top Level of a List of Lists in Reverse Order

It usually takes less time to retrieve an item from the front of a list than from the end, because fewer items have to be traversed to reach the desired item. When more processing occurs at the end of list than at the front, it may be more efficient to reverse the list.

The following discussions develop two subroutines for reversing a list of lists:

1. COPY_REVT, which copies the top level of a list of lists in reverse order and returns the address of the new list
2. COPY_REVT1, which is the recursive equivalent of COPY_REVT

Generating a new top-level list in reverse order generally takes less time than relinking the top level of the original list.

COPY_REVT Function

Figures 2.32A, 2.32B, and 2.32C present COPY_REVT, which requires the name of a list as its only argument. The function returns the address of the new list, which has been reversed at the top level.

Note that the function does not produce a distinct copy of the new list. Both the new and old lists share components at lower levels.

COPY_REVT Function	Remarks
<p>Purpose</p> <p>To copy the top level of a list of lists in reverse order and to return the address of the new list</p>	<p>Components at lower levels of LIST are not copied but are shared between the old and new lists. When LIST is null, a null address is returned.</p>
<p>Reference</p> <p>COPY_REVT(LIST)</p>	<p>Other Programmer-Defined Procedures Required</p> <p>FORM_BODY, GET_VALUE, GET_TYPE, and GET_LINK</p>
<p>Entry-Name Declaration</p> <pre> DEclare COPY_REVT ENTRY(POINTER) RETURNs(POINTER); </pre>	<p>Method</p> <p>GET_LINK obtains successive addresses of list components at the top level of LIST.</p>
<p>Meaning of Argument</p> <p>LIST — the list whose top level is to be copied in reverse order</p>	<p>GET_VALUE and GET_TYPE obtain the value and type of each list component.</p> <p>FORM_BODY creates and links the components in the new top-level list.</p>

Figure 2.32A. Description of the COPY_REVT function for copying the top level of a list of lists in reverse order and returning the address of the new list.

```

COPY_REVT:
  PROCEDURE (LIST)
  RETURNS (POINTER);
DECLARE
  (LIST, ADDRESS1, ADDRESS2) POINTER;
  ADDRESS1 = LIST;
  ADDRESS2 = NULL;
L:
  IF
    ADDRESS1 = NULL
  THEN
    RETURN(ADDRESS2);
    ADDRESS2 = FORM_BODY(GET_VALUE
      (ADDRESS1),
      GET_TYPE(ADDRESS1),
      ADDRESS2);
    ADDRESS1 = GET_LINK(ADDRESS1);
  GO TO
    L;
END
COPY_REVT;

```

Figure 2.32B. The COPY_REVT function

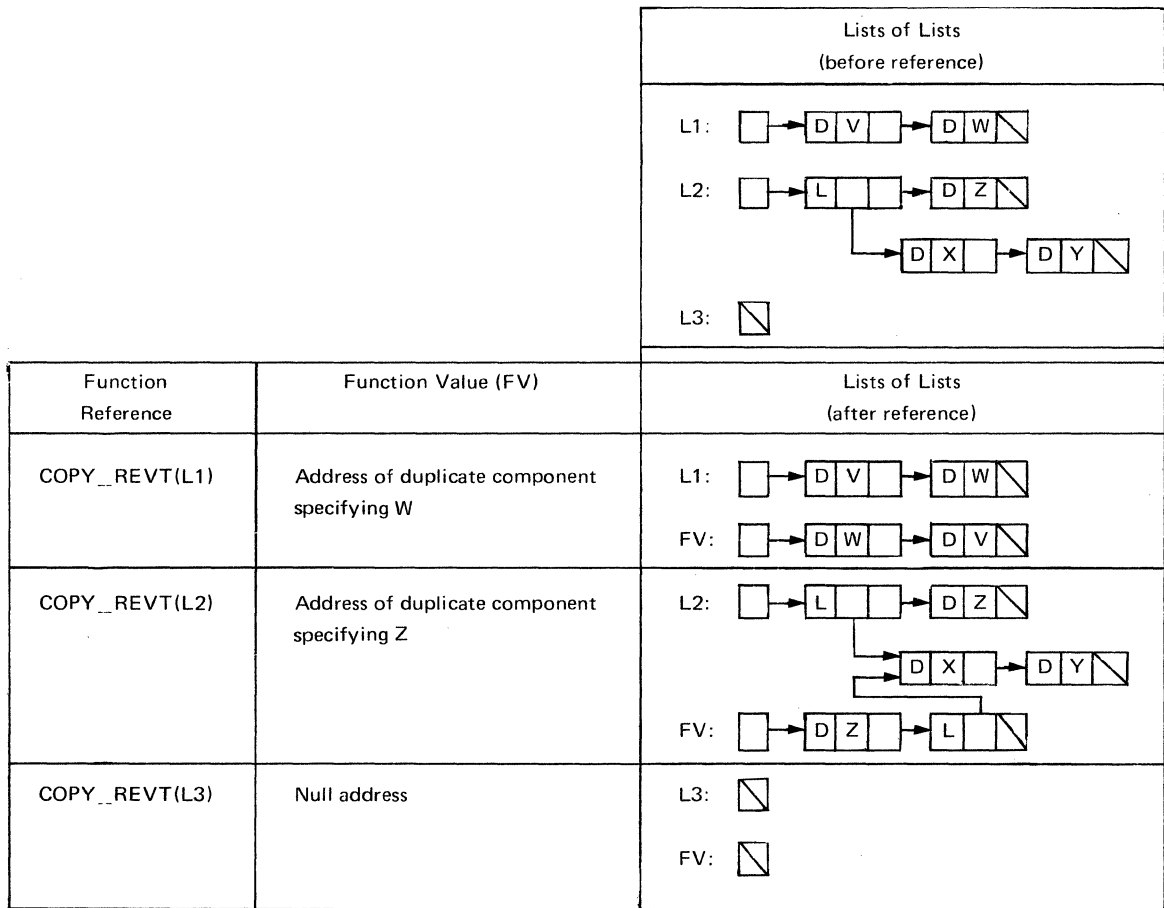


Figure 2.32C. Examples of references to the COPY_REVT function

COPY_REVT1 Function

Figures 2.33A and 2.33B present the COPY_REVT1 function, which uses recursive techniques to produce the same results as COPY_REVT in the previous discussion.

COPY_REVT1 Function	
Purpose	To copy the top level of a list of lists in reverse order and to return the address of the new list
Reference	COPY_REVT1(LIST,NULL)
Entry-Name Declaration	DECLARE COPY_REVT1 ENTRY(P POINTER, P POINTER) RETURNS(P POINTER);
Meaning of Argument	LIST — the list whose top level is to be copied in reverse order
Remarks	Components at lower levels of LIST are not copied but are shared with the new list. When LIST is null, a null address is returned.
Other Programmer-Defined Procedures Required	FORM_BODY, GET_VALUE, GET_TYPE, and GET_LINK
Method	COPY_REVT1 performs the recursive equivalent of the method developed in the previous function COPY_REVT. COPY_REVT1 uses two pointer parameters. The first parameter (LIST) represents the list whose top level is to be copied. The second parameter (ADDRESS) represents the new list; initially ADDRESS is null. COPY_REVT1 returns the value of the following recursive expression: COPY_REVT1(GET_LINK(LIST), FORM_BODY(GET_VALUE(LIST), GET_TYPE(LIST), ADDRESS)) Each level of recursion is terminated by a null link address.

Figure 2.33A. Description of the alternative function COPY_REVT1 for copying the top level of a list of lists in reverse order and returning the address of the new list

```
COPY_REVT1:PROCEDURE(LIST, ADDRESS)
  RETURNS(P POINTER) RECURSIVE;
  DECLARE (LIST, ADDRESS) POINTER;
  IF LIST = NULL THEN RETURN(ADDRESS);
  RETURN(COPY_REVT1(GET_LINK(LIST),
    FORM_BODY(GET_VALUE(LIST),
      GET_TYPE(LIST),ADDRESS)));
END COPY_REVT1;
```

Figure 2.33B. The COPY_REVT1 function

Manipulating all Levels of a List of Lists

So far, the procedures developed for processing items in a list of lists are restricted to individual items or to the items at the top level of a list. The following discussions develop subroutines and functions for manipulating all levels of a list of lists. These procedures are concerned with the following operations:

1. Obtaining the first and last data values in a list of lists
2. Counting the data values in a list of lists
3. Deleting values from a list of lists
4. Copying lists of lists
5. Testing lists of lists
6. Replacing data values in a list of lists

Obtaining First and Last Data Values in a List of Lists

To obtain either the first or the last data value in a list of lists, it may be necessary to search sublists at many levels. The following discussions develop four functions for performing such searches:

1. GET_FD, which gets the first, or leftmost, data value in a list of lists
2. GET_FDR, which is the recursive equivalent of GET_FD
3. GET_LD, which gets the last, or rightmost, data value in a list of lists
4. GET_LDR, which is the recursive equivalent of GET_LD

GET_FD Function

Figures 2.34A, 2.34B, and 2.34C present the GET_FD function, which requires the name of a list as its only argument. The function returns the address of the first data value in the list.

GET_FDR Function

Figures 2.35A and 2.35B present the GET_FDR function, which uses recursive techniques to produce the same result as GET_FD in the previous discussion.

GET_LD Function

Figures 2.36A, 2.36B, and 2.36C present the GET_LD function, which requires the name of a list as its only argu-

ment. The function returns the address of the last data value in the list.

GET_LDR Function

Figures 2.37A and 2.37B present the GET_LDR function, which uses recursive techniques to produce the same result as GET_LD in the previous discussion.

GET_FD Function	
Purpose	
To obtain the address of the first (leftmost) value associated with a data component in a list of lists	
Reference	
GET_FD(LIST)	
Entry-Name Declaration	
<pre> DECLARE GET_FD ENTRY(POINTER) RETURNS(POINTER); </pre>	
Meaning of Argument	
LIST	– the pointer variable that is the head of the list to be processed
Remarks	
When LIST is null, a null address is returned.	
Other Programmer-Defined Procedures Required	
GET_TYPE and GET_VALUE	
Method	
When the first component at the top level of LIST contains a data address value ('D'), this value is returned. When the first component contains a list address value ('L'), this sublist is searched. Searching continues in this manner until the first data address value to the left is encountered.	

Figure 2.34A. Description of the GET_FD function for getting the first data item in a list of lists

```

GET_FD:
  PROCEDURE (LIST)
  RETURNS (POINTER);
  DECLARE
    (LIST,ADDRESS) POINTER;
    ADDRESS = LIST;
  L:
    IF
      GET_TYPE(ADDRESS) = 'D'
    THEN
      RETURN(GET_VALUE(ADDRESS));
      ADDRESS = GET_VALUE(ADDRESS);
      GO TO
        L;
    END
  GET_FD:
  
```

Figure 2.34B. The GET_FD function

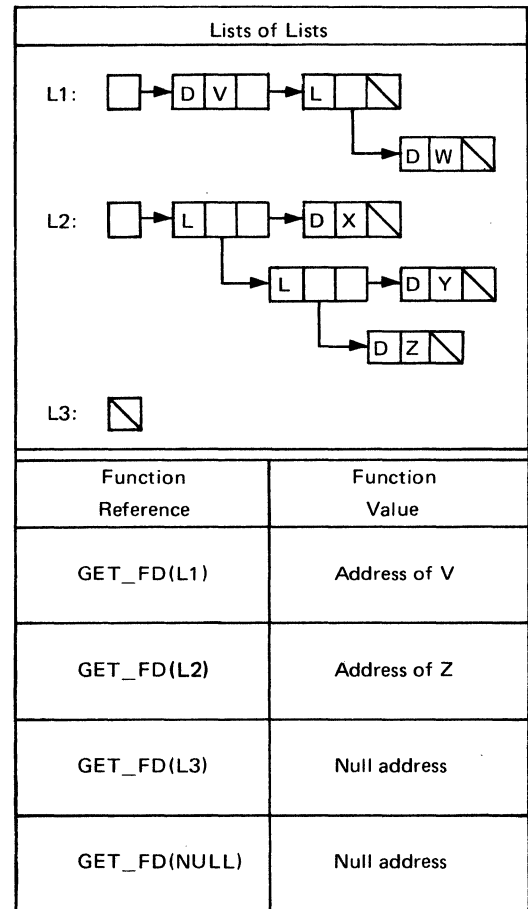


Figure 2.34C. Examples of references to the GET_FD function

GET_FDR Function	
Purpose	LIST – the pointer variable that is the head of the list to be processed
To obtain the address of the first (leftmost) value associated with a data component in a list of lists	Remarks
Reference	When LIST is null, a null address is returned.
GET_FDR(LIST)	Other Programmer-Defined Procedures Required
Entry-Name Declaration	GET_TYPE and GET_VALUE
DECLARE GET_FDR ENTRY(POINTER) RETURNS(POINTER);	Method
Meaning of Argument	GET_FDR performs the recursive equivalent of the previous function GET_FD.

Figure 2.35A. Description of the recursive function GET_FDR for getting the first data item in a list of lists

```

GET_FDR:
    PROCEDURE (LIST)
    RETURNS (POINTER) RECURSIVE;
    DECLARE
        LIST POINTER;
    IF
        GET_TYPE(LIST) = 'D'
    THEN
        RETURN(GET_VALUE(LIST));
        RETURN(GET_FDR(GET_VALUE(LIST)));
    END
    GET_FDR;

```

Figure 2.35B. The GET_FDR function

GET_LD Function	
Purpose	Remarks
To obtain the address of the last (rightmost) value associated with a data component in a list of lists	When LIST is null, a null address is returned.
Reference	Other Programmer-Defined Procedures Required
GET_LD(LIST)	ADDRESS_LVT, GET_TYPE, and GET_VALUE
Entry-Name Declaration	Method
DECLARE GET_LD ENTRY(POINTER) RETURNS(POINTER);	When the last component at the top level of LIST contains a data address value ('D'), this value is returned. When the last component contains a list address value ('L'), the sublist is searched. Searching continues in this manner until a sublist is encountered that contains a data address value in the last component of its top level.
Meaning of Argument	
LIST – the pointer variable that is the head of the list to be processed	

Figure 2.36A. Description of the GET_LD function for getting the last data item in a list of lists

```

GET_LD:
  PROCEDURE (LIST)
  RETURNS (POINTER);
  DECLARE
    (LIST, ADDRESS) POINTER;
    ADDRESS = LIST;
  L:
    ADDRESS = ADDRESS_LVT(ADDRESS);
  IF
    GET_TYPE(ADDRESS) = 'D'
  THEN
    RETURN(GET_VALUE(ADDRESS));
    ADDRESS = GET_VALUE(ADDRESS);
    GO TO
    L;
  END
  GET_LD;.

```

Figure 2.36B. The GET_LD function

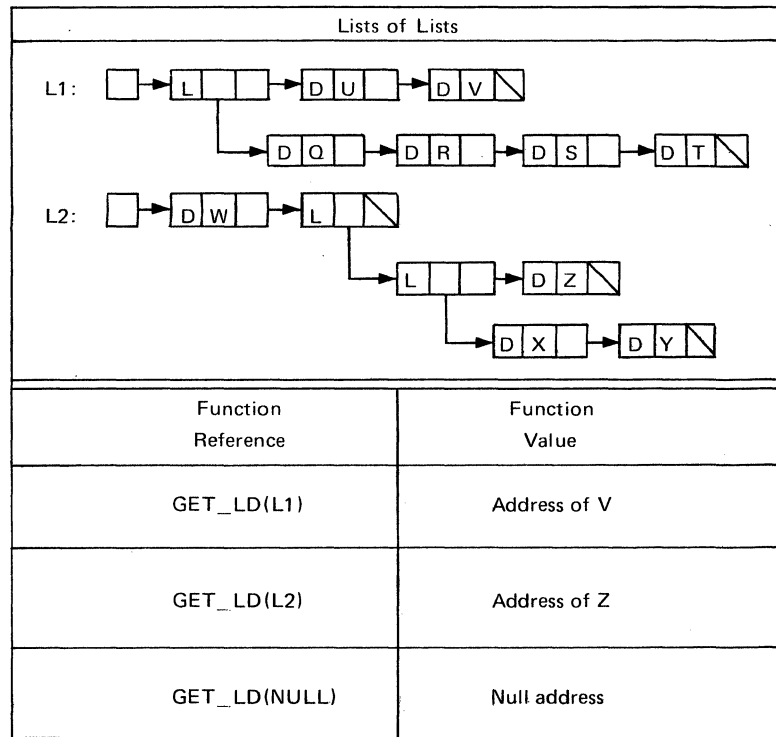


Figure 2.36C. Examples of references to the GET_LD function

GET_LDR Function	
Purpose	LIST — the pointer variable that is the head of the list to be processed
To obtain the address of the last (rightmost) value associated with a data component in a list of lists	Remarks
Reference	When LIST is null, a null address is returned.
GET_LDR(LIST)	Other Programmer-Defined Procedures Required
Entry-Name Declaration	ADDRESS_LVT, GET_TYPE, and GET_VALUE
DECLARE GET_LDR ENTRY(POINTER) RETURNS(POINTER);	Method
Meaning of Argument	GET_LDR performs the recursive equivalent of the previous function GET_LD.

Figure 2.37A. Description of the recursive function GET_LDR for getting the last data item in a list of lists

```

GET_LDR:
  PROCEDURE (LIST)
  RETURNS (POINTER) RECURSIVE;
  DECLARE
    (LIST, ADDRESS) POINTER;
    ADDRESS = ADDRESS_LVT(LIST);
  IF
    GET_TYPE(ADDRESS) = 'D'
  THEN
    RETURN(GET_VALUE(ADDRESS));
    RETURN(GET_LDR(GET_VALUE(ADDRESS)));
  END
  GET_LDR;

```

Figure 2.37B. The GET_LDR function

Counting Data Values in a List of Lists

The overall size of a list of lists is determined by the number of data values at all levels of the list. The following discussion develops the COUNT_D function for counting the number of data values throughout a list.

COUNT_D Function

Figures 2.38A, 2.38B, and 2.38C present the COUNT_D function, which requires the name of a list as its only argument. The function returns a count of all data values in the list.

COUNT_D Function	a zero size is returned. Only data items ('D') are counted.
Purpose	Other Programmer-Defined Procedures Required
To count all the data (D) values in a list of lists	GET_TYPE, GET_LINK, and GET_VALUE
Reference	Method
COUNT_D(LIST)	When the first component at the top level of LIST contains a data item ('D'), the following recursive expression is evaluated:
Entry-Name Declaration	1 + COUNT_D(GET_LINK(LIST))
DECLARE COUNT_D ENTRY(POINTER) RETURNS(FIXED DECIMAL(5));	When the first component contains a list item ('L'), the following recursive expression is evaluated:
Meaning of Argument	COUNT_D(GET_VALUE(LIST)) + COUNT_D (GET_LINK(LIST))
LIST — the pointer variable that is the head of the list to be processed	
Remarks	Each level of recursion is terminated when GET_LINK returns a null address.
The maximum size is 99999. When LIST is null,	

Figure 2.38A. Description of the COUNT_D function for counting all the data items in a list of lists

```

COUNT_D:
  PROCEDURE (LIST)
  RETURNS (FIXED DECIMAL(5))
  RECURSIVE;
  DECLARE
    LIST POINTER;
  IF
    LIST = NULL
  THEN
    RETURN(0);
  IF
    GET_TYPE(LIST) = 'D'
  THEN
    RETURN(1+COUNT_D (GET_LINK(LIST)));
    RETURN(COUNT_D (GET_VALUE(LIST)) +
    COUNT_D (GET_LINK(LIST)));
  END
  COUNT_D;

```

Figure 2.38B. The COUNT_D function

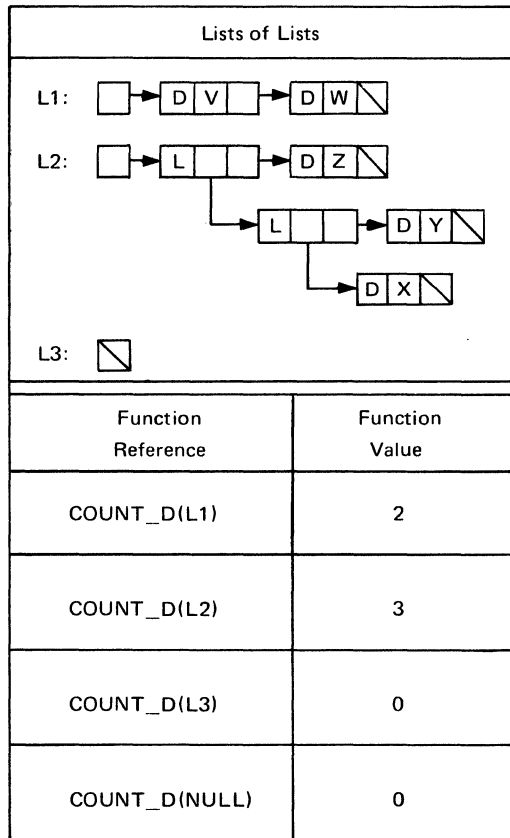


Figure 2.38C. Examples of references to the COUNT_D function

Deleting List Components from a List of Lists

The following discussions develop two subroutines for deleting list components from a list of lists:

1. DELETE_LIST, which deletes all components from a list of lists
2. DELETE_NVT, which deletes the nth component from the top level of a list of lists

DELETE_LIST Subroutine

Figures 2.39A, 2.39B, and 2.39C present the DELETE_LIST subroutine, which requires as its only argument the name of the list to be deleted.

DELETE_NVT Subroutine

Figures 2.40A, 2.40B, and 2.40C present the DELETE_NVT subroutine, which requires two arguments:

1. Name of a list of lists
2. Position of a value at the top level of the list

The subroutine deletes the value from the specified position at the top level of the list.

DELETE_LIST Subroutine

Purpose

To delete all values from a list of lists

Reference

DELETE_LIST(LIST)

Entry-Name Declaration

DECLARE DELETE_LIST ENTRY(POINTER);

Meaning of Argument

LIST – the list to be deleted

Remarks

LIST is null after deletion.

Other Programmer-Defined Procedures Required

GET_TYPE, GET_VALUE, GET_LINK, and SET_LINK

Method

DELETE_LIST is a recursive subroutine. When the value of the first component at the top level of the list is a data item ('D'), the component is inserted into the list of available storage components AVAIL, and the next component, which has now become the first component at the top level, is examined. When the value of the first component is a list item ('L'), the sublist (and any sublists within it) is deleted recursively. The component that contains the head of the deleted sublist is then inserted into the list of available storage components, and the next component at the top level is examined. These steps are repeated until the list becomes null.

Figure 2.39A. Description of the DELETE_LIST subroutine for deleting all values from a list of lists

```

DELETE_LIST:
  PROCEDURE(LIST) RECURSIVE;
  DECLARE
    (LIST,S,AVAIL EXTERNAL) POINTER;
  L:
  IF
    LIST = NULL
  THEN
    RETURN;
    /* IF THE VALUE OF THE FIRST
    COMPONENT IS A SUBLIST,
    RECURSIVELY DELETE THIS SUBLIST AND
    PROCEED TO THE FOLLOWING
    STATEMENTS, WHICH TREAT THE
    COMPONENT THAT CONTAINS THE HEAD OF
    THE DELETED LIST AS A DATA ('D')
    COMPONENT. */
  IF
    GET_TYPE(LIST) = 'L'
  THEN
    CALL DELETE_LIST (GET_VALUE(LIST));
    /* OTHERWISE, THE VALUE OF THE
    FIRST COMPONENT IS A DATA ITEM.
    THEREFORE, INSERT THE COMPONENT
    INTO THE LIST OF AVAILABLE STORAGE
    COMPONENTS. */
    S = AVAIL;
    AVAIL = LIST;
    LIST = GET_LINK(LIST);
    CALL SET_LINK(AVAIL,S);
    /* DELETE NEXT COMPONENT. */
  GO TO
  L;
END
  DELETE_LIST;

```

Figure 2.39B. The DELETE_LIST subroutine

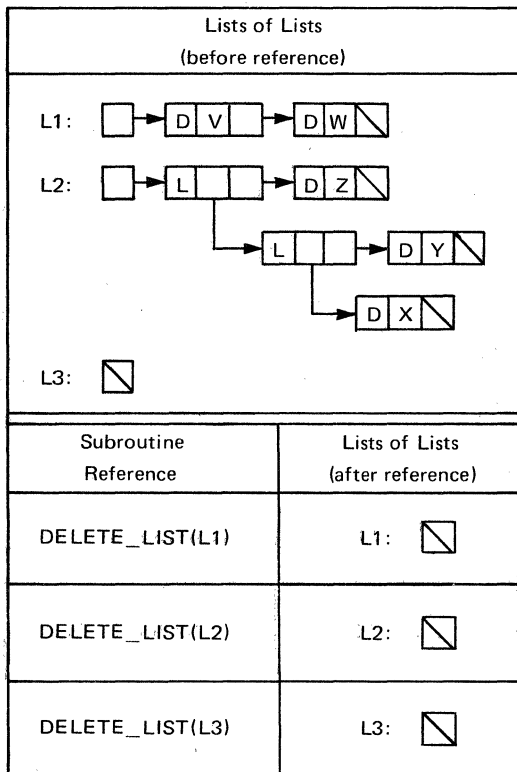


Figure 2.39C. Examples of references to the DELETE_LIST subroutine

DELETE_NVT Subroutine

Purpose
To delete the nth value from the top level of a list of lists

Reference
DELETE_NVT(LIST, N)

Entry-Name Declaration
DECLARE DELETE_NVT ENTRY(POINTER, FIXED DECIMAL(5));

Meaning of Arguments

LIST — the list from which the nth value at the top level is to be deleted

N — an integer that specifies the position of the value to be deleted from the top level of the list

Remarks
No value is deleted when N is less than one or greater than the size of the top level.

Other Programmer-Defined Procedures Required
ADDRESS_NVT, GET_TYPE, DELETE_LIST, GET_VALUE, GET_LINK, and SET_LINK

Method
When the nth component contains a data item ('D'), it is inserted into the list of available storage components AVAIL. When the nth component contains a list item ('L'), the sublist is deleted first, and then the nth component at the top level is inserted into AVAIL.

Figure 2.40A. Description of the DELETE_NVT subroutine for deleting the nth value from the top level of a list of lists

```

DELETE_NVT:
  PROCEDURE(LIST,N);
  DECLARE
    N FIXED DECIMAL(5),
    (LIST,COMPONENT,S,AVAIL EXTERNAL)
    POINTER;
    /* OBTAIN ADDRESS OF N-TH COMPONENT
    AT TOP OF LIST. */
    COMPONENT = ADDRESS_NVT(LIST,N);
    /* RETURN IF ADDRESS OF N-TH
    COMPONENT IS NULL. */
  IF
    COMPONENT = NULL
  THEN
    RETURN;
    /* IF THE VALUE OF THE N-TH
    COMPONENT IS A LIST, THEN DELETE
    THIS LIST. */
  IF
    GET_TYPE(COMPONENT) = 'L'
  THEN
    CALL DELETE_LIST(GET_VALUE
    (COMPONENT));
    /* OTHERWISE THE VALUE OF THE N-TH
    COMPONENT IS A DATA ITEM. THEREFORE,
    INSERT THE N-TH COMPONENT INTO THE
    LIST OF AVAILABLE STORAGE
    COMPONENTS. */
    S = AVAIL; AVAIL = COMPONENT;
  END
DELETE_NVT;

```

Figure 2.40B. The DELETE_NVT subroutine

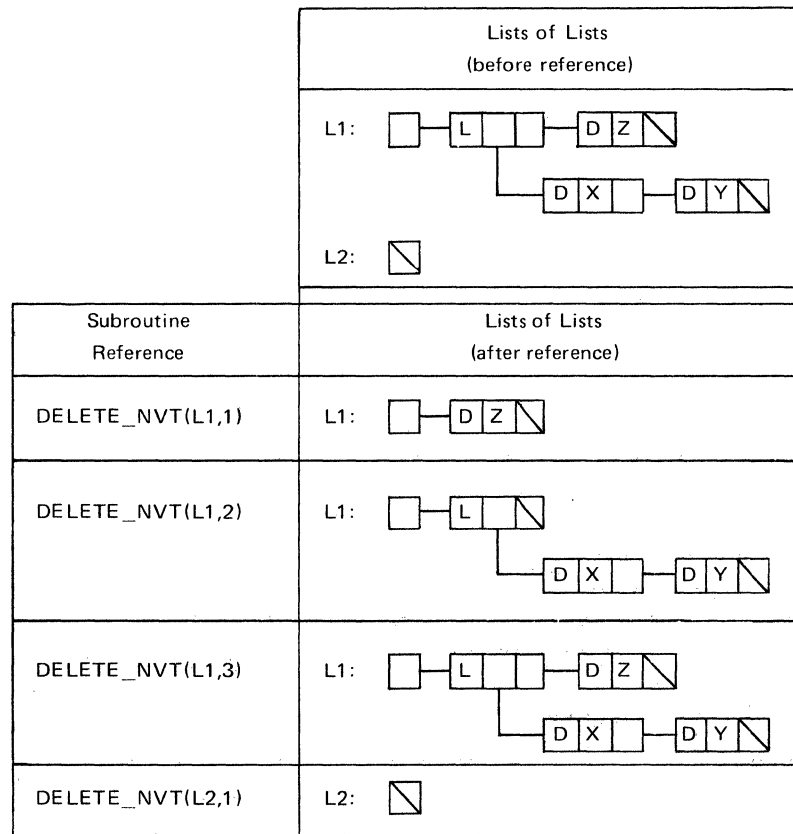


Figure 2.40C. Examples of references to the DELETE_NVT subroutine

Copying Lists of Lists

It is often necessary to create a distinct copy of a list of lists for working purposes, while retaining the original list. The second list can be an exact duplicate of the first, or it can contain modifications, such as having its sublists leveled, so that all its data values appear at the top level. The following discussions develop two functions for such duplication:

1. `COPY_LIST`, which copies a list without modifications
2. `COPY_LEV`, which copies a list with all sublists leveled

Each function returns the address of the new list. Assignment of this address to a pointer variable is effectively equivalent to list assignment.

COPY_LIST Function

Figures 2.41A, 2.41B, and 2.41C present the `COPY_LIST` function, which uses the name of a list as its only argument. The function returns the address of the duplicate list.

COPY_LEV Function

Figures 2.42A, 2.42B, and 2.42C present the `COPY_LEV` function, which uses as an argument the name of the list to be copied with all sublists leveled. The function returns the address of the new list.

COPY_LIST Function	Other Programmer-Defined Procedures Required
Purpose	
To copy a list of lists and to return the address of the new list	<code>FORM_BODY</code> , <code>GET_TYPE</code> , <code>GET_VALUE</code> , and <code>GET_LINK</code>
Reference	Method
<code>COPY_LIST(LIST)</code>	<code>COPY_LIST</code> is a recursive function procedure. When the first component at the top level of <code>LIST</code> specifies a data item ('D'), <code>COPY_LIST</code> returns the value of the following recursive expression:
Entry-Name Declaration	<code>FORM_BODY(GET_VALUE(LIST), 'D', COPY_LIST(GET_LINK(LIST)))</code>
<code>DECLARE COPY_LIST ENTRY(POINTER) RETURNS(POINTER);</code>	When the first component at the top level of <code>LIST</code> specifies a list item ('L'), <code>COPY_LIST</code> returns the value of the following recursive expression:
Meaning of Argument	<code>FORM_BODY(COPY_LIST(GET_VALUE(LIST)), 'L', COPY_LIST(GET_LINK(LIST)))</code>
<code>LIST</code> – the list to be copied	Recursion is terminated by a null link address.
Remarks	
Distinct components are created at all levels for the new list, so that no components are shared between the old list and the new list. When <code>LIST</code> is null, a null address is returned.	

Figure 2.41A. Description of the `COPY_LIST` function for copying a list of lists and returning the address of the new list

```

COPY_LIST:
  PROCEDURE (LIST)
    RETURNS (POINTER) RECURSIVE;
  DECLARE
    LIST POINTER;
  IF
    LIST = NULL
  THEN
    RETURN (NULL);
  IF
    GET_TYPE(LIST) = 'D'
  THEN
    RETURN(FORM_BODY(GET_VALUE(LIST),
                      'D',
                      COPY_LIST (GET_LINK(LIST))));
  ELSE
    RETURN(FORM_BODY(COPY_LIST
                     (GET_VALUE(LIST)),
                     'L',
                     COPY_LIST (GET_LINK(LIST))));
  END
  COPY_LIST;

```

Figure 2.41B. The `COPY_LIST` function

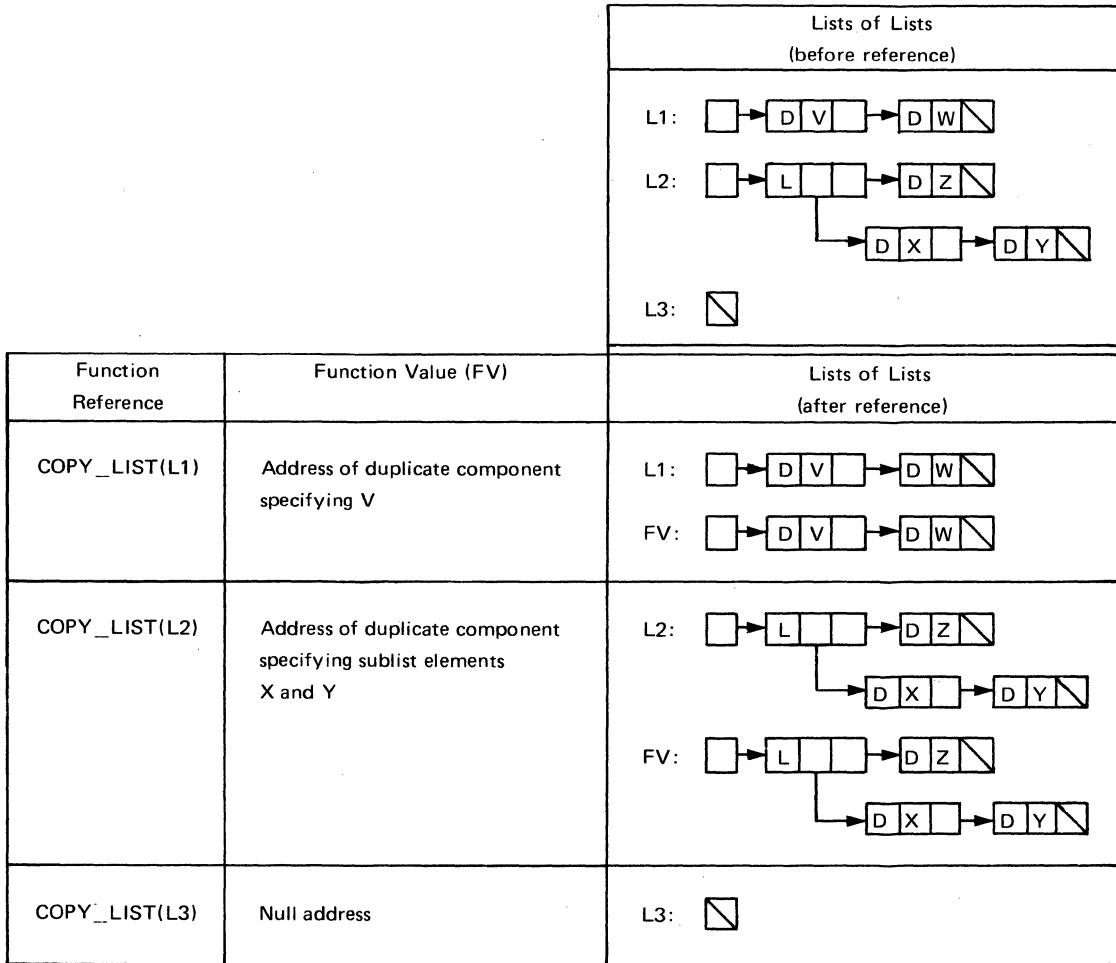


Figure 2.41C. Examples of references to the COPY_LIST function

COPY_LEV Function	Other Programmer-Defined Procedures Required
Purpose	FORM_BODY, GET_TYPE, GET_LINK, and GET_VALUE
To copy a list of lists with all sublists leveled and to return the address of the new list	Method
Reference	COPY_LEV is a recursive function procedure that uses two pointer parameters. The first parameter (LIST) represents the list to be copied. The second parameter (ADDRESS) represents the new list; initially, ADDRESS is null. When the first component at the top level of LIST is a data item ('D'), COPY_LEV returns the pointer value of the following recursive expression:
COPY_LEV(LIST,NULL)	
Entry-Name Declaration	
DECLARE COPY_LEV ENTRY (POINTER, POINTER) RETURNS (POINTER);	
Meaning of Argument	FORM_BODY(GET_VALUE(LIST), 'D', COPY_LEV(GET_LINK(LIST), ADDRESS))
LIST — the list to be copied	
Remarks	When the first component at the top level of LIST is a list item ('L'), COPY_LEV returns the value of the following recursive expression:
The new list contains only data items ('D'). Each sublist in LIST is replaced in the new list by a linked sequence of the data items contained in the sublist. When LIST is null, a null address is returned.	COPY_LEV(GET_VALUE(LIST), COPY_LEV(GET_LINK(LIST), ADDRESS))
	Each level of recursion is terminated by a null link address in LIST.

Figure 2.42A. Description of the COPY_LEV function for copying a list of lists with all sublists leveled and returning the address of the new list

```

COPY_LEV:
  PROCEDURE (LIST, ADDRESS)
  RETURNS (POINTER) RECURSIVE;
  DECLARE
    (LIST, ADDRESS) POINTER;
  IF
    LIST = NULL
  THEN
    RETURN (ADDRESS);
  IF
    GET_TYPE (LIST) = 'D'
  THEN
    RETURN (FORM_BODY (GET_VALUE (LIST),
      'D',
      COPY_LEV (GET_LINK (LIST), ADDRESS)));
    RETURN (COPY_LEV (GET_VALUE (LIST),
      COPY_LEV (GET_LINK (LIST), ADDRESS)));
  END
  COPY_LEV;

```

Figure 2.42B. The COPY_LEV function

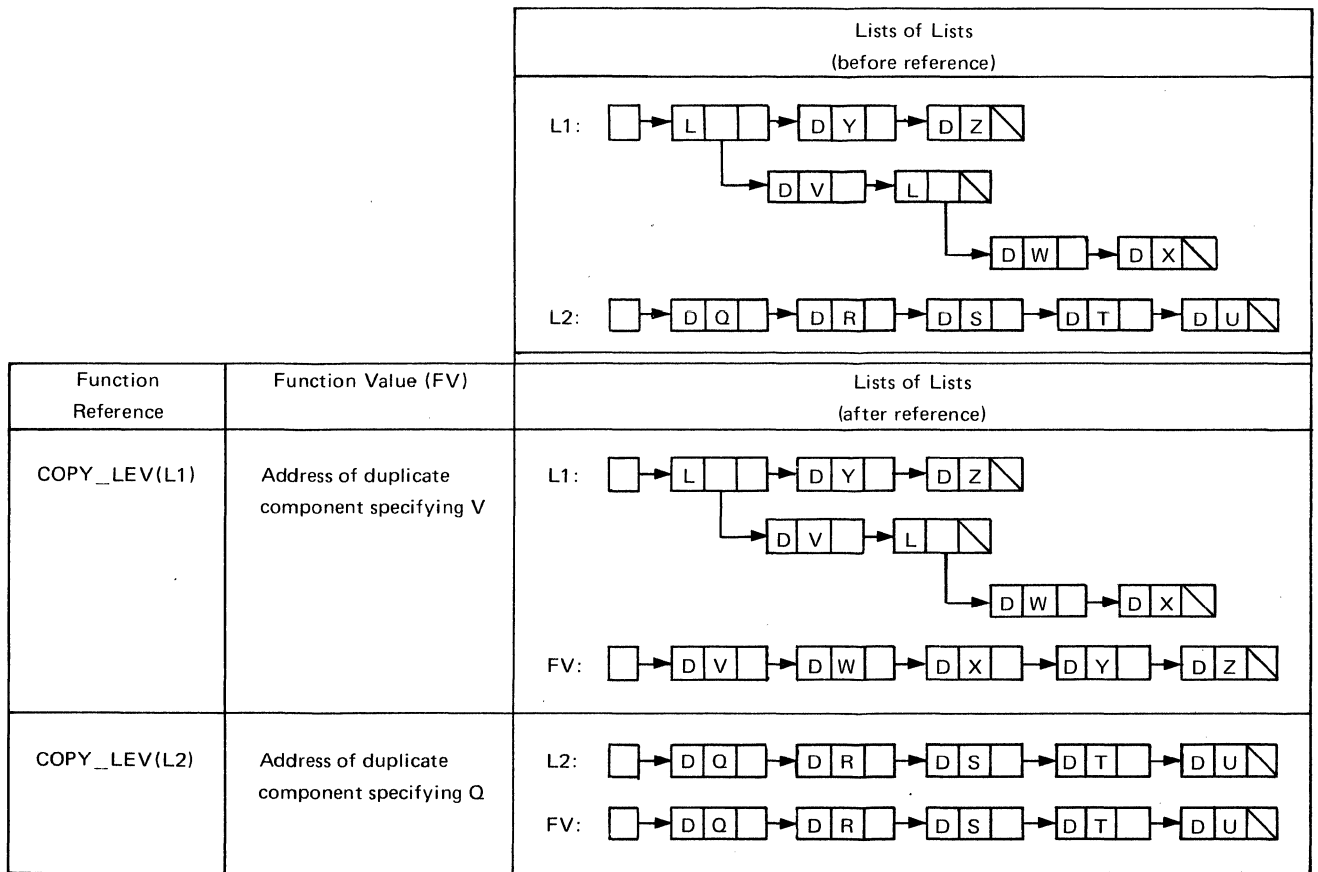


Figure 2.42C. Examples of references to the COPY_LEV function

Testing Lists of Lists

The following discussions develop two functions for performing tests on lists of lists:

1. EQUAL_L, which tests two lists for equality
2. MEMBER, which tests for the presence of a specified data value in a list of lists

EQUAL_L Function

Figures 2.43A, 2.43B, and 2.43C present the EQUAL_L function, which uses the names of two lists for its argu-

ments. When the two lists are equal, the function returns '1'B; otherwise, '0'B is returned.

MEMBER Function

Figures 2.44A, 2.44B, and 2.44C present the MEMBER function, which uses two arguments:

1. Address of a data item
2. Name of a list of lists

When the data item is a member of the list, the function returns '1'B; otherwise, it returns '0'B.

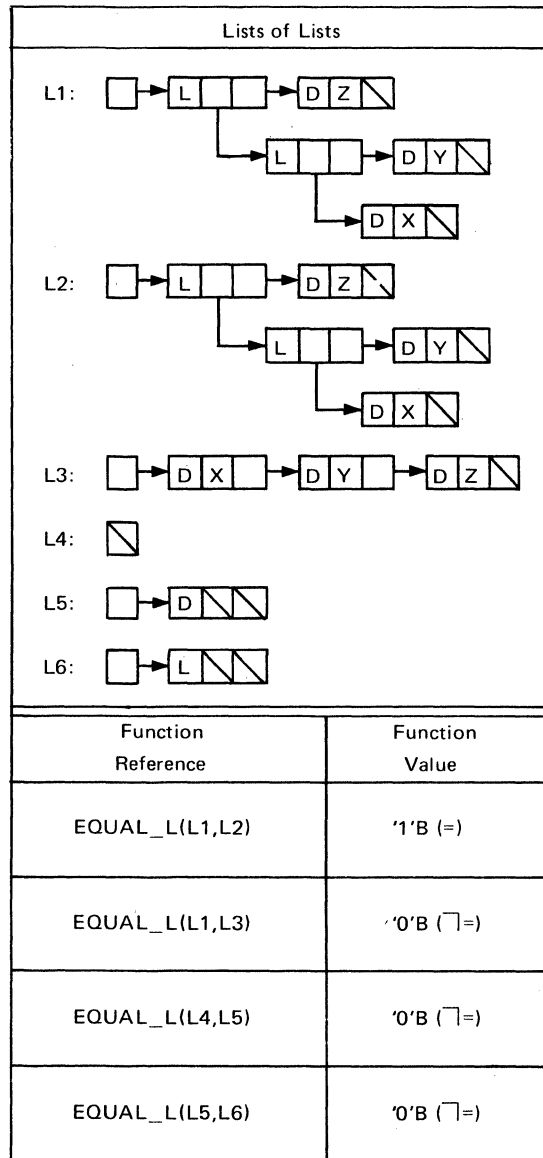


Figure 2.43C. Examples of references to the EQUAL_L function

MEMBER Function		GET_TYPE, GET_VALUE, and GET_LINK
Purpose	To test for the presence of a data item in a list of lists	Method
Reference	MEMBER(D, LIST)	MEMBER is a recursive function. When LIST is null, '0'B is returned. When the first component at the top level of LIST has type 'D' and its value equals D, MEMBER returns '1'B. When the first component at the top level of LIST has type 'D' but its value does not equal D, MEMBER executes the following statement:
Entry-Name Declaration	<pre> DECLARE MEMBER ENTRY(POINTER, POINTER) RETURNS(BIT(1)); </pre>	<pre> IF MEMBER(D, GET_LINK(LIST)) THEN RETURN('1'B); ELSE RETURN('0'B); </pre>
Meaning of Arguments	<p>D — the address of the data item being tested for</p> <p>LIST — the list being tested for the presence of D</p>	<p>When the first component at the top level of LIST has type 'L', MEMBER executes the following statements:</p> <pre> IF MEMBER(D, GET_VALUE(LIST)) THEN RETURN('1'B); </pre>
Remarks	When D is in LIST, MEMBER returns '1'B; otherwise, it returns '0'B.	<pre> IF MEMBER(D, GET_LINK(LIST)) THEN RETURN('1'B); ELSE RETURN('0'B); </pre>
Other Programmer-Defined Procedures Required		Each level of recursion is terminated by a null link address.

Figure 2.44A. Description of the MEMBER function for testing a list of lists for the presence of a data item

```

MEMBER:
  PROCEDURE (D, LIST)
    RETURNS (BIT(1)) RECURSIVE;
  DECLARE
    (D, LIST) POINTER;
  IF
    LIST = NULL
  THEN
    RETURN('0'B);
  IF
    GET_TYPE(LIST) = 'D'
  THEN
    IF
      GET_VALUE(LIST) = D
    THEN
      RETURN('1'B);
    ELSE;
  ELSE
    IF
      MEMBER(D, GET_VALUE(LIST))
    THEN
      RETURN('1'B);
    IF
      MEMBER(D, GET_LINK(LIST))
    THEN
      RETURN('1'B);
    ELSE
      RETURN('0'B);
  END
  MEMBER;

```

Figure 2.44B. The MEMBER function

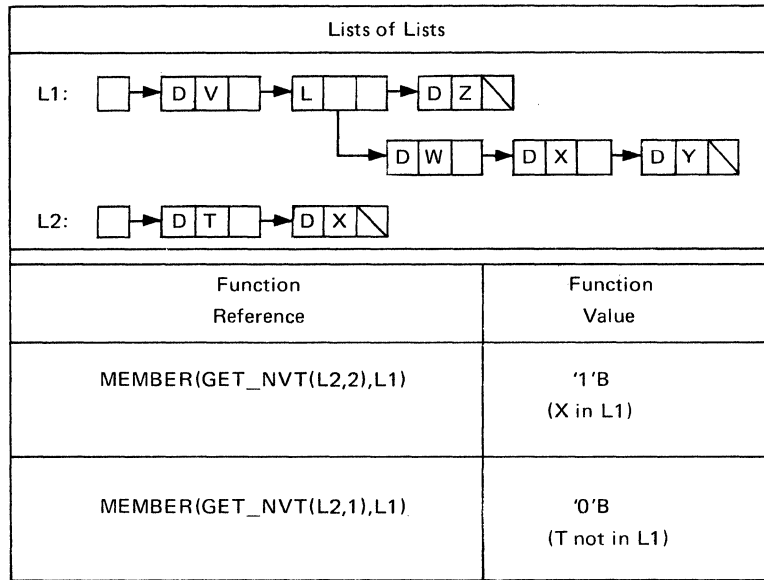


Figure 2.44C. Examples of references to the MEMBER function

Replacing Data Values in a List of Lists

The following discussion presents the REPLACE subroutine, which replaces each occurrence of a data value in a list of lists with another data value.

REPLACE Subroutine

Figures 2.45A, 2.45B, and 2.45C present the REPLACE subroutine, which uses three arguments:

1. Address of the data value being replaced
2. Address of the new data value
3. Name of the list of lists in which the replacement occurs

	Remarks
REPLACE Subroutine	
Purpose	D1 and D2 must be addresses of data items and not of lists (D1 or D2 can be null).
To replace each occurrence of a data item in a list of lists with another data item	Other Programmer-Defined Procedures Required
Reference	GET_TYPE, GET_VALUE, GET_LINK, and SET_VALUE
REPLACE(D1, D2, LIST)	Method:
Entry-Name Declaration	REPLACE is a recursive subroutine. When LIST is null, no replacement occurs.
DECLARE REPLACE ENTRY(POINTER, POINTER, POINTER);	When the first component at the top level of LIST has type 'D' and its value equals D1, value D2 replaces D1. When the first component at the top level of LIST has type 'L', the sublist is processed recursively with the following statement:
Meaning of Arguments	
D1 – the address of the data item to be replaced	CALL REPLACE(D1, D2, GET_VALUE(LIST));
D2 – the address of the data item replacing D1	In both cases, the remainder of the list is processed recursively with the following statement:
LIST – the list within which replacement occurs	CALL REPLACE(D1, D2, GET_LINK(LIST));

Figure 2.45A. Description of the REPLACE subroutine for replacing each occurrence of a data item in a list of lists with another data item

```

REPLACE:
PROCEDURE (D1,D2,LIST) RECURSIVE;
DECLARE
(D1,D2,LIST) POINTER;
IF
LIST = NULL
THEN
RETURN;
IF
GET_TYPE(LIST) = 'D'
THEN
IF
GET_VALUE(LIST) = D1
THEN
CALL SET_VALUE(LIST,D2,'D');
ELSE;
ELSE
CALL REPLACE (D1,D2,
GET_VALUE(LIST));
CALL REPLACE (D1,D2,GET_LINK(LIST));
END
REPLACE;

```

Figure 2.45B. The REPLACE subroutine

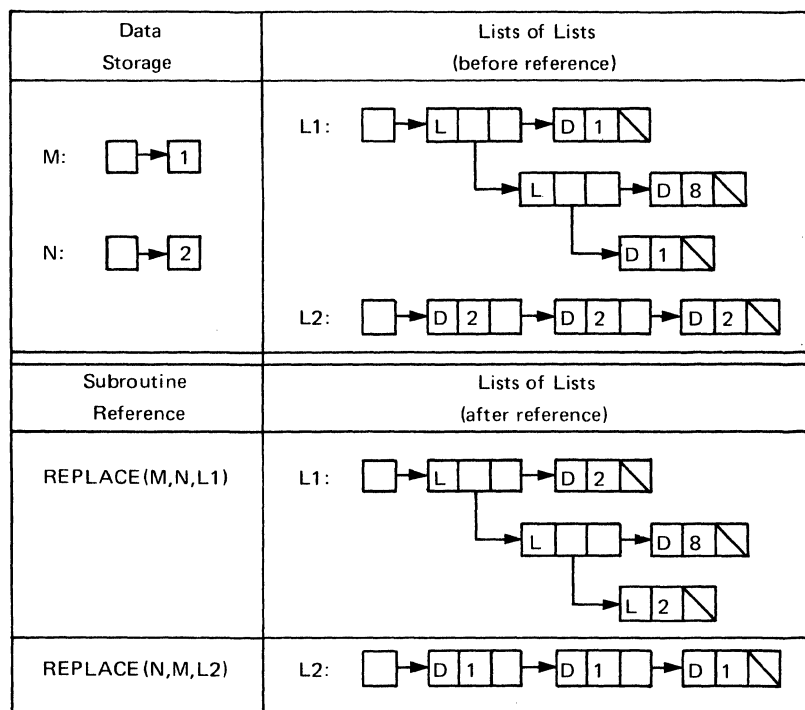


Figure 2.45C. Examples of references to the REPLACE subroutine

USING LISTS OF LISTS

Lists of lists provide the same reductions in data duplication and data movement that are available with pointer lists. Unlike pointer lists, however, lists of lists eliminate the need to know exactly how many lists a program will require during execution. New lists can be generated as the need arises for them during program execution, and they can be treated as sublists within a major list of lists.

The following discussions provide two examples of this flexibility in list generation.

The first example shows how a list of lists may be used to represent a binary tree that has an arbitrary number of branches. It then applies the tree list to a sort application.

The second example creates an index from descriptor words contained in a set of catalog cards. The index is organized as a list of lists. Each sublist specifies a descriptor word and the catalog cards that contain the descriptor. The number of descriptors, and consequently the number of sublists, is arbitrary.

Sorting With a Binary Tree

The following discussion shows how a list of lists may be used to represent a binary tree and how the resultant tree list may be applied to sorting.

Figure 2.46A applies a tree sort to the seven integers 4, 2, 6, 3, 1, 5, and 7, received in that order. Each node of the tree contains an integer. The left branch of each node always leads to a smaller integer, and the right branch always leads to a larger integer. Successive integers enter the tree at the bottom, as shown in Figure 2.46A.

The shape of the tree will vary according to the original order of the integers. If the integers are already in ascending sequence, the tree will contain right branches only. Similarly, a descending sequence will produce left branches only.

Although the placement of the integers in the binary tree of Figure 2.46A does not correspond to a conventional sort arrangement, the integers are readily retrieved in sort order. The smallest integer is reached by always taking the left branch of successive nodes. Taking the right branch of successive nodes leads to the largest integer. Appropriate combinations of left and right branches lead to the other integers. Later discussions develop recursive procedures for performing such retrieval.

Figure 2.46B contains a list representation of the binary tree constructed in Figure 2.46A. This list uses three list components for each node in the tree. The first component specifies the data value at the node. The second component is an L-component whose value pointer branches to the next node on the left. The third component is also an L-component whose value pointer branches to the next

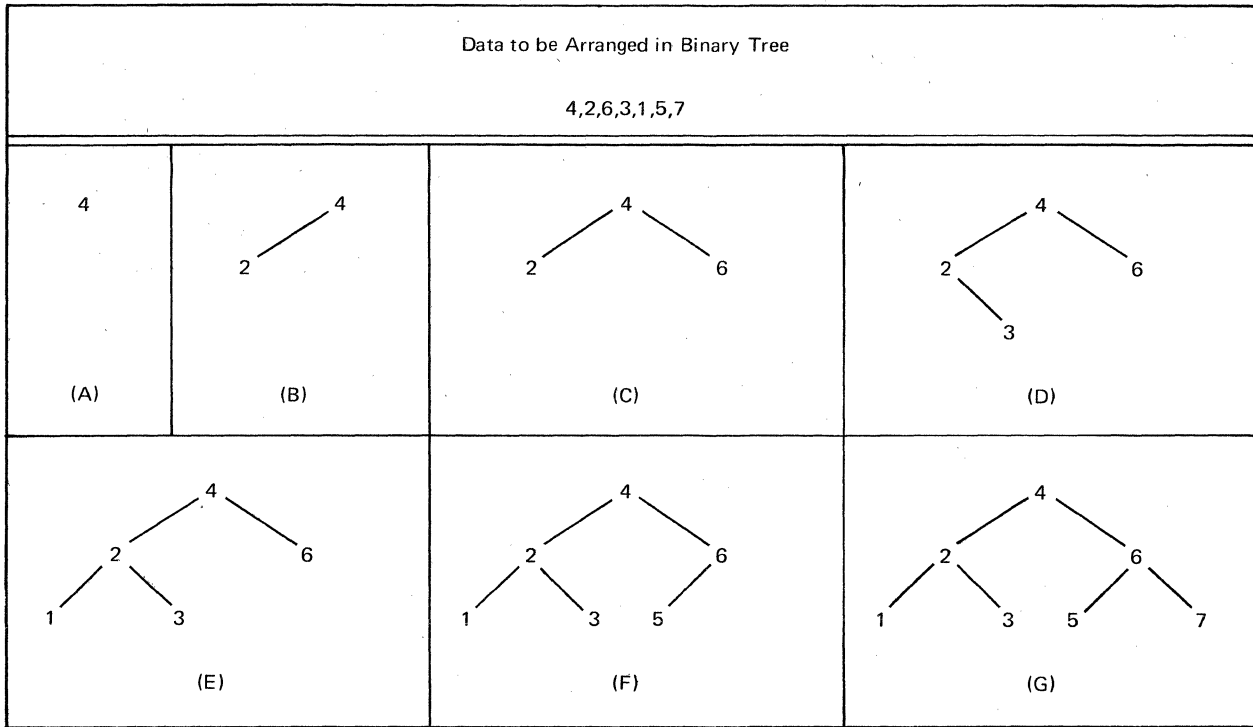


Figure 2.46A. Sorting data items by arranging them in a binary tree

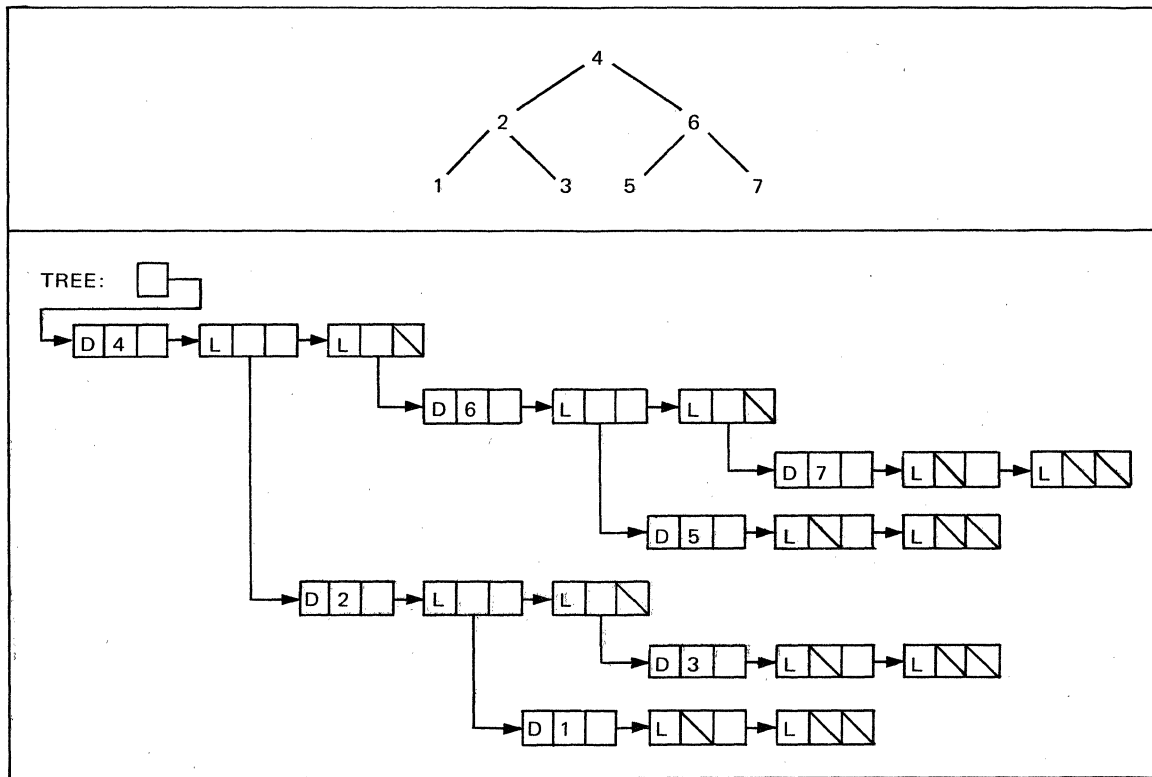


Figure 2.46B. Representing a binary tree with a list of lists

node on the right. When a node is not followed by another node (either on the left or on the right), the appropriate L-components contain null value pointers.

Representation of each node as a sublist within a list of lists allows the main list to contain an arbitrary number of nodes. It also frees the programmer from having to know exactly how many nodes will be required and what their arrangement will be during program execution.

The remainder of this discussion shows how a tree list (as illustrated in Figure 2.46B) may be used to sort successive sets of input cards, where each set contains an arbitrary number of cards. The cards are assumed to have the following structure:

```
1 CARD,
  2 KEY CHARACTER(3)
  2 DATA CHARACTER(77)
```

Sorting occurs in ascending sequence on KEY. Each set of cards is terminated by a card that contains three asterisks (***) in its KEY field.

To simplify the organization of the sort program, a main procedure (T_SORT) is designed to operate with a function procedure (NODE) and two subroutine procedures (ADD_NODE and T_PRINT):

1. NODE creates a three-component node for the list representation of a binary tree and returns the address of the leftmost component in the node.

2. ADD_NODE inserts a node (created by NODE) into the list representation of a binary tree in sort order.

3. T_PRINT prints in sort order the data values specified in the list representation of a binary tree.

Figures 2.46C and 2.46D present the NODE function, which uses the address of a card as its only argument. The function generates a three-component node for the card, assigns the address of the card to the value pointer of the leftmost component, and returns the address of the leftmost component. The value pointers of second and third components as well as the link pointer of the third component are null. The function creates the node by using three nested references to the function FORM_BODY, which was developed earlier.

```
NODE:
  PROCEDURE (CARD_ADDRESS)
  RETURNS (POINTER);
  DECLARE
    CARD_ADDRESS POINTER,
    NULL_PTR POINTER;
    NULL_PTR = NULL;
  RETURN(FORM_BODY(CARD_ADDRESS, 'D',
    FORM_BODY(NULL_PTR, 'L',
    FORM_BODY(NULL_PTR, 'L',
    NULL_PTR))));
  END
  NODE;
```

Figure 2.46C. The NODE function

To simplify the diagrams in Figure 2.46D, single-position character strings are used instead of 80-position cards. This simplification is also used in the remaining diagrams of the discussion.

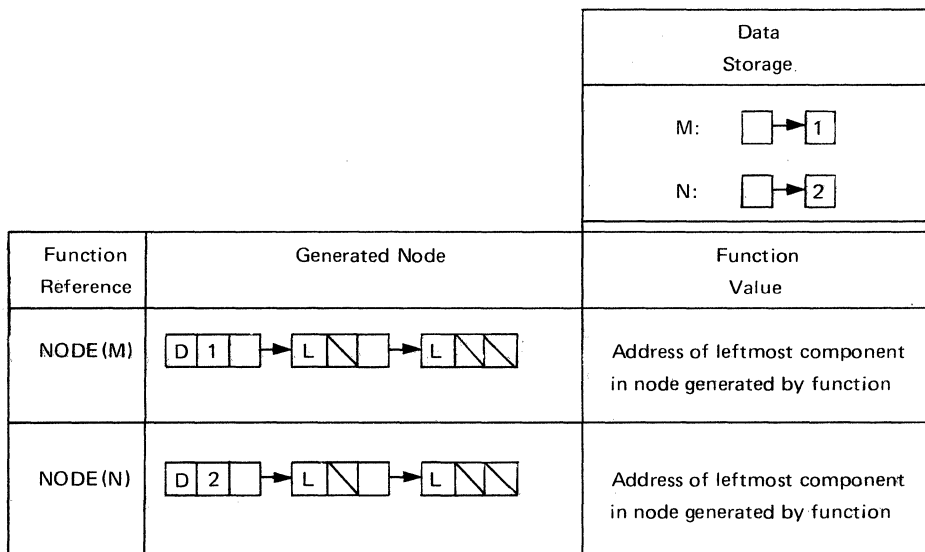


Figure 2.46D. Examples of references to the NODE function

Figures 2.46E and 2.46F present the ADD_NODE subroutine, which is a recursive procedure that uses two arguments:

1. Address of a card that contains KEY and DATA fields as described earlier
2. Name of a list that represents a binary tree

```

ADD_NODE:
  PROCEDURE (CARD_ADDRESS, TREE)
  RECURSIVE;
  DECLARE
    1 CARD_IMAGE BASED (NODE_CARD),
    2 KEY CHARACTER (3),
    2 DATA CHARACTER (77),
    NODE_CARD POINTER,
    (TREE, CARD_ADDRESS, LEFT, RIGHT)
    POINTER;
  IF
    TREE = NULL
  THEN
    DO;
      TREE = NODE (CARD_ADDRESS); RETURN;
    END;
    NODE_CARD = GET_VALUE (TREE);
    LEFT = GET_LINK (TREE);
    RIGHT = GET_LINK (GET_LINK (TREE));
  IF
    CARD_ADDRESS->KEY < NODE_CARD->KEY
  THEN
    IF
      GET_VALUE (LEFT) = NULL
    THEN
      CALL SET_VALUE (LEFT, NODE
        (CARD_ADDRESS), 'L');
    ELSE
      CALL ADD_NODE (CARD_ADDRESS,
        GET_VALUE (LEFT));
    ELSE
      IF
        GET_VALUE (RIGHT) = NULL
      THEN
        CALL SET_VALUE (RIGHT, NODE
          (CARD_ADDRESS), 'L');
      ELSE
        CALL ADD_NODE (CARD_ADDRESS,
          GET_VALUE (RIGHT));
    END
  END
  ADD_NODE;
  
```

Figure 2.46E. The ADD_NODE subroutine

The subroutine creates a node for the card by invoking the NODE function and then links the node to the bottom of the specified tree list. The KEY field of the card determines where the new node is inserted into the tree list.

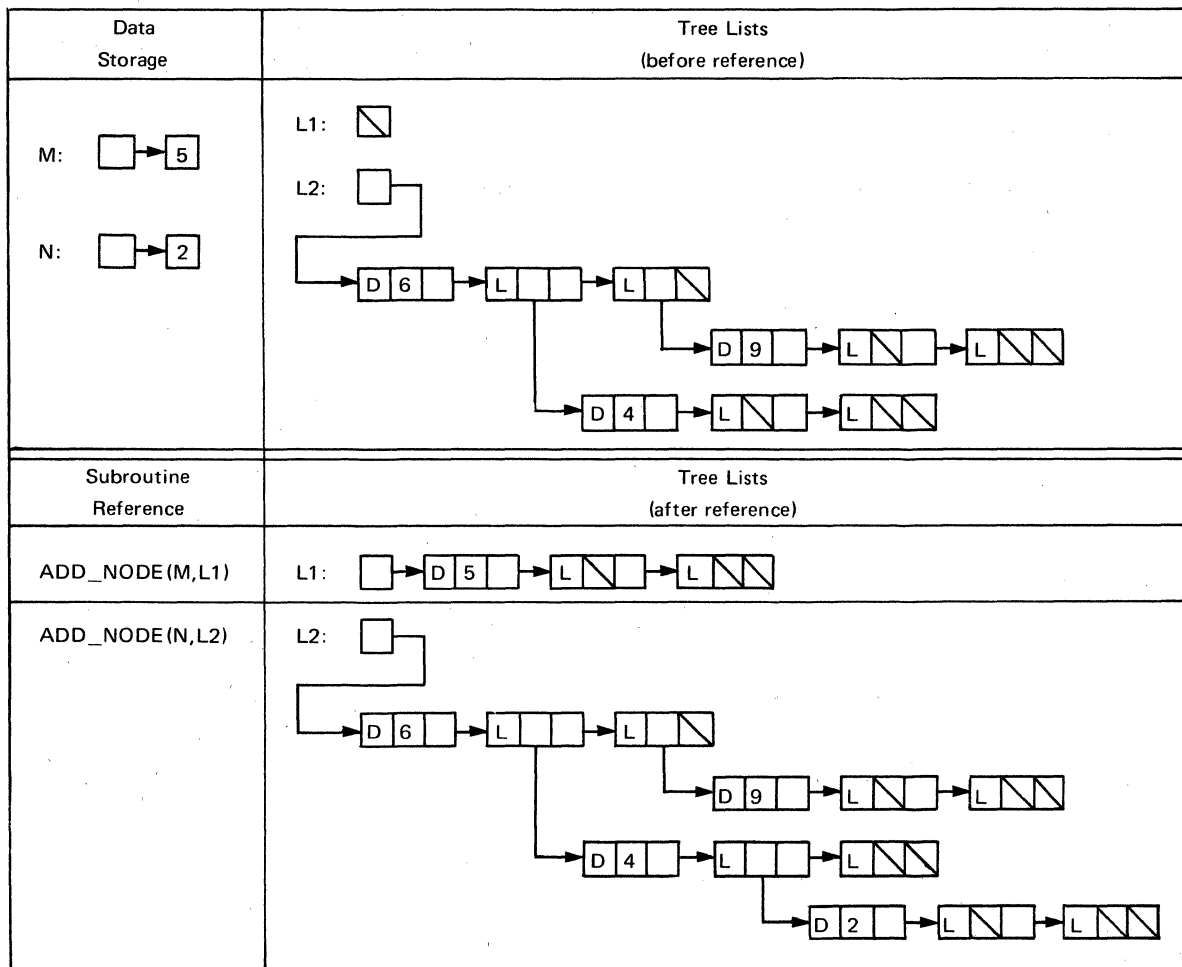


Figure 2.46F. Examples of references to the ADD_NODE subroutine

Figures 2.46G and 2.46H present the T_PRINT subroutine, which is a recursive procedure that uses the name of a tree list as its only argument. The subroutine prints the cards specified at the nodes of the tree list. The cards are printed in sort order on the standard system-output file, SYSPRINT.

```

T_PRINT:
  PROCEDURE(TREE) RECURSIVE;
  DECLARE
    1 CARD_IMAGE BASED(NODE_CARD),
    2 KEY CHARACTER(3),
    2 DATA CHARACTER(77),
    NODE_CARD POINTER,
    (TREE,LEFT,RIGHT) POINTER;
  IF
    TREE = NULL
  THEN
    RETURN;
    NODE_CARD = GET_VALUE(TREE);
    LEFT = GET_VALUE(GET_LINK(TREE));
    RIGHT = GET_VALUE(GET_LINK
      (GET_LINK(TREE)));
    CALL T_PRINT(LEFT);
  PUT
    EDIT(NODE_CARD->CARD_IMAGE)(A);
  PUT
    SKIP;
    CALL T_PRINT(RIGHT);
  END
    T_PRINT;

```

Figure 2.46G. The T_PRINT subroutine

Actual sorting is under control of the main procedure T_SORT, which appears in Figure 2.46I. T_SORT invokes the AREA_OPEN subroutine (developed earlier) to create a list of available storage components (AVAIL) in the

storage area called LIST_AREA. Cards are read from the standard system-input file, SYSIN, and printed unsorted on the standard system-output file, SYSPRINT. As each card is read, based storage is allocated for it in the area called CARD_AREA. Reading stops when the KEY field of a card contains three asterisks (***). Should the number of cards in a set exceed the capacity of CARD_AREA, the remaining cards in the set are skipped, and only those cards allocated in CARD_AREA are sorted.

```

T_SORT:
  PROCEDURE;
  DECLARE
    1 CARD,
    2 KEY CHARACTER(3),
    2 DATA CHARACTER(77),
    1 CARD_IMAGE BASED(CARD_ADDRESS),
    2 KEY CHARACTER(3),
    2 DATA CHARACTER(77),
    (CARD_AREA, LIST_AREA) AREA,
    CARD_ADDRESS POINTER,
    (TREE,AVAIL EXTERNAL) POINTER;
    /* WHEN ALL SETS OF INPUT CARDS
    HAVE BEEN SORTED, TERMINATE
    PROGRAM. */
  ON ENDFILE (SYSIN)
  GO TO
    END_T_SORT;
    /* WHEN THE NUMBER OF CARDS IN A SET
    EXCEEDS THE CAPACITY OF CARD_AREA,
    SKIP REMAINING CARDS IN SET, AND
    PRINT THE FOLLOWING MESSAGE AFTER
    LAST CARD: '*** INPUT EXCEEDED AREA
    CAPACITY'. THEN PRINT CONTENT OF
    TREE LIST IN SORT ORDER. */
  ON AREA
  BEGIN;
  SKIP:
  GET
    EDIT (CARD) (A(3), A(77));
  IF
    CARD.KEY = '***'

```

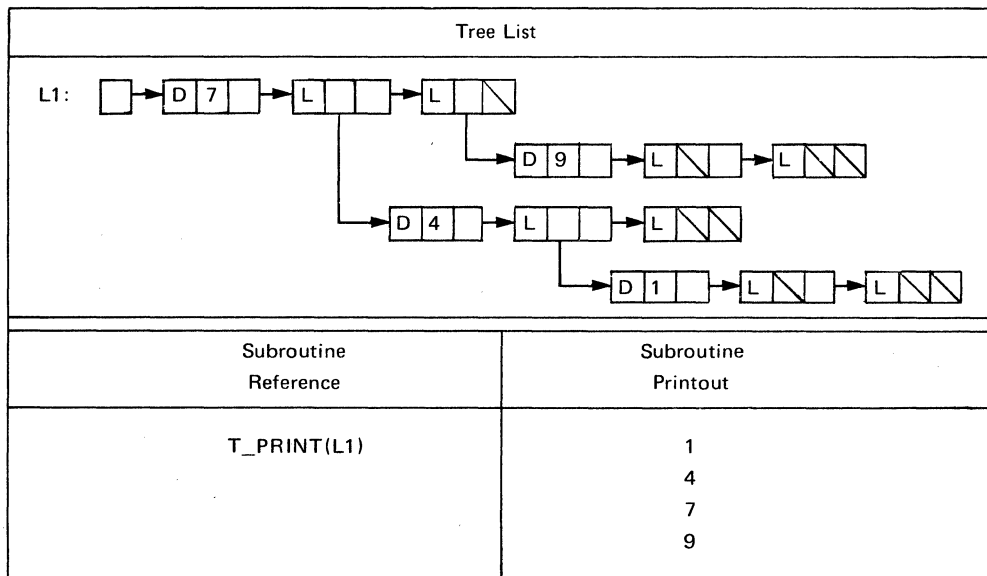


Figure 2.46H. Example of a reference to the T_PRINT subroutine

```

THEN
DO;
  PUT
    LIST('*** INPUT EXCEEDED AREA
    CAPACITY');
  PUT
    SKIP;
  GO TO
    PRINT_OUTPUT;
END;
ELSE
  GO TO
    SKIP;
END;

/* INITIALIZE. */
TREE, AVAIL = NULL;
LIST_AREA = EMPTY;
CALL AREA_OPEN(LIST_AREA,AVAIL);
/* GET NEXT SET OF INPUT CARDS.
PRINT EACH CARD AS IT IS READ, AND
CREATE A NODE FOR IT IN TREE LIST. */

START:
  GET
    EDIT(CARD)(A(3),A(77));
  PUT
    PAGE
    LIST('T_SORT INPUT:');
  PUT
    SKIP;
PRINT_INPUT:
  PUT
    EDIT(CARD)(A);
  PUT
    SKIP;
  IF
    CARD.KEY = '****'
  THEN
    GO TO
      PRINT_OUTPUT;
    ALLOCATE CARD_IMAGE IN(CARD_AREA)
    SET(CARD_ADDRESS);
    CARD_ADDRESS->CARD_IMAGE = CARD;
    CALL ADD_NODE(CARD_ADDRESS,TREE);
  GET
    EDIT(CARD)(A(3),A(77));
  GO TO
    PRINT_INPUT;
/* PRINT TREE LIST IN SORT ORDER, */
PRINT_OUTPUT:
  PUT
    PAGE
    LIST('T_SORT OUTPUT:');
  PUT
    SKIP;
  CALL T_PRINT(TREE);
  PUT
    EDIT(CARD)(A);
/* CLEAR CARD_AREA AND TREE LIST,*/
/*THEN PROCESS NEXT SET OF INPUT */
/* CARDS. */
  CARD_AREA = EMPTY;
  CALL DELETE_LIST(TREE);
  GO TO
    START;
END_T_SORT:
END
  T_SORT;

```

Figure 2.46I. The T_SORT procedure

The address of each card in CARD_AREA is used to form a node in the tree list called TREE. The ADD_NODE subroutine inserts the node into TREE, as described earlier. When TREE contains a node for each card, the cards are printed in sort order by the T_PRINT subroutine. A sample printout appears in Figure 2.46J.

```

T_SORT OUTPUT:
1 ONE
2 TWO
3 THREE
4 FOUR
5 FIVE
6 SIX
7 SEVEN
***

```

Figure 2.46J. Printout from T_SORT and T_PRINT

Before the next set of input cards is sorted, CARD_AREA and TREE are cleared. Processing is terminated when an end-of-file condition occurs on the standard system-input file, SYSIN.

In summary, a tree list permits varying numbers of input cards to be sorted with a minimum of data movement and data duplication.

Indexing Catalog Cards

The use of a list of lists to index catalog cards is illustrated in Figures 2.47A through 2.47H. To avoid complexity, the discussion uses a simplified version of a catalog card, as shown in Figure 2.47A. The first ten columns of each card contain an accession number, which consists of any combination of characters acceptable to the computer. In the case of a book catalog, the accession number might correspond to a Dewey decimal number; or it might serve as a part number for a catalog of machine parts. It could also represent the identification number used in an art collection.

INPUT TO INDEX:

```

T2-XY4-16 PHOTOGRAPH BLACK WHITE LARGE
A9-1L7-RZ PAINTING BLACK WHITE SMALL
Y9-016-X8 SCULPTURE MARBLE BLACK SMALL
Y8-123-X7 SCULPTURE MARBLE WHITE MEDIUM

```

Figure 2.47A. Input to INDEX

The remaining 70 columns of each card contain descriptive information about the item being cataloged. This information consists of descriptive words (descriptors) that specify particular features of the item being cataloged. For this discussion, a descriptor cannot exceed ten characters in length, but it can contain any combination of characters and need not be restricted to a word. The number of

descriptors in a card is arbitrary, but at least one blank character must separate successive descriptors. When a card cannot hold all the desired descriptors, additional cards may be used, provided that they contain the same accession number.

The four catalog cards in Figure 2.47A provide information about art objects, and the index produced for these cards appears in Figure 2.47H. The descriptors are printed in sort order, and each descriptor is followed by all cards that contain the descriptor. The cards for each descriptor are arranged in ascending sequence on accession number.

A possible list representation for this type of index appears in Figure 2.47B. The list contains an arbitrary number of sublists, each of which is associated with a separate descriptor. The value pointer for the first component in a sublist specifies the descriptor for that sublist. The value pointer for each of the remaining components in a sublist specifies a card that contains the descriptor for the sublist.

To simplify the organization of the program that creates the index list and prints it, a main procedure (INDEX) is designed to operate with two function procedures (GET_DESCRIPTOR and GET_DESCRIPTOR_COMPONENT) and two subroutine procedures (INSERT_CARD and PRINT_INDEX):

1. GET_DESCRIPTOR obtains the next descriptor in a catalog card.
2. GET_DESCRIPTOR_COMPONENT obtains the address of the first component in a sublist that is associated with a specified descriptor. When no sublist exists for the descriptor, a sublist is created, and the address of its first component is returned.
3. INSERT_CARD inserts a catalog card in a specified sublist.
4. PRINT_INDEX prints the index list in sort order.

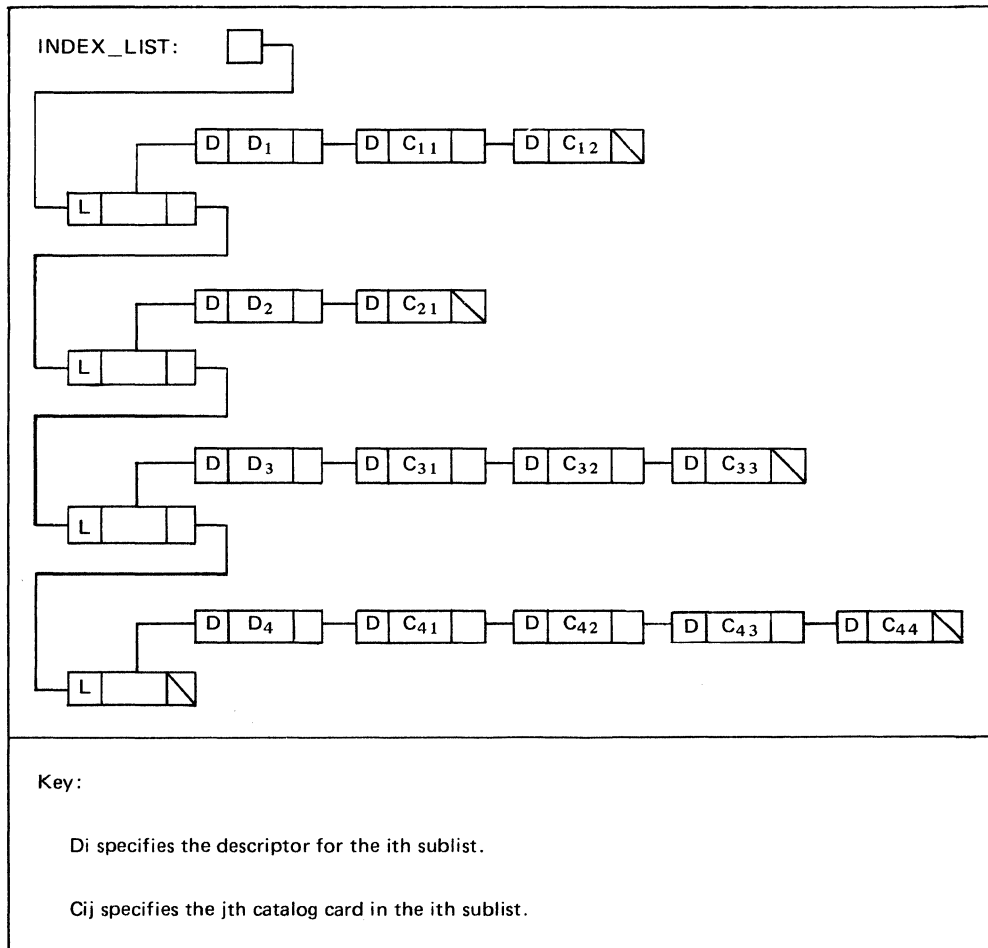


Figure 2.47B. Representing an index with a list of lists

Figure 2.47C contains the GET_DESCRIPTOR function procedure. It uses one parameter (DESCRIPTOR_STRING), which is a varying-length character string that has a maximum length of 71 characters. The value of the parameter initially consists of the characters in columns 11 through 80 of a catalog card and an additional blank character appended on the right.

```

GET_DESCRIPTOR:
  PROCEDURE (DESCRIPTOR_STRING)
  RETURNS (CHARACTER (10));
  DECLARE
    DESCRIPTOR_STRING CHARACTER(71)
    VARYING,
    DESCRIPTOR CHARACTER(10),
    (C1,C2) FIXED DECIMAL(2);
  FIRST_CHARACTER:
    DO
      C1 = 1 TO LENGTH(DESCRIPTOR_STRING);
      IF
        SUBSTR(DESCRIPTOR_STRING, C1,1) = ' '
      THEN
        GO TO
          LAST_CHARACTER;
    END;
  RETURN_BLANK_DESCRIPTOR:
    RETURN((10) ' ');
  LAST_CHARACTER:
    DO
      C2 = C1 TO LENGTH
        (DESCRIPTOR_STRING);
      IF
        SUBSTR(DESCRIPTOR_STRING,C2,1) = ' '
      THEN
        GO TO
          EXTRACT_DESCRIPTOR;
    END;
  EXTRACT_DESCRIPTOR:
    DESCRIPTOR = SUBSTR
      (DESCRIPTOR_STRING,C1,C2-1);
  SHIFT_DESCRIPTOR_STRING:
    DESCRIPTOR_STRING =
      SUBSTR(DESCRIPTOR_STRING, C2);
  RETURN_DESCRIPTOR:
    RETURN(DESCRIPTOR);
  END
  GET_DESCRIPTOR;

```

Figure 2.47C. The GET_DESCRIPTOR function

Each invocation of GET_DESCRIPTOR obtains the first descriptor in parameter DESCRIPTOR_STRING and returns it to the point of invocation. The first descriptor is also deleted from DESCRIPTOR_STRING, and if the descriptor contains more than ten characters, the leftmost ten are returned. When no descriptors remain in DESCRIPTOR_STRING, an invocation of GET_DESCRIPTOR produces a string of ten blank characters.

The function procedure GET_DESCRIPTOR_COMPONENT appears in Figure 2.47D. The function uses two parameters: DESCRIPTOR, which specifies a descriptor word, and INDEX_LIST, which is the pointer head of a list of lists that has the organization shown in Figure 2.47B.

```

GET_DESCRIPTOR_COMPONENT:
  PROCEDURE (DESCRIPTOR,INDEX_LIST)
  RETURNS (POINTER);
  DECLARE
    N FIXED DECIMAL(5),
    DESCRIPTOR CHARACTER(10),
    DESCRIPTOR_IMAGE BASED
      (DESCRIPTOR_ADDRESS) CHARACTER(10),
    (INDEX_LIST, SUBLIST) POINTER,
    DATA_AREA AREA EXTERNAL;
    N = 1;
    SUBLIST = INDEX_LIST;
  TEST_DESCRIPTOR:
    IF
      SUBLIST = NULL
    THEN
      GO TO
        INSERT_DESCRIPTOR;
      DESCRIPTOR_ADDRESS = GET_VALUE
        (GET_VALUE(SUBLIST));
    IF
      DESCRIPTOR_ADDRESS->DESCRIPTOR_IMAGE
      < DESCRIPTOR
    THEN
      DO;
      N = N + 1;
      SUBLIST = GET_LINK(SUBLIST);
      GO TO
        TEST_DESCRIPTOR;
    END;
    IF
      DESCRIPTOR_ADDRESS->DESCRIPTOR_IMAGE
      = DESCRIPTOR
    THEN
      RETURN(GET_VALUE(SUBLIST));
  INSERT_DESCRIPTOR:
    ALLOCATE DESCRIPTOR_IMAGE IN
      (DATA_AREA) SET (DESCRIPTOR_ADDRESS);
    DESCRIPTOR_ADDRESS->DESCRIPTOR_IMAGE
      = DESCRIPTOR;
    CALL INSERT_NVT(INDEX_LIST,N,
      FORM_BODY(DESCRIPTOR_ADDRESS,'D',
        NULL), 'L');
    RETURN(GET_VALUE(ADDRESS_NVT
      (INDEX_LIST,N)));
  END
  GET_DESCRIPTOR_COMPONENT;

```

Figure 2.47D. The GET_DESCRIPTOR_COMPONENT function

GET_DESCRIPTOR_COMPONENT searches INDEX_LIST for a sublist whose first component specifies the same descriptor as parameter DESCRIPTOR. When the sublist is found, the address of its first component is returned. If no sublist exists for the descriptor, the function creates a sublist and inserts it into INDEX_LIST in ascending sequence on the descriptor. The function then returns the address of the first component in this new sublist.

Note that DESCRIPTOR is a ten-position character string whose descriptor value is adjusted to the left and extended, if necessary, with blanks on the right.

Figure 2.47E contains the subroutine procedure INSERT_CARD. The subroutine uses two parameters: CATALOG_CARD_ADDRESS, which represents the

storage address of a catalog card, and DESCRIPTOR_LIST, which specifies the address of the first list component in a sublist within the index list (INDEX_LIST).

```

INSERT_CARD:
    PROCEDURE(CATALOG_CARD_ADDRESS,
              DESCRIPTOR_LIST);
DECLARE
    N FIXED DECIMAL(5),
    CATALOG_CARD_ADDRESS
    POINTER,
    DESCRIPTOR_LIST POINTER,
    CARD_COMPONENT POINTER,
    1 CARD_IMAGE BASED(IMAGE_ADDR),
    2 ACCESSION#_IMAGE CHARACTER(10),
    2 DESCRIPTOR_GROUP_IMAGE
    CHARACTER(70);
INITIALIZE:
    N = 1;
    CARD_COMPONENT =
    DESCRIPTOR_LIST;
NEXT_CARD:
    N = N + 1;
    CARD_COMPONENT = GET_LINK
    (CARD_COMPONENT);
    IF
        CARD_COMPONENT = NULL
    THEN
        GO TO
        INSERT;
        IMAGE_ADDR = GET_VALUE
        (CARD_COMPONENT);
    IF
        (IMAGE_ADDR->ACCESSION#_IMAGE
        < CATALOG_CARD_ADDRESS->
        ACCESSION#_IMAGE)
    THEN
        GO TO
        NEXT_CARD;
    IF
        (IMAGE_ADDR->ACCESSION#_IMAGE
        = CATALOG_CARD_ADDRESS->
        ACCESSION#_IMAGE)
    THEN
        RETURN;
INSERT:
    CALL INSERT_NVT(DESCRIPTOR_LIST,
    N,CATALOG_CARD_ADDRESS,'D');
    RETURN;
END
    INSERT_CARD;

```

Figure 2.47E. The INSERT_CARD subroutine

INSERT_CARD creates a new list component for the catalog card and inserts the component into the specified sublist. Insertion occurs in ascending sequence on accession number.

When all catalog cards have been inserted into the index list, the list is printed by the subroutine PRINT_INDEX given in Figure 2.47F. The subroutine uses the name of the index list (INDEX_LIST) as its only parameter and prints the list on the standard system-output file, SYSPRINT.

```

PRINT_INDEX:
    PROCEDURE(INDEX_LIST);
DECLARE
    (INDEX_LIST,SUBLIST) POINTER,
    POINTER POINTER,
    DESCRIPTOR BASED(DESCRIPTOR_ADDRESS)
    CHARACTER(10),
    1 CARD_IMAGE BASED(CARD_ADDRESS),
    2 ACCESSION#_IMAGE CHARACTER(10),
    2 DESCRIPTOR_GROUP_IMAGE
    CHARACTER(70);
    SUBLIST = INDEX_LIST;
    PUT PAGE LIST('OUTPUT FROM PRINT_INDEX
    PUT SKIP(2);
GET_SUBLIST:
    DO
        WHILE(SUBLIST<=>NULL);
        DESCRIPTOR_ADDRESS = GET_VALUE
        (GET_VALUE(SUBLIST));
PRINT_DESCRIPTOR:
    PUT
        EDIT(DESCRIPTOR_ADDRESS->DESCRIPTOR)
        (A);
    PUT
        SKIP;
        POINTER = GET_LINK
        (GET_VALUE(SUBLIST));
PRINT_CARDS:
    DO
        WHILE(POINTER <=> NULL);
        CARD_ADDRESS = GET_VALUE(POINTER);
    PUT
        EDIT(CARD_ADDRESS->CARD_IMAGE)
        (A);
    PUT
        SKIP;
        POINTER = GET_LINK(POINTER);
END_PRINT_CARDS:
END;
    PUT
        SKIP(2);
        SUBLIST = GET_LINK(SUBLIST);
END_GET_SUBLIST:
END;
END
    PRINT_INDEX;

```

Figure 2.47F. The PRINT_INDEX subroutine

Construction and printing of the index list is controlled by the procedure INDEX, which appears in Figure 2.47G. INDEX reads an arbitrary number of catalog cards from the standard system-input file, SYSIN, and allocates storage for each card in the storage area called DATA_AREA. If the number of cards exceeds the storage capacity of DATA_AREA, a message is printed to indicate insufficient storage. Reading then ceases, and only those cards allocated in DATA_AREA are indexed.

As each card is read, its descriptors are scanned, and the address of the card is inserted into the proper sublist for each descriptor within the index list. When all cards have been processed in this manner, the index list is printed, as shown in Figure 2.47H.

```

INDEX:
PROCEDURE;
DECLARE
  1 CATALOG_CARD,
  2 ACCESSION# CHARACTER(10),
  2 DESCRIPTOR_GROUP CHARACTER(70),
  1 CARD_IMAGE BASED
  (IMAGE_ADDRESS),
  2 ACCESSION#_IMAGE CHARACTER(10),
  2 DESCRIPTOR_GROUP_IMAGE
  CHARACTER(70),
  DESCRIPTOR CHARACTER(10),
  DESCRIPTOR_STRING CHARACTER(71)
  VARYING,
  LIST_AREA AREA,
  DATA_AREA AREA,
  (INDEX_LIST, AVAIL EXTERNAL)
  POINTER;
  INDEX_LIST, AVAIL = NULL;
  ON ENDFILE (SYSIN) GO TO PRINT;
  ON AREA BEGIN;
PUT
  LIST('INSUFFICIENT STORAGE FOR
  COMPLETE INDEX');
  GO TO PRINT; END;
  CALL AREA_OPEN(LIST_AREA,AVAIL);
  PUT PAGE LIST('INPUT TO INDEX:');
  PUT SKIP;
GET_CARD:
  GET
  EDIT(CATALOG_CARD)(A(10),A(70));
  PUT SKIP EDIT(CATALOG_CARD)(A);
  ALLOCATE CARD_IMAGE IN(DATA_AREA)
  SET(IMAGE_ADDRESS);
  IMAGE_ADDRESS->CARD_IMAGE =
  CATALOG_CARD;
  DESCRIPTOR_STRING =
  DESCRIPTOR_GROUP||' ';
NEXT_DESCRIPTOR:
  DESCRIPTOR = GET_DESCRIPTOR
  (DESCRIPTOR_STRING);
  IF DESCRIPTOR = (10)' '
  THEN GO TO GET_CARD;
  CALL INSERT_CARD(IMAGE_ADDRESS,
  GET_DESCRIPTOR_COMPONENT
  (DESCRIPTOR, INDEX_LIST));
  GO TO
  NEXT_DESCRIPTOR;
PRINT:
  CALL PRINT_INDEX(INDEX_LIST);
END INDEX;

```

Figure 2.47G. The INDEX procedure

```

OUTPUT FROM PRINT_INDEX:

BLACK
A9-1L7-RZ PAINTING BLACK WHITE SMALL
T2-XY4-16 PHOTOGRAPH BLACK WHITE LARGE
Y9-016-X8 SCULPTURE MARBLE BLACK SMALL

LARGE
T2-XY4-16 PHOTOGRAPH BLACK WHITE LARGE

MARBLE
Y8-123-X7 SCULPTURE MARBLE WHITE MEDIUM
Y9-016-X8 SCULPTURE MARBLE BLACK SMALL

MEDIUM
Y8-123-X7 SCULPTURE MARBLE WHITE MEDIUM

PAINTING
A9-1L7-RZ PAINTING BLACK WHITE SMALL

PHOTOGRAPH
T2-XY4-16 PHOTOGRAPH BLACK WHITE LARGE

SCULPTURE
Y8-123-X7 SCULPTURE MARBLE WHITE MEDIUM
Y9-016-X8 SCULPTURE MARBLE BLACK SMALL

SMALL
A9-1L7-RZ PAINTING BLACK WHITE SMALL
Y9-016-X8 SCULPTURE MARBLE BLACK SMALL

WHITE
A9-1L7-RZ PAINTING BLACK WHITE SMALL
T2-XY4-16 PHOTOGRAPH BLACK WHITE LARGE
Y8-123-X7 SCULPTURE MARBLE WHITE MEDIUM

```

Figure 2.47H. Printout from PRINT_INDEX

REVIEW OF LISTS OF LISTS

This chapter shows how to extend the flexibility of a pointer list so that it can be used to link other lists as well as data items (see Figure 2.48). The resulting list of lists uses a type code within each list component to distinguish

between sublists and data items. With this code, sublists can in turn contain other sublists to an arbitrary depth. As a result, new lists can be generated as the need arises during the course of program execution, and the programmer is freed from having to know the exact number of lists a program will require.

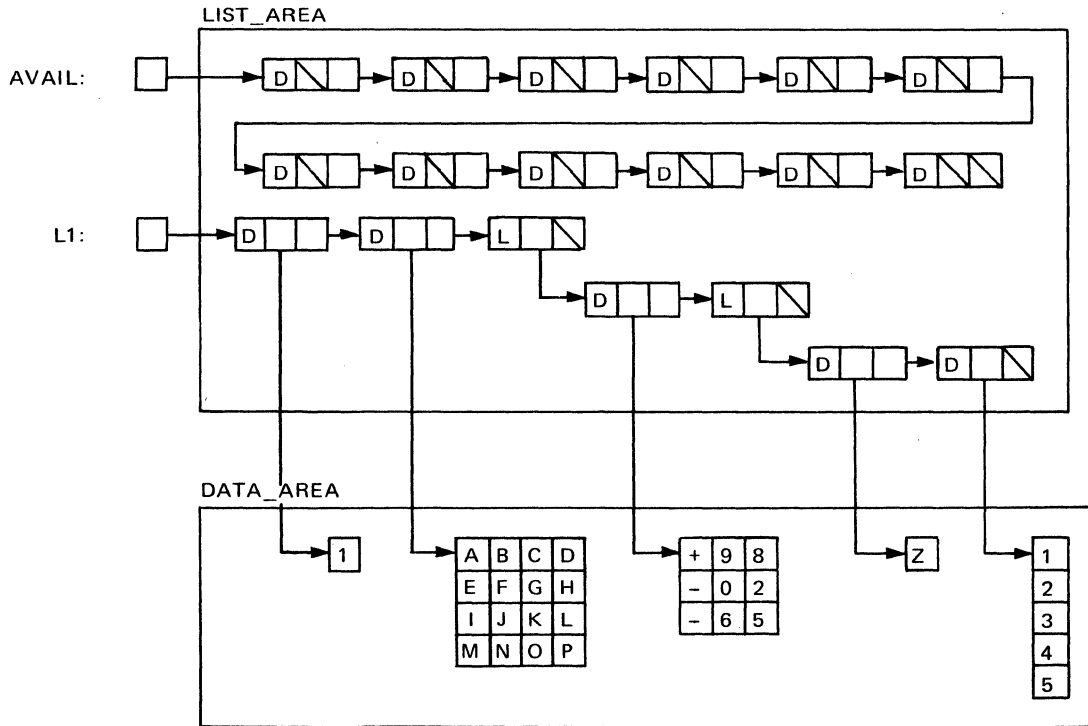


Figure 2.48. List of lists

SUMMARY

1. A list of lists is a more general type of pointer list.
2. Besides permitting element items, arrays, and structures to be members of a list, a list of lists also permits a list itself to be a member of another list.
3. A list of lists provides the same advantages as pointer lists: avoiding data duplication and reducing data movement.
4. A list of lists removes the need to know the exact number of lists a program will require during execution. As the need arises during program execution, a new list can be generated automatically and inserted into a master list of lists.

5. A type code within each list component determines whether the component specifies the address of a data item or the address of a sublist.
6. The subroutines and functions developed in this chapter for processing lists of lists fall into four categories:
 - a. Creating a list of available storage components
 - b. Manipulating component elements in a list of lists
 - c. Manipulating the top level of a list of lists
 - d. Manipulating all levels of a list of lists

Elementary procedures are developed first and used in turn to create higher-level procedures.

Index

	<i>Page Number</i>		<i>Page Number</i>
ADD_NODE subroutine	68	GET_NVT function	40
ADDRESS_LVT function	26	GET_TYPE function	30
ADDRESS_NVT function	25	GET_VALUE function	29
APPEND function	44	INDEX procedure	73
AREA_OPEN subroutine	23	Indexing	70
Array representation	18	INSERT_CARD subroutine	72
Binary tree lists	65	INSERT_NVT subroutine	36
Circular lists	21	LINKL subroutine	42
COPY_LEV function	58	LISP	22
COPY_LIST function	56	List storage	3
COPY_REVT function	46	Lists of lists	1, 15
COPY_REVT1 function	48	MEMBER function	62
COUNT_D function	52	Mixed data	4
Data lists	1	NODE function	67
Data storage	3	Null lists	17
DELETE_LIST subroutine	53	Parenthetic list representation	20
DELETE_NVT subroutine	54	Pointer lists	1, 3
EQUAL_D function	33	PRINT_INDEX subroutine	73
EQUAL_L function	60	REPLACE subroutine	64
FORM_BODY function	38	Searching	12
Freeing storage	5	SET_LINK subroutine	31
GET_DESCRIPTOR function	72	SET_VALUE subroutine	32
GET_DESCRIPTOR_COMPONENT function	72	Sharing data	4
GET_FD function	49	Sharing lists	20
GET_FDR function	50	SIZE_TOP function	35
GET_LD function	50	Sorting	10
GET_LDR function	51	Structure representation	18
GET_LINK function	28	T_PRINT subroutine	69
GET_NTT function	41	T_SORT procedure	69

READER'S COMMENT FORM

Techniques for Processing Pointer Lists
and Lists of Lists in PL/I

GF20-0019-0

Please comment on the usefulness and readability of this publication, suggest additions and deletions, and list specific errors and omissions (give page numbers). All comments and suggestions become the property of IBM. If you wish a reply, be sure to include your name and address.

COMMENTS

—
fold

—
fold

—
fold

—
fold

- Thank you for your cooperation. No postage necessary if mailed in the U.S.A.
FOLD ON TWO LINES, STAPLE AND MAIL.

YOUR COMMENTS PLEASE...

Your comments on the other side of this form will help us improve future editions of this publication. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material.

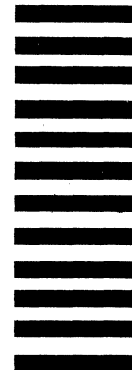
Please note that requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or the IBM branch office serving your locality.

fold

fold

FIRST CLASS
PERMIT NO. 1359
WHITE PLAINS, N. Y.

BUSINESS REPLY MAIL
NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES



POSTAGE WILL BE PAID BY ...

IBM Corporation
1133 Westchester Avenue
White Plains, N.Y. 10604

Attention: Technical Publications

fold

fold

Techniques for Processing Pointer Lists and Lists of Lists in PL/I Printed in U.S.A. GF20-0019-0



International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
[U.S.A. only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]



International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)