Z77-6254

TECHNICAL INFORMATION EXCHANGE

IBM®

August 8, 1966

LIST PROCESSING WITH MULTI-CELL DATA ELEMENTS

Mr. Stanley F. Zitello
IBM Corporation
2925 Euclid Avenue
Cleveland, Ohio 44115

This paper describes a set of list-processing features which can be implemented on a computer to allow dynamic storage allocation where the elements of data contain a multiple number of cells. Classical list-processing operates with unit cells. Dynamic storage allocation of data blocks is becoming necessary in areas such as file organization, time-sharing compiler design, Teleprocessing, information processing, and engineering design, among others. This paper discusses traditional list-processing, defines the features needed to handle multi-cell data, presents programming examples, and offers a set of FORTRAN subroutines which illustrate the proposed system.

IBM Corporation Technical Publications Dept., 112 E. Post Road, White Plains, N.Y. 10601

Z77-6254

TABLE OF CONTENTS

INTRODUCTION

In list processing a single block of storage of fixed size and location is used to contain data of dynamic size and structure.

One of the tasks of an assembler or compiler is to establish addresses for the object time storage of data. Even though a variable is used in only a small portion of a program, space is occupied for the variable throughout the execution phase. This "static allocation" is acceptable as long as the amount of storage involved is a small part of the total storage needed. When large amounts of storage are involved "dynamic allocation" of storage is desirable. List processing offers a means of obtaining dynamic allocation whereby storage is only used during the time the variable is needed.

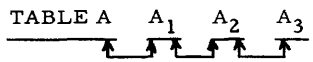
Internal storage in a computer is one-dimensional. It is numbered 0, 1, 2, and so on up to the limit of the machine. Compilers and assemblers give the programmer the ability to define data structures which are not one-dimensional, such as in a DIMENSION statement in FORTRAN. The processor converts references to data items in the structure into a one-dimensional mapping of addresses. In some cases there is no easy language word to describe a desired structure. In other cases the size of the structure cannot be determined until object time. In these instances list processing is an effective means of achieving the desired structuring of storage.

DEFINITIONS

Suppose that a programmer needs a table to store values for A_i, that the maximum size of the table is five entries, that each entry uses one word of storage, and that only three entries are being occupied. We might illustrate such a table as:

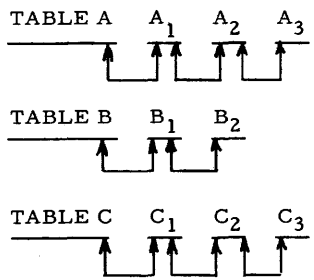
$$A_1\ A_2\ A_3\ 0\ 0$$

If list processing were to be used to store the table, a graphic representation might be:

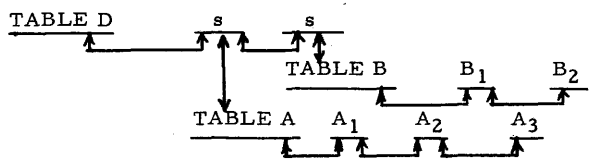


In the traditional method each data element occupies a word of storage, all of the elements are contiguous, and all five words are used even though only three are needed. In the list approach, each data element occupies a word of storage, however the elements are not, in general, contiguous. Also, storage is needed for a header element for the list (represented above by TABLE A) and for pointers (depicted as arrows) which logically connect the data elements.

A programmer using list processing reserves a single block of storage of fixed size and location. Within the block there can be any number of lists. For example, we might have lists for A_i, B_i, and C_i:



Part of the power of list processing is that a list can contain not only data elements but also other lists. For example, Table A and Table B could be "sublists" of a list TABLE D:



THE ELEMENTS

To implement the kinds of lists shown, there are three kinds of elements which must appear in storage: A list header (such as TABLE A), a data element (such as B_2), and a sublist pointer (such as "s", above).

Header

A list always contains one header element. The header identifies the list to the list processing routines. An "empty list" is defined as a list consisting of only a header element.

Data Element

Data elements contain the information stored in a list. For example A_1, A_2, and A_3 are the values that comprise the list whose header is TABLE A. A list consisting of a header and one or more data elements is defined as a "simple list."

Sublist Pointer

A list that is part of another list is called a sublist. To represent the sublist relationship internally, a sublist pointer is placed in the higher-level list. A list may contain only sublists or both sublists and data elements -- in either case it is defined as a "complex list." Two other important properties are: 1) A list may be a sublist of more than one higher-level list, and 2) A higher-level list may be a sublist of an even higher-level list.

The List Label

The three previous elements all appear in the block of storage used for lists. The actual storage address for each element

is not determined until execution time. It is therefore necessary that some item of information be fixed at compilation time to serve as a link between compile and object times. That item is called the "label" of a list. The use of a label is similar to a symbolic reference to an I/O unit whose actual address is determined at run time. When a list is created at object time, the address of the header of the list is placed in the fixed core location for the list label. The list header address is technically the "name" of the list and therefore the label can be thought of as the "alias" which the programmer uses in place of the name.

THE ELEMENT PREFIX

We now turn our attention to the manner in which list elements are linked together. Linkage is accomplished by prefixing each element in the list with three fields -- an identification code (ID) and two pointers. The ID is used to indicate which kind of element is being prefixed -- header, data, or sublist pointer. The pointers in the prefix are used to indicate the addresses of the logically-adjacent elements in the list. Normally one pointer indicates the next element in the list and the other pointer indicates the previous element; however, there are two special uses for the pointers: 1) In a list header the pointers are used to point to the first element in the list and the last element of the list, and 2) The last element in a list points to the previous list element and all the way back to the list header. Defining pointers in this fashion makes it possible to traverse a list either forward or backward from any element of the list to any other element, an ability particularly important for dynamic storage allocation.

The internal representation of the prefix is strongly machine-dependent. On a binary machine, the ID value could be represented in two bit positions of a word. On a decimal character machine the ID could be stored in a single core position. On a binary machine a pointer could be stored in sixteen bits, giving a maximum range of 65,536. On a decimal machine a pointer might be stored in three character positions. The total prefix would then be stored on a 36-bit binary machine in a single word and on a decimal character machine in seven core locations.

Suppose that a list consists of three data elements A, B, and C; furthermore that the prefixes for the elements are located at core addresses 1720, 1764, and 1782. Then B's pointers could be the actual machine addresses 1720 and 1782 or the pointers could be values directly convertible to machine addresses. For example, the pointers might have the values 20 and 82 and the value 1700 could be stored in a register, representing the beginning core address of the list area.

The list programmer must often assign a name to a pointer value returned to him by a list subroutine. A pointer value has attributes different from those of fixed-point numbers, floating point numbers, or character strings. In the FORTRAN implementation presented later in this paper, pointer values are treated as quasi-integers and so are given integer names, such as MARK, L4, and IADVD.

CELLS

Different computers often have different storage structures. To store a given integer may take one word, four characters, or two bytes of storage. To avoid storage-dependent terms, we will use the word "cell" in this paper to mean a sufficient amount of core to store a value. That value might be an integer, a floating-point number, or a pointer. Some systems use more core locations for floating-point numbers than for integers. The implications of such a system are discussed later in the paper.

In traditional list processing all of the data elements occupy single data cells. In the example above the entries A_1, A_2, and A_3 were each single data values in the list called TABLE A. This paper presents a system which allows the data entries in a list to consist of any number of data cells. In such an implementation, the entry A_1 could itself be an array of many data values. Within a given list each data element contains the same number of cells, but the number may vary from list to list.

USING LIST PROCESSING

None of the prevalent programming languages have statements for list processing. List processing has traditionally been implemented by adding subroutines to a standard language. In this paper are presented language statements that call upon FORTRAN list processing subprograms, both Subroutines and Functions. (The coding of the subprograms is shown later in this paper). The statements offered give a basic capability for using list processing with multi-cell data elements. No attempt is made to completely survey the many special purpose subroutines that can be added to a list processing system. For such information, the reader can turn to the documents listed in the bibliography.

In the following sections, statements are shown for creating a list, adding and deleting items from a list, forming sublists, and extracting data from lists. Each section is divided into two parts - Externals and Internals. The externals are concerned with the use and format of a list statement. This information is what the list programmer must know. The systems programmer is concerned with the internal mechanisms that occur as a result of the given list statement.

INITIALIZATION

Externals

Before any list processing can be done, it is necessary to initialize the system. The purpose of the initialization is to inform the list subroutines of the address of the block of storage reserved for lists and the total amount of storage reserved. In this paper, we will arbitrarily decide that the list area is at the beginning of the FORTRAN COMMON area and we need, therefore, communicate only the amount of storage being reserved.

To initialize the system, call a subroutine INTAS and supply it with a count. Also write a COMMON statement for the list area, for example:

```
COMMON KORE (1000)
CALL INTAS (1000)
```

Internals

The INTAS subroutine reserves part of the list area for its own use. Core is used to store the number of core locations left in the list area. Some core is used to point to the first core location available for lists. Finally, an empty list is created for the "delete" list. Whenever the programmer deletes an item, that item will be added to the delete list.

CREATING AN EMPTY LIST

Externals

Creating an empty list has two purposes:

1. The programmer assigns his own label to the list, and
2. The programmer designates the number of data cells that will be required for each data element in the list.

To create an empty list, use the function LIST and supply it with the number of data cells desired. The value of the function is the name of the list. For example, the following statement will create an empty list that can contain data elements of five cells in length:

```
MINE = LIST (5)
```

The label MINE is a variable name created by the programmer and represents the name of the list.

Internals

The LIST function must obtain sufficient contiguous storage to contain the following fields:

1. The prefix.
2. The value, N, supplied by the programmer for the number of data cells in each data element.
3. A field to store a pointer in case this list is made a sublist of another list.

Into the prefix is placed an ID of 1 to indicate a header element. Both pointers address the header itself to facilitate later processing. The programmer's input parameter is stored in the cell for N. The last field is set to zero. If the list later becomes a sublist then this field will point upward to the higher-level list. Lastly, the address of the header element is returned to the programmer's label.

PLACING DATA ON A LIST

Externals

Two statements are used to place data on a list - one statement is used to reserve a data element in the list and another statement is used to load data into the various cells in the element.

To reserve a data element, use the function NEXT and supply the label of the list. The value of the function is a pointer to the data element. For example, to reserve a data element on the list labeled MINE, write:

```
MARK = NEXT (MINE)
```

The variable MARK is a user-created name that points to the data element. MARK is subsequently used to load data into the element.

To place a data value into a data element, call the subroutine LOAD and supply:

1. The pointer to the data element;
2. The number of the data cell to be loaded;
3. The data value or name.

For example, to place the value 32 in the first cell of data element MARK, write:

```
CALL LOAD (MARK, 1, 32)
```

Internals

The function NEXT is supplied the label of a list. The header of the list is retrieved and the value N is extracted. NEXT must then obtain sufficient contiguous storage to hold the prefix for the data element plus the number of data cells desired. Using information from the list header, the prefix pointers are set to address the previous last item in the list and the list header. The ID is set to 2. The header for the list is changed to indicate the new last item in the list. To the programmer is returned the address of the data element just created.

The subroutine LOAD is supplied the address of the data element. This address points to the prefix of the element. Knowing the number of core locations required by the prefix and each data cell, the subroutine can calculate the address for data storage. The inputted data is then moved to the calculated address. Note that in a FORTRAN system using different amounts of storage for real and integer variables that it is necessary to have two separate loading routines, perhaps LOADR and LOADI.

FORMING SUBLISTS

Externals

To add a sublist to a list, call the subroutine JOIN and supply the labels of the major list and the sublist, as in:

```
CALL JOIN (MINE, L4)
```

MINE and L4 are the labels of previously created lists. Calling the subroutine JOIN makes L4 a sublist of MINE.

Internals

The JOIN subroutine obtains sufficient contiguous storage for a sublist pointer. This element is to contain a prefix and a pointer to the header of the sublist. An ID of 3 is placed in the prefix. One pointer addresses the previous last element in the main list and the other pointer addresses the main list header. The main list header is updated to show that the sublist pointer is now the last element of the list. Finally, a pointer is placed

in the header of the sublist. This pointer addresses the sublist element just created and is used later to move upward from a sublist into the main list. Notice that since a list can be a sublist of more than one list, the upward pointer will indicate the last list to which the sublist was joined.

READING LISTS

Externals

List structures are used to store data. The previously discussed subroutines make it possible to create structures and insert data into the lists. The programmer also needs the ability to extract data from lists. The process of retrieving previously stored information is called "reading" a list. To read a list, the programmer needs statements to move through a list structure to a desired data element and then extract data from that element.

Moving through a list is accomplished by using the function IADVD to advance to a data element and the function IADVS to advance to a sublist. In either case the programmer supplies the function with a pointer to a beginning element and a count of the number of elements to be traversed.

For example, to advance to the first data element in the list labelled MINE, write:

```
K = IADVD (MINE, 1)
```

The value of the function, K, is a pointer to the prefix of the data element desired. To advance to the third sublist in the list NAL, write:

```
MARKL = IADVS (NAL, 3)
```

To advance from a current element to the next element in the list, one could write:

```
IMAT = IADVD (IMAT, 1)
```

To advance backward through a list the count value is made minus. For example, to move from a current element to the previous element in the list, write:

```
IMAT = IADVD (IMAT, -1)
```

To extract data from a data element, use the function KONT and supply as parameters the pointer to the data element and the number of the data cell desired. For example, to obtain the fourth data value from the element IMAT, write:

```
NUM=KONT (IMAT, 4)
```

This statement will cause the desired data to be stored in NUM. The function KONT retrieves integers. To retrieve floating-point numbers, write:

```
FNUM = CONT (IMAT, 4)
```

Internals

The function IADVD works in the following manner. Depending upon the sign of the count specified, the routine extracts either the forward or backward pointer from the beginning element. The ID of the next element is examined. If the ID indicates a data element, then a counter is incremented and checked for an equal with the count specified by the programmer. If the values are equal the address of the current element is returned to the programmer. If the ID indicates a sublist pointer, the element is ignored and the routine moves to the next element. If the ID indicates a header, then the list has been exhausted and a value of zero is returned to the programmer.

The function IADVS works in the following manner. The ID of the next element is examined. If the ID indicates a data element the routine moves to the next element. If the ID indicates a header, then the list has been exhausted and a value of zero is returned to the programmer. When the ID indicates a sublist pointer, a counter is incremented and checked for an equal with the count specified by the programmer. If the values are

unequal the routine moves to the next element. If the values are equal then the pointer value is extracted from the sublist entry and returned to the programmer. This pointer value is the address of the sublist header. To complete processing, the routine places the address of the sublist element into the sublist header. The purpose of this action is to allow the programmer to later ascend from the sublist back into the main list. (In that case, the programmer uses the function IPOP which returns the programmer to the main list).

The two functions KONT and CONT extract data from an element. The address of the data element and the desired cell number are inputs to the routine. Knowing the number of storage locations occupied by a prefix and each data cell, the routine can calculate the address of the desired cell. The data value at that address is obtained and returned to the programmer.

DELETIONS

Externals

When a list or part of a list is no longer needed, the programmer can delete the unneeded elements, making their core locations available to new data.

To delete an item, call the subroutine REMOV and supply the pointer to the element. For example, to delete the element at MARK, write:

```
CALL REMOV (MARK)
```

Internals

The REMOV routine first checks to see if the input parameter points to a data element or a list header.

If the ID indicates a data element, the following actions are performed. From the given element the list is traversed until the list header is reached. The count of the number of data cells in each element is obtained from the header. This value is placed into the first data cell of the element being deleted. Next, the pointers in the elements before and after the deleted item are updated. The removed element is then added to the delete list. Later, when some routine requests a storage area, an internal subroutine will check the original storage block for sufficient storage. If the original block is exhausted, the delete list is traversed in search of an item of sufficient length. Notice that the delete list is one wherein each element can be of a different length and that the length is the first data word of each element.

If the argument for removal is a list header, the entire list is deleted. The REMOV subroutine moves to each data element in the list and places the count value into the first data cell. The entire list is then attached to the end of the delete list.

PROGRAMMING EXAMPLES

PROBLEM 1

Construct a simple list containing 20 data elements, each of which holds 10 data values. The values are obtained from 20 cards of 10 fields each.

```
1    LISTA=LIST (10)
2    DO 6 J=1, 20
3    MARK=NEXT (LISTA)
4    READ ( ) (A(I), 1=, 10)
5    DO 6 I=1, 10
6    CALL LOAD (MARK, I, A(I))
```

Statement 1 creates an empty list labelled LISTA. Each data element in the list can contain ten data cells. Statement 2 counts 20 cards. Statement 3 reserves a data element for each new card. Statement 4 reads ten values from a card. Statement 5 counts the loading of ten values into the list element. Statement 6 is executed ten times for each card, thereby placing all of the input values onto the list.

PROBLEM 2

Print the values stored in the list created in Problem 1. Print 20 lines of 10 fields each.

```
1    MB=LISTA
2    DO 6 J=1, 20
3    MB=IADVD(MB, 1)
4    DO 5I=1, 10
5    B(I)=CONT(MB, I)
6    WRITE ( ) (B(I), I=1, 10)
```

Statement 1 initializes a pointer value at the beginning of the list. Statement 2 counts 20 print lines. Statement 3 is used to advance through the list. The function IADVD is instructed to move from the element at MB to the next data element in the list. Ten values, counted by statement 4, are extracted from the element at MB by the function CONT in statement 5. The values are then printed by statement 6.

PROBLEM 3

Read and store a matrix of variable size. Use a list of one data element with sufficient cells to contain the matrix. A single card contains the number of rows and columns in the matrix. This card is followed by a set of cards, each of which contains one row of the matrix.

```
1    READ ( ) NR, NC
2    LISTM=LIST (NR*NC)
3    MARK=NEXT (LISTM)
4    K=O
5    DO 9 J=1, NR
6    READ ( ) (VAL(I), I=1, NC)
7    DO 8 I=1, NC
8    CALL LOAD (MARK, K+I, VAL (I))
9    K=K+NC
```

The card with the number of rows and columns is read by statement 1. Statement 2 creates a list labelled LISTM. The product of the number of rows and columns is the amount of data that will be stored in the list. In statement 3 a data element is reserved and is pointed to by MARK. The variable K, initialized in statement 4, is used to locate each row of the matrix. Statement 5 controls processing all of the rows of the matrix. Each row of the matrix is read by statement 6. Statement 7 counts the loading of each column entry. All data values are loaded by the LOAD subroutine in statement 8 . MARK is the pointer to the single data element of the list. The sum "K+I" positions data in the list element. The value of I represents the column number and K represents the beginning of each row of the matrix. K is updated for each row in statement 9.

Any value in the matrix can be retrieved by using the row and column numbers. For example, if the desired numbers are represented by MYR and MYC, write:

```
N=MYC+NC*(MYR-1)

VALUE=CONT(MARK,N)
```

When the matrix is no longer needed it can be deleted from storage by writing:

```
CALL REMOV(LISTM)
```

PROBLEM 4

Construct a list containing data elements followed by sublists. Input to the program is a deck of cards. Each card contains a code and a value. Each time the code changes place the code's value into a data element of the main list. Place all values with the same code into a sublist.

```
1    LISTM=LIST(1)
2    TYPO=9999.
3    READ (  ) TYPN,VAL
4    IF(TYPN-TYPO) 5,9,5
5    CALL LOAD (NEXT(LISTM), 1, TYPN)
6    LISTS=LIST (1)
7    CALL JOIN (LISTM, LISTS)
8    TYPO=TYPN
9    MARK=NEXT(LISTS)
10   CALL LOAD(MARK, 1, VAL)
11   GO TO 3
```

Statement 1 creates the main list. The control code is initialized by statement 2. Statement 3 reads a card with a code and a value. Statement 4 checks for a control break. For each new code, statement 5 loads the code into a new data element of the main list. Each new sublist is created in statement 6. Statement 7 joins the two lists, forming a sublist of LISTS. The new code is saved by statement 8. A new data element is reserved in the sublist by statement 9. In statement 10 each input value is placed into a data element of the sublist. Statement 11 returns to continue card reading.

PROBLEM 5

Read a value from the list created in Problem 4. The value is the last entry in the sublist for a code read from an input card.

```
1    READ ( ) TYPE
2    MH=LISTM
3    MH=IADVD(MH, 1)
4    IF (MH) 5,5,6
5    STOP
6    IF(CONT(MH, 1)-TYPE)3,7,3
```

```
7       MS=IADVS(MH,1)

8       VALUE=CONT(IADVD(MS,-1),1)
```

Statement 1 reads the code field. MH, the pointer for reading the main list, is initialized in statement 2. Statement 3 advances to the next element of data in the main list. A check is made to ensure that the list has not been exhausted by comparing for a zero value in statement 4. If a data element exists, its stored value is compared to the input code. When an unequal occurs, a branch is made to statement 3 which continues traversing the list. Statement 7 advances to the sublist immediately following the proper code in the main list. Statement 8 advances to the last element in the sublist and extracts the data item desired.

FORTRAN SUBPROGRAMS

The following pages contain listings of subroutines and functions discussed in this paper along with the subroutines called internally by the major subprograms.

These subroutines are not meant to be universally applicable. Indeed, the last page of listings contains the subroutines which should be written in assembler language in order to take advantage of the core structure of the machine of implementation.

An attempt has been made to keep the subprograms free of idioms unique to any of the FORTRAN dialects. To make processing more efficient, no subprogram ever goes deeper than one level into internal processing routines.

The basic unit of storage in this scheme is a FORTRAN integer word. The entire list area is a single-dimensional array (KORE). Thus, all pointers in this system are not machine addresses, but subscript values for each word in the array.

```
      SUBROUTINE INTAS(N)
      COMMON KORE(10)
C  NO. OF WORDS LEFT IN BLOCK
      KORE(1)=N-8
C  ADDR OF FIRST AVAILABLE WORD
      KORE(2)=9
C  SET UP EMPTY DELETE LIST,VARIABLE LENGTH
      IDUMY=LIST(-1)
      RETURN
      END&



      SUBROUTINE OBTAN(N,IADDR)
      COMMON KORE(10)
C  UPDATE NO. OF WORDS LEFT
      KORE(1)=KORE(1)-N
C  IS THERE ROOM IN MAIN BLOCK
      IF (KORE(1))2,1,1
C  YES, SAVE ITS ADDRESS
    1 KORE(3)=KORE(2)
      IADDR=KORE(3)
C  UPDATE NEXT AVAILABLE WORD ADDR
      KORE(2)=KORE(2)+N
      RETURN
C  SEARCH DELETE LIST FOR SUFFICIENT CORE
    2 KORE(1)=KORE(1)+N
      CALL SCANL(N,IADDR)
      RETURN
      END&



      FUNCTION NEXT(NAME)
      COMMON KORE(10)
C  GET NO. OF WORDS OF DATA IN EACH ENTRY
      NWDS=KONT(NAME,1)
C  GET A BLOCK OF FREE STORAGE
      CALL OBTAN(NWDS+3,NXAD)
C  GET ADDR OF LAST ENTRY IN LIST
      LAST=KPRA(NAME)
C  LOAD DATA ID, FWD POINTER AND BACK POINTER
      CALL STORL(2,NAME,LAST)
C  UPDATE LAST POINTER IN LIST HEADER
      CALL LPRA(NAME,NXAD)
C  UPDATE PREVIOUS LAST ENTRY TO POINT TO THIS NEW ENTRY
      CALL LNXA(LAST,NXAD)
      NEXT=NXAD
      RETURN
      END&
```

```
      FUNCTION LIST(NWDS)
      COMMON KORE(10)
C  GET 5 WORDS OF FREE STORAGE
      CALL OBTAN(5,NLIST)
      LIST=NLIST
C  SET ID AND TWO POINTERS
      CALL STORL(1,LIST,LIST)
C  LOAD THE NO. OF WORDS PER ITEM
      CALL LOAD(LIST,1,NWDS)
C  LOAD A ZERO INTO SUBLIST UP-POINTER
      CALL LOAD(LIST,2,0)
      RETURN
      END6


      SUBROUTINE JOIN(MAIN,ISUB)
      COMMON KORE(10)
C  GET ADDR OF LAST ENTRY IN MAIN
      LAST=KPRA(MAIN)
C  GET 4 WORDS OF FREE STORAGE
      CALL OBTAN(4,ISLP)
C  LOAD ID, FWD POINTER AND THE BACK POINTER
      CALL STORL(3,MAIN,LAST)
C  LOAD POINTER TO SUBLIST
      CALL LOAD(ISLP,1,ISUB)
C  UPDATE LAST ENTRY IN MAIN LIST HEADER
      CALL LPRA(MAIN,ISLP)
C  UPDATE PREVIOUS LAST ENTRY
      CALL LNXA(LAST,ISLP)
C  PLACE UP-POINTER IN SUBLIST HEADER
      CALL LOAD(ISUB,2,ISLP)
      RETURN
      END6


      SUBROUTINE LOAD(MARK,N,IDATA)
      COMMON KORE(10)
C  COMPUTE ADDR TO LOAD DATA
      IS=MARK+N+2
C  LOAD THE DESIRED DATA
      KORE(IS)=IDATA
      RETURN
      END6


      FUNCTION KONT(MARK,N)
      COMMON KORE(10)
C  COMPUTE ADDR OF DATA DESIRED
      IS=MARK+N+2
C  EXTRACT THE DATA
      KONT=KORE(IS)
      RETURN
      END6
```

```
      FUNCTION IADVD(MARK,N)
      COMMON KORE(10)
      IAM=MARK
      KOUNT=0
      KDIR=1
      NABS=N
      IF(N)1,7,3
    1 NABS=IABS(N)
      KDIR=2
    2 GO TO (3,8),KDIR
    3 IAM=KNXA(IAM)
    4 ID=IDOF(IAM)
      GO TO (5,6,2),ID
    5 IADVD=0
      RETURN
    6 KOUNT=KOUNT+1
      IF(KOUNT-NABS)2,7,7
    7 IADVD=IAM
      RETURN
    8 IAM=KPRA(IAM)
      GO TO 4
      END6


      FUNCTION IADVS(MARK,N)
      COMMON KORE(10)
      IAM=MARK
      KOUNT=0
      KDIR=1
      NABS=N
      IF(N)1,9,3
    1 NABS=IABS(N)
      KDIR=2
    2 GO TO (3,8),KDIR
    3 IAM=KNXA(IAM)
    4 ID=IDOF(IAM)
      GO TO (5,2,16),ID
    5 IADVS=0
      RETURN
   16 KOUNT=KOUNT+1
      IF(KOUNT-NABS)2,7,7
    7 IADVS=KONT(IAM,1)
      CALL LOAD(IADVS,2,IAM)
      RETURN
    8 IAM=KPRA(IAM)
      GO TO 4
    9 IADVS=MARK
      RETURN
      END6
```

```
      SUBROUTINE STORL(ID,NEXT,LAST)
      COMMON KORE(10)
C  GET ADDRESS OF NEW ENTRY
      IS=KORE(3)
C  STORE THE ID, THE NEXT ADDR, AND THE PREV ADDR
      KORE(IS)=ID
      KORE(IS+1)=NEXT
      KORE(IS+2)=LAST
      RETURN
      END&


      SUBROUTINE LNXA (MARK,IADR)
      COMMON KORE(10)
C  LOAD NEXT ADDR POINTER
      KORE(MARK+1)=IADR
      RETURN
      END&


      SUBROUTINE LPRA(MARK,IADR)
      COMMON KORE(10)
C  LOAD PREVIOUS ADDR POINTER
      KORE(MARK+2)=IADR
      RETURN
      END&


      FUNCTION IDOF(MARK)
      COMMON KORE(10)
C  EXTRACT THE ID CODE
      IDOF=KORE(MARK)
      RETURN
      END&


      FUNCTION KNXA(MARK)
      COMMON KORE(10)
C  EXTRACT NEXT ADDR POINTER
      KNXA=KORE(MARK+1)
      RETURN
      END&


      FUNCTION KPRA(MARK)
      COMMON KORE(10)
C  EXTRACT PREVIOUS ADDR POINTER
      KPRA=KORE(MARK+2)
      RETURN
```

BIBLIOGRAPHY

Berztiss, A. T., "A Note on Storage of Strings," Communications of the ACM, Vol. 8, No. 8, August 1965

Brown, W. S., "An Operating Environment for Dynamic-Recursive Computer Programming Systems," Communications of the ACM, Vol. 8, No. 6, June 1965

Lind, John H., "Implementation of List Structures," IBM Systems Research Institute Term Project No. 6-35

Weizenbaum, J., "Symmetric List Processor," Communications of the ACM, Vol. 6, No. 9, September 1964

_____, IBM Operating System/360 PL/I: Language Specifications (C28-6571)

_____, IBM System/360 Basic Programming Support FORTRAN IV (C28-6504)