MULTI-WORD LIST ITEMS

by

W. T. Comfort

November 1961

International Business Machines Corporation
Space Guidance Center
Owego, New York

ABSTRACT

The list concept as originally proposed by Newell, Simon and Shaw specified single computer words as elements of a list. This report describes the use of two or more consecutive words as one element. Such use results in a considerable saving both in space required to hold a given amount of data and in execution time required to effect a given process on the data.

Following a brief discussion of standard list structures with single-word items, the multi-word items are introduced. Then variable-length items are discussed, with the corresponding problems concerning the utilization of available space. Finally, several examples are given to illustrate the use of multi-word lists.

Catalog Descriptors:

1. Programming
2. Digital Computers
3. List Structure
4. Multi-word Items

PREFACE

This report represents the first published information concerning an automatic programming effort begun in November 1958. Its objective was to develop a compiler wherein the IBM 704 - 7090 was to be used to compile programs for a guidance and navigation computer. It was planned from the beginning that in order to gain experience in their use and to evaluate their capability and flexibility, the compiler should utilize multi-word lists wherever it appeared reasonable. The more interesting problems of the compiler (including the arithmetic scan, the automatic scaling of fixed point equations, the internal processing of a flow diagram, and the drum optimization problem) will be published in the near future.

MULTI-WORD LIST ITEMS

Webb T. Comfort

I. Introduction

In an automatic programming effort started in 1958, IBM's Space Guidance Center found that the technique of organizing a computer memory into list structures, an approach introduced by Newell, Simon, and Shaw[1] in 1957, was particularly pertinent. This report introduces the concept of the multi-word list item, which was developed to offset the inefficiencies of single-word items.

Following a brief discussion of standard list structures with single-word items, called SINGLETS, the multi-word items are introduced. Then variable-length items are discussed, with the corresponding problems concerning the utilization of available space. Finally, several examples are given illustrating the use of multi-word lists.

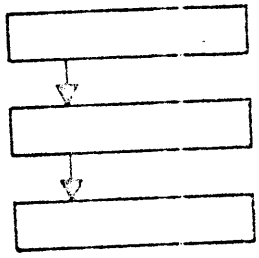
There are two objectives of this report. First, and most important, it will show how multi-word list items can provide considerable savings both in terms of execution time and memory space required over normal single-word items. Secondly, it will help the uninitiated reader to appreciate the simplicity of list-structures in themselves, and the ease with which a large class of relatively non-complex problems can be solved through their use.

A detailed description of the system resulting from the automatic programming effort is not given; future literature will provide this description.

II. Lists

A list is a connected sequence of items*. In the form in which lists were originally introduced by Newell, Simon, and Shaw[1], and in which they have generally been used, an item consists of one computer word. These items are connected through a field within each word which contains the address of the succeeding item. ** Note that since each item "points to" (i.e., contains the address of) its successor, successive list items need not be -- and in general are not -- consecutive words in memory. In fact, one of the powerful features of lists is their ability to utilize arbitrary, disjoint sections of memory.

Following Reference 1, such a list can be diagrammed as follows:



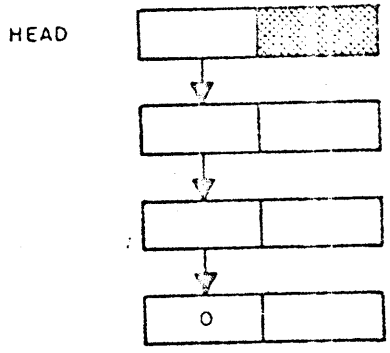
The arrow indicates the successor of each item. Later it will be useful

---

* This definition and those to follow are not intended to be rigorous, but rather to serve as shorthand for later discussions and to orient those who are familiar with a different terminology.

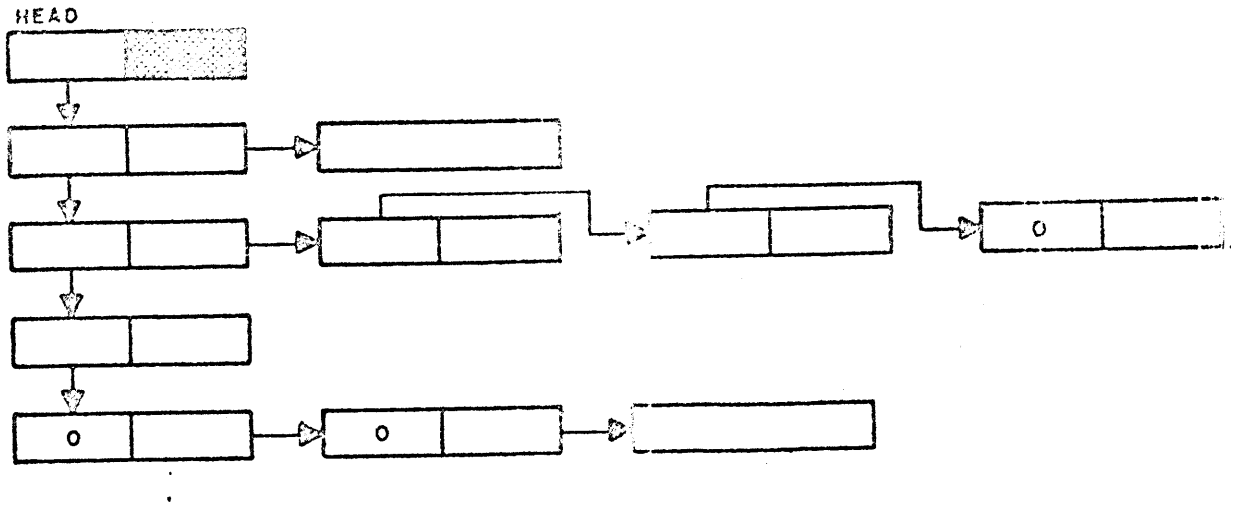
** Historical Note: Upon reflection it will be observed that a program for the IBM 650 (or any machine of the 1+1 address type) is such a list; it is a sequence of computer words (instructions), each of which contains the address of the next instruction.

to indicate the field in the element which contains the "pointer" (i.e., the address of the successor). Also, since the list items are arbitrary storage words, each list is provided with a "head", which is a known location in memory; that is, it is a word the location of which is known to the programmer, and through which he is able at all times to locate the list since it points to the first item on the list. Similarly, a special mark is required to indicate the end of a list. This is usually done by using a pointer of zero. Thus, a simple list containing three items would appear as follows:



(Note that a list head containing a zero pointer is an empty list; i.e., it has no items.)

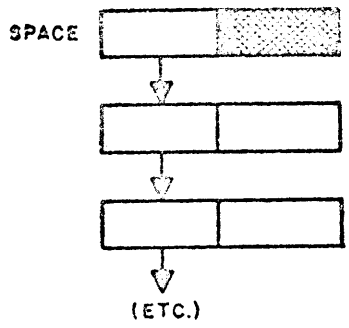
The remaining portion of each element may contain data, may point to a full word of data, or may point to another list (called a sublist); for example:

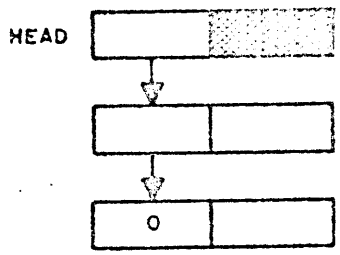


Here, the first item points to a data word, the second points to a 3-item sublist, and the fourth points to a 1-item sublist, which in turn points to a data word. Such a configuration is called a "list structure". It is necessary that either the programmer knows what each item represents, or else that there is sufficient information stored in the list so that the programmer may find out what each represents.

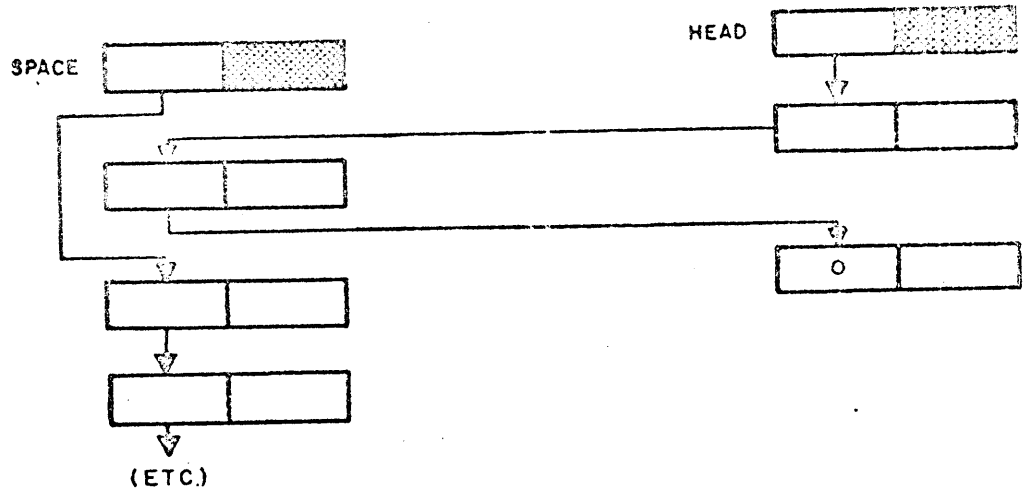
(Note that unless the computer word is long enough to hold two pointers, only the most rudimentary operations can be performed on the singlet lists. However, a pointer need not be of full address length. It may be several bits shorter, but with a corresponding decrease in the amount of storage available for use on lists. An alternate approach is the use of multi-word items, which are discussed later.)

In normal use, all available storage is initially put on a special "list of available space." As the program progresses and builds various lists, it obtains empty items from the available space list. Diagrammatically, consider the space list and another list:

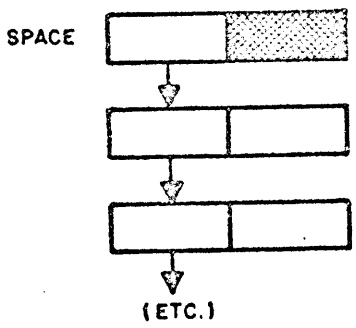


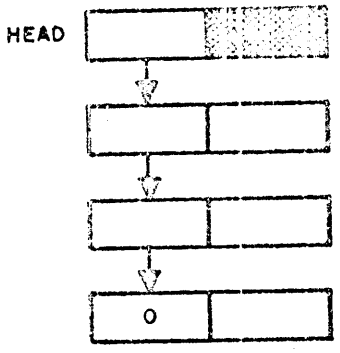


Suppose it is desired to take an item from the space list and insert it between the first and second items of the other list. The pointers are represented as follows:



The new diagram would look like this:





Thus, the most frequent list operations -- namely, inserting and deleting items from a list -- are accomplished through simple manipulation of the pointers. (Note that when an item is removed from a list, it is returned to the list of available space, for possible later use.)

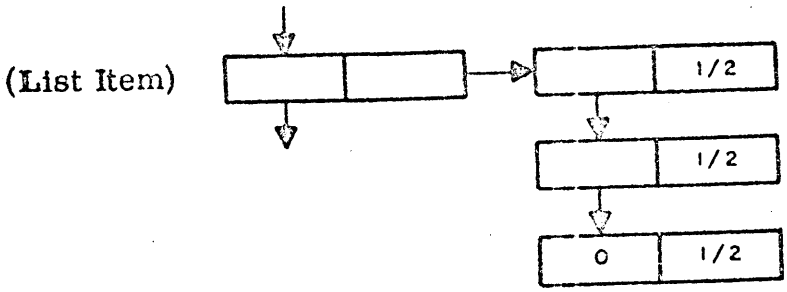
Now, it can be observed that in using singlet lists, somewhat less than 1/2 of every singlet must be given over to use as a pointer, thus decreasing the number of bits available to hold data or other information. To illustrate this, assume that a pointer requires exactly 1/2 of a word of a singlet item. Table I shows the number of singlet list items required to hold a given amount of data-type information.

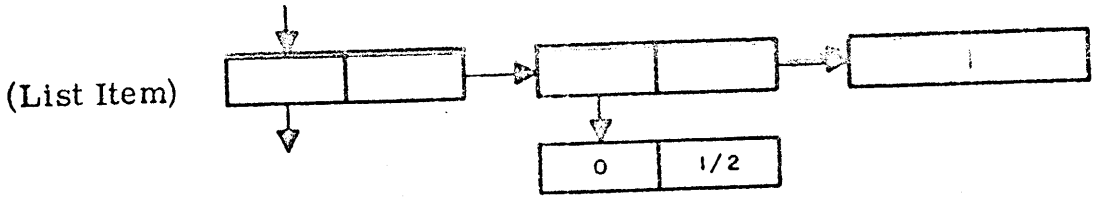
Table I

NUMBER OF SINGLET ITEMS REQUIRED TO HOLD GIVEN AMOUNT OF DATA

# WORDS DATA	# SINGLET ITEMS
1/2	1
1	2
1-1/2	4
2	5
2-1/2	6
3	7
•	•
•	•
•	•

Note that there are two ways in which 1-1/2 words of data can be stored:





Similar situations hold for larger amounts of data.

Thus, about half of storage is used up in pointers. In more complex problems, this apparent wastage of storage is considerably reduced by the amount of information implied in the structure, and through the use of "borrowed" sublists.[2] However, these topics are not pertinent to this discussion.

The other main problem with singlet lists involves the access time for data. It is not possible to directly compute the location of any particular item on the list; rather, the list must be searched from the head until the required item is reached. It follows, then, that if each item can only be located through a pointer, the number of instructions which must be executed to fetch a given amount of data depends directly upon the number of list items required to store it.

It will be shown in the next section that multi-word items for lists will greatly alleviate both of these problems.

III. Multi-Word Items

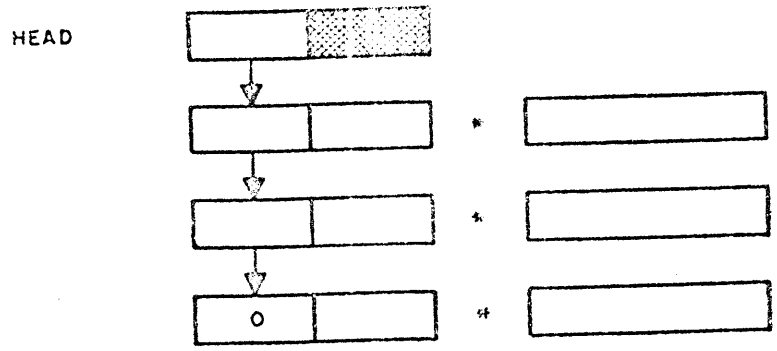
A multi-word item is a single list item which is organized in storage as a sequence of two or more consecutive words of memory. (It can be observed in the following discussion that the multi-word item concept is the basis for the applicability of the n-component element of Ross.[3])

The simplest form of a multi-word item is the two-word item, or "doublet"*, which consists of two consecutive words in storage. Diagrammatically, two words separated by an asterisk (*) are in consecutive memory locations (e. g. , L and L+1):

---

* Evans, et al. , [4] mentions what appears to be the same thing and gives it the same name, "doublet."

Thus, a doublet list has the following appearance:



The effect of this simple step is illustrated as follows: Where a singlet list requires 4 items to store 1-1/2 words of information, a doublet list requires only one two-word item. Also, where a singlet list requires 8 to 10 instructions to fetch the information, the doublet requires only 3, thus decreasing the program size as well as speeding up the execution time.

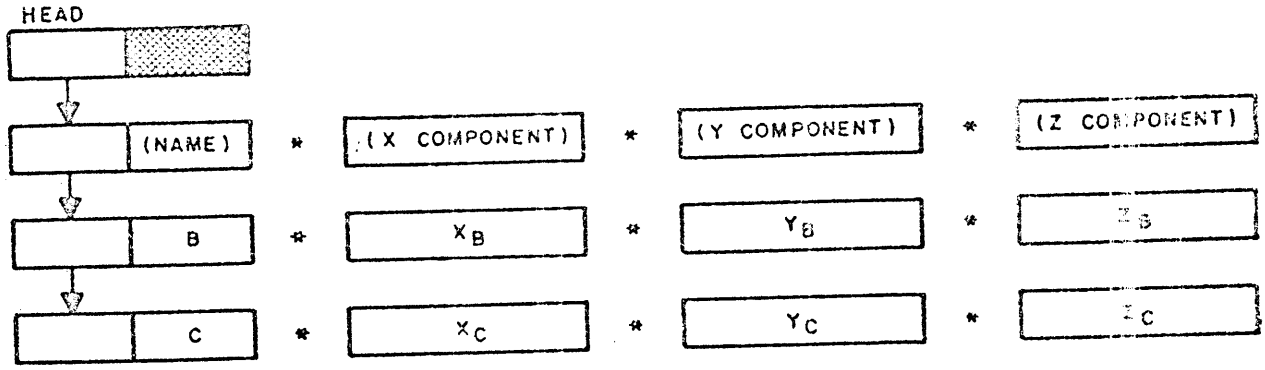
Perhaps it is appropriate now to make a comment on programming techniques. With singlet lists, it is sometimes debatable whether the process of sequencing down a list through selection and use of pointers should be accomplished by brute force modification of addresses (such as storing a pointer as the address portion of a fetch instruction) or by the use of index registers (whereby the pointer is loaded into an index register* and the fetch instruction has an address of zero, but is

---

* On IBM 704-9-90 machines, the index register is loaded with the complement of the address.

tagged by the index register). However, with multi-word items the indexing technique is necessary to obtain the indicated speed increases. In the case of the triplet, the index register need be loaded only once, whereupon fetch instructions with tagged addresses of zero, one, and two will make available the first, second, and third words, respectively, of the three-word item. (This is called reverse indexing by Ross and is discussed by him in detail in Reference 3.)

Figure 2 of Reference 2 shows (on the left half) a singlet list containing 22 singlet items. However, it appears that the same information could be stored on a simple list containing only 13 words when set up with four-word items. This is illustrated as follows:

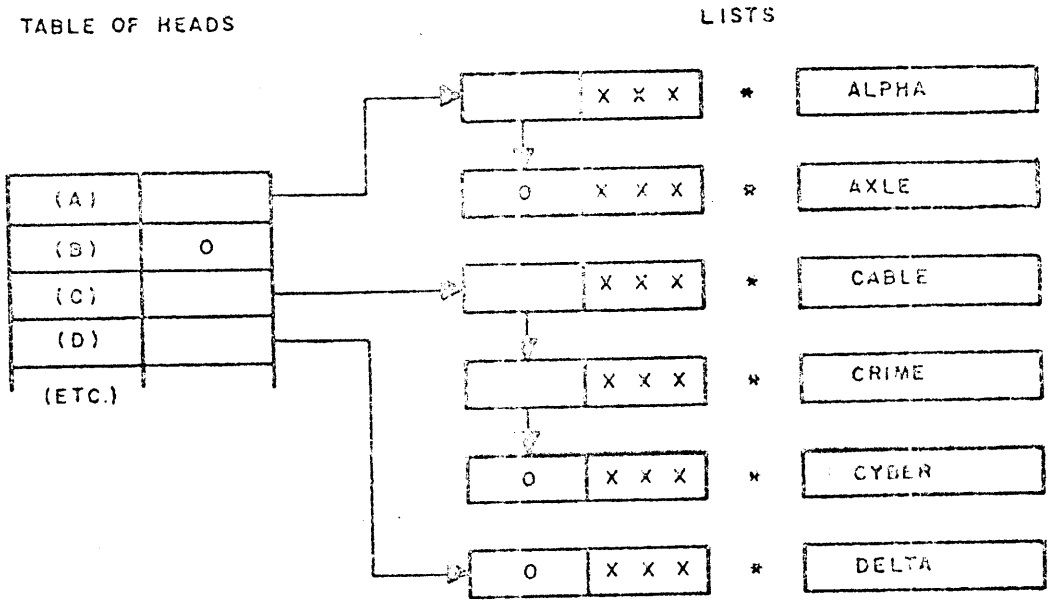


As an example of the use of a doublet list, consider the preparation of a symbol table. For a reasonably standard two-pass assembly program for a machine with random access memory, the first pass involves building a table of all different symbol names (restricting their length to 1 word) and assigning to each an absolute machine address (1/2 word). The

second pass then requires that every symbol be looked up in the table as it occurs and the corresponding absolute address be inserted into the instruction being assembled. The doublet list works very nicely in this case:

As the symbol table is being built, symbols may be sorted into the list alphabetically, if desired, or merely put onto the list in their order of occurrence. On the second pass of the assembly program, a simple searching routine will effect the translation.

To further illustrate the AD HOC use of lists, the obvious lack of high execution speed of such a symbol table can be off-set by a "Table of Lists." Here a table is set up of 26 list heads, one for each letter of the alphabet, and ordered according to the numerical code corresponding to the 26 letters. Thus, a symbol table would appear as shown on page 12. The advantage of this is that the first character of the symbol can be extracted, and the location of the head of the appropriate list in the table can be computed immediately. This is essentially a technique of block-sorting, which has been used on some occasions with 26 tables rather than lists. But in



such cases, this requires the provision of special overflow procedures when a table becomes full. If all 26 letters have an equal probability of occurring as the first character of a symbol, the search time is reduced by a factor of 26.

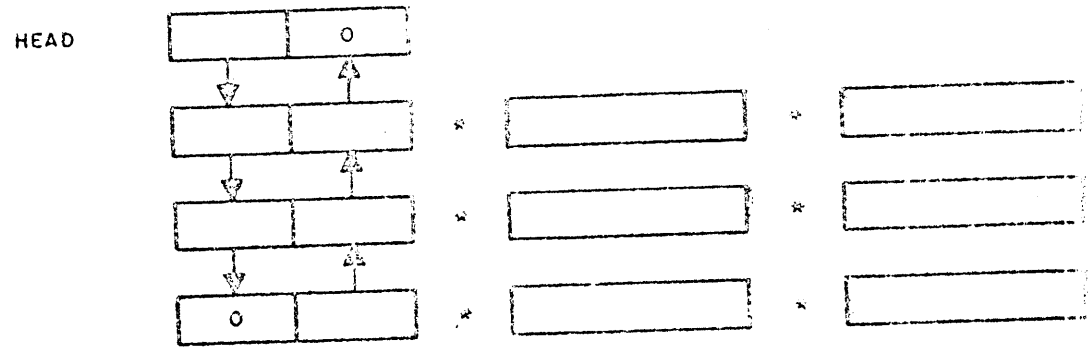
This technique can be extended to handle almost any problem of sorting within the high speed memory. For example, suppose it is desired to read in a deck of cards and sort them on some field prior to some other operation; e.g., editing a tape file. In the 7090 computer, 13-1/2 words of memory are required to store the contents of one card. By simply using a list with 14-word items (the additional 1/2 word holds

the list pointer), the sorting may be accomplished with a minimum of effort, and since the 1/2 word would normally be unused, no additional storage is necessary.

In the last two examples, the lists are of a continuously growing nature; i.e., no individual items are removed from the list until the function of the whole list is complete. In such cases, it is not necessary to go through the bookkeeping effort of constructing and maintaining a list of available space. It suffices to remember the starting point of the as yet unused portion of memory, and take a new item as required.

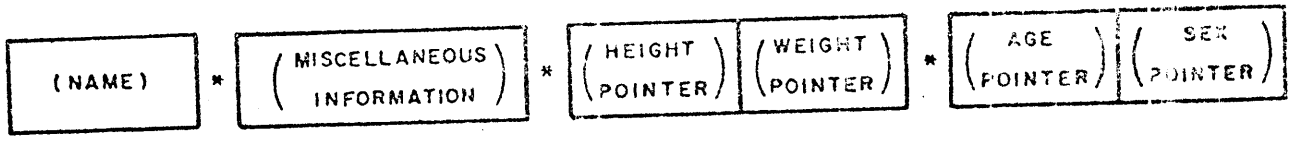
IV. Multiple Pointers

Probably the most significant contribution of multi-word items to the processing of list structures lies in the availability of multiple pointers. For example, a singlet list is a one-way device; it is possible to start at the head and move down the list, but it is not possible to move back up the list. This movement is easily accomplished with multi-word items, by simply making the item long enough to contain a pointer for this purpose. Thus, a two-way list of three-word items might appear as follows:



It would require somewhat more work to effect the insertion and deletion of items. However, this is readily accomplished, and there are situations where a two-way list is exactly what is desired. An example is in the process of automatic scaling during formula translation, to be described in detail at a later date. (See Section VII.)

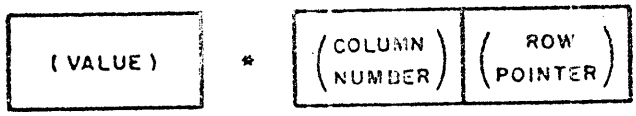
In Reference 5, the organization for an information processing system to be used for information retrieval is proposed. The basic idea is that a large file of items is given, e.g., a personnel file. Associated with each item are several characteristics such as height, weight, age, and sex. Associated with each characteristic is a set of values. For example, in a given file, the characteristic age may take on values 20, 21, 22, ... 35. For each such value, there is a list, which connects every item which has that value. Thus, every item contains a pointer for each characteristic, connecting it to the list which represents the proper value of that characteristic. Such items are readily obtainable through the multi-word technique. The following diagram illustrates a possible item of this type:



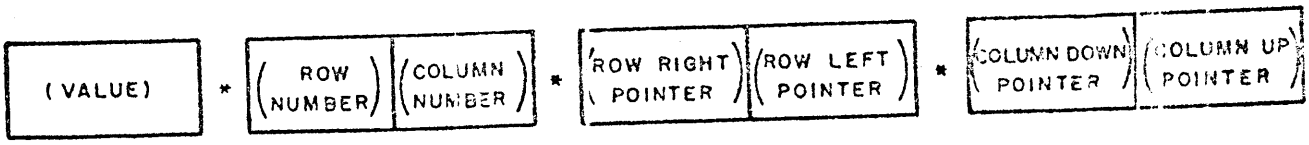
If the file of such items represents a college student body, an enterprising basketball coach may wish to obtain the names of all male students who are over 6 feet 1 inch in height and over 160 pounds in weight. In this configuration, it is necessary to search the lists representing the proper

values of each characteristic in sequence, selecting those items which appear on all. In this way, it is unnecessary to search all items in the file, only those which have the proper value of at least one characteristic. It is not proposed at this time to discuss whether or not this is the best technique for solving this type of problem. However, it should be pointed out that whereas the reference proposes a machine to operate in this fashion, multi-word items with multiple pointers allow the same flexibility on a general purpose machine with a minimum of space.

Another possible application of multiple pointers might arise in working with very large but very sparse matrices. For example, some linear programming problems may involve a 1000 by 5000 matrix with only 10,000 to 25,000 non-zero elements. One way to store only the non-zero elements might be to have a list for each row. Then the non-zero matrix elements would each be represented by an item of the following form:



To locate element (i, j), it is necessary to search the i list for an item with column number j. However, to perform the more general computations of linear programming in a reasonably efficient way, it might be necessary to provide a list for each column as well, and in fact to make them two-way lists. This can also be done quite readily:



Thus, having located element (i, j), it is relatively easy to locate elements (i-k, j) and (i, j-l).

The specific application where the multiple pointers proved most valuable involves the storage in memory of a flow diagram in order to analyze its flow characteristics. This will be discussed in considerable detail in the near future. (See Section VII.)

V. Variable Length Items

It has been implicitly assumed in the preceding discussion that if, for example, three-word items are being used, then all lists have three-word items and the space list problem is quite simple; namely, it is itself a list with three-word items. However, this need not be the case. There might be several lists simultaneously in storage, each with different size items. Alternatively, there may only be one list structure which itself has items of various sizes. In either case, the problem of how to handle the space list becomes significant. *

One approach is to provide a separate space list for each size item. However, to do this directly requires that the programmer decide

---

* As indicated previously, if the list length is monotonic increasing — i.e., no items are returned to the space list — there is no problem, since there is no space list, as such.

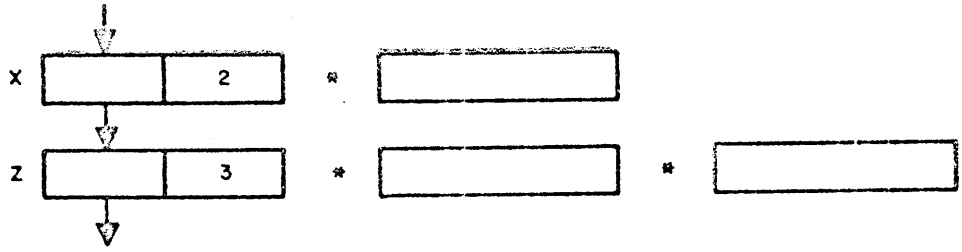
A Priori how much space will be alloted to each list, and this is directly contrary to one of the basic ideas behind the list structure concept; namely, that the programmer does not decide this, and if there is any space available, it can be used anywhere.

A modification of this approach will improve things somewhat. Suppose there are three types of items: singlets, doublets, and triplets. Initially, all available space is placed on the triplet space list (both doublet and singlet space lists being empty). Thereafter, if a doublet is required, and the doublet space list is empty, a triplet can be obtained and divided into a doublet and a singlet, each of which goes on its corresponding space list. If a singlet is needed (and the space list is empty), either a triplet can be obtained and split into a doublet and singlet, or a doublet can be used to get two singlets. When singlets or doublets are returned, they go onto the proper space list. This to a certain extent alleviates the problem indicated above; namely, as long as there is a triplet available, a doublet or singlet may be obtained. However, it is not unreasonable to expect that some problems may require triplets when the only space left consists of singlets and doublets. The finding of new triplets from a set of doublets and singlets can be accomplished, but it is quite inconvenient.

Consider now a generalized space list, i.e., one from which an item of arbitrary length can be taken, if that many consecutive words exist anywhere in available space. Such a list can be realized by having it consist of variable length items. Initially, it contains only one

item which is made up of all words of available space. When a specific item is required, the proper number of words are removed from the space list item, thus reducing its size. Then, when an item is to be returned, since it is desired to determine whether or not this item fits consecutively onto some current space list item, it is sorted onto the space list.

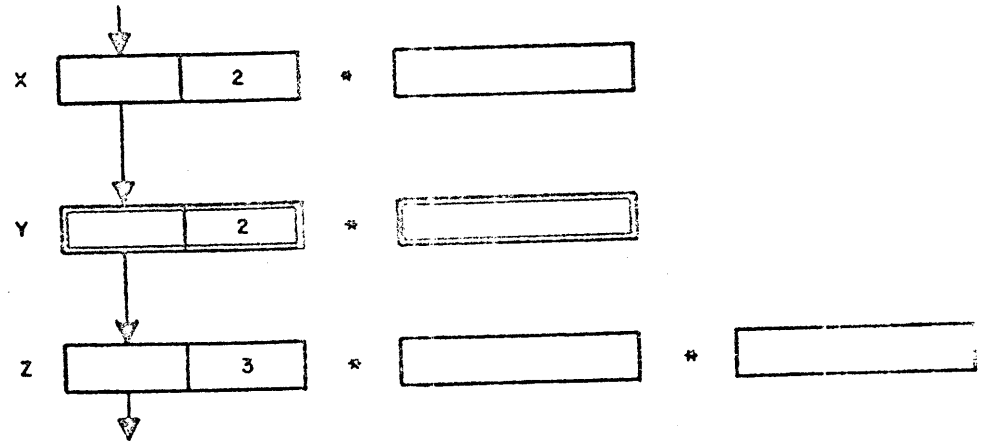
To illustrate how such a list might be handled, suppose the following is a portion of the space list:



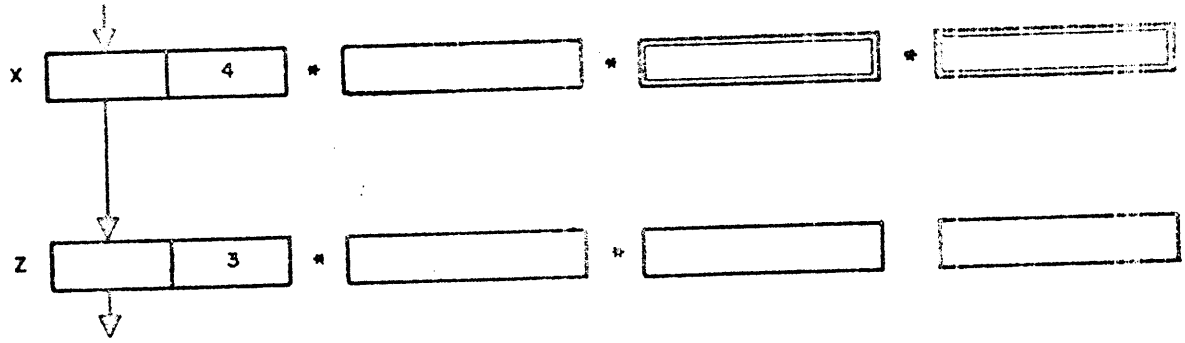
The number represents the length of the item, which is required for the sorting process; X and Z represent the memory locations of the first word of the two items. Suppose a two-word item is to be put back onto the space list:

However, if possible, it is to be put back in order to make a longer item. Therefore, it is necessary to first search down the items of the space list to the point where $X < Y < Z$. Then, the doublet can go onto the list in four possible ways.

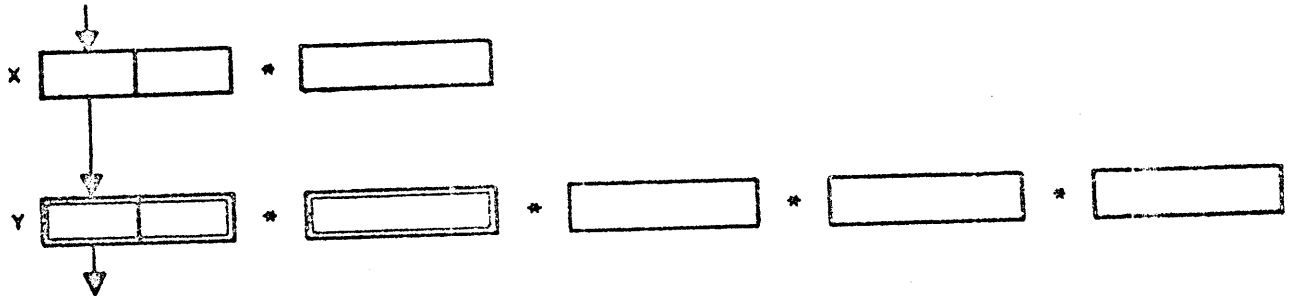
(1) Between the two items (in this case, $X + 2 < Y < Z - 2$):



(2) At the end of the preceding item (in this case, $Y = X + 2$):



(3) At the beginning of the next item ($Y + 2 = Z$):



(4) Both situations (2) and (3) hold (X + 2 = Y = Z - 2), resulting in one space list item which begins at X and is seven words long.

In this way the space list items are always of maximum length. When it is desired to select an item of a given length, the space list is searched until an item is found which is at least as long as the one desired, whereupon the proper number of words are removed from the space list item.

Admittedly, this operation takes a small but significant amount of time, since the space list must be searched both coming and going. However, in those cases where simpler techniques are not sufficient, it is the price to be paid. *

VI. Conclusions

There are certain comments which can properly be made at this point.

1. A list is basically a simple device, notwithstanding the pseudo-mathematical haze which has arisen from some areas.

---

* A 7090 program which handles such a space list has been in operation for almost two years. While there are no direct utilization figures, this routine does not appear to significantly affect the operating time of the main program using it.

2. A structure of multi-word items, if applicable, requires less storage and operation time than the corresponding structure of standard singlet items.

3. A large and complex list-processing programming system, such as IPL or LISP, is not necessary in order to obtain the advantages which the list concept has to offer.

4. A list structure can be made, in many instances, considerably more efficient than has been demonstrated in the past. All that is needed is some common sense and ingenuity on the part of the user.

5. A list structure is good for some problems; tables are good for others; and in some cases a combination of both is appropriate. In general, both should be considered.

ACKNOWLEDGMENTS

The author wishes to acknowledge the assistance of F. R. Palm, J. S. Hughes, J. W. Joachim, and E. S. Schulze for much of the programming and establishing of the techniques discussed in this report.

REFERENCES

1. Newell, A., and Shaw, J. C., "Programming the Logic Theory Machine," Proceedings of the Western Joint Computer Conference, 1957.
2. Gelernter, H., Hansen, J. R., and Gerberich, C. L., "A Fortran-Compiled List-Processing Language," Journal of the ACM, April 1960.
3. Ross, D. T., "A Generalized Technique for Symbol Manipulation and Numerical Calculation", Communications of the ACM, March 1961.
4. Evans, A., Perlis, A. J., and VanZoeren, H., "The Use of Threaded Lists in Constructing a Combined ALGOL and Machine-Like Assembly Processor," Communications of the ACM, January 1961.
5. Gray, H. J. and Prywes, N. S., "Outline for a Multi-List Organized System," Presented at the 14th National ACM Meeting, September 1959, MIT.