DATA STRUCTURES
AND THEIR
REPRESENTATION IN STORAGE

BY

M. E. D'Imporio
ABSTRACT

An attempt is made to define certain basic concepts involved in the representation of data and processes to be performed upon data within a high-speed, random-access computer memory. An important distinction is made between "data structures" and "storage structures," and it is suggested that the programmer or problem-solver will find his effort amply repaid if he makes a thorough initial study of the data for his problem in terms of various alternative "data structures," or patterns of mutual accessibility among elements of data, defined independently of any particular computer order code or programming language. An artificial problem is analyzed, and nine different solutions, involving different "data structures," are described and compared to illustrate the application of the concepts and definitions provided. The formation of operands from particular data structures and the construction of new data structures during the course of a problem solution are then discussed. A method is described for representing data structures in diagrams and tables, and for representing storage structures in diagrams, in accordance with the concepts and definitions previously introduced. Twelve recently developed memory systems which appear to have a high potential utility for information storage and retrieval and for natural language processing are described and analyzed with the aid of the diagrams and tables. Some tentative conclusions are drawn concerning the comparative utility of the ten memory systems and programming languages for problems involving storage, manipulation, and retrieval of intelligence information from natural language text.
INTRODUCTION

The past few years have seen the development of a rapidly increasing interest in artificial intelligence, symbol manipulation, information storage and retrieval, and similar applications of computers. This trend has brought with it a growing need for methods of handling irregular, dynamically changing, and highly complex operands. As a result, an ever-swelling throng of specialized storage organizations and data-handling techniques are pressing themselves upon the attention of anyone attempting to keep up with current literature in the computer field. The burgeoning population of special structures and languages, each accompanied by its own logical or mathematical model and a new array of formidable terminology, tends to confound and overwhelm the average reader approaching the literature with a set of actual problems which he feels may call for some one of the new techniques. He must attempt to winnow out of the jumble of imposing mathematics and idiosyncratic nomenclature those novel and useful contributions which each writer really has to offer. He must also decide whether his problems actually require the new facilities enough to warrant his paying their frequently high cost in programming time, running time, or storage space, and, in addition, taking on the burden of learning to use an unaccustomed and perhaps bizarre way of thinking about his data and processing steps. Since many of the proposed techniques involve interpretive programming systems, software simulation of unconventional memory structures in conventional, sequentially addressed hardware, and the handling of data in quantities or by methods that are not "built in" to conventional input-output and addressing systems, they are apt to be prohibitively wasteful if inappropriately used.

The examination of data structures and storage structures presented in this paper was motivated by the writer's practical need for a method of assessing and comparing the possible value of various programming techniques for a certain class of computer applications — those essentially non-numerical applications involving the storage, manipulation, and retrieval of information from natural language text. First, an attempt will be made to clarify certain distinct basic concepts involved in the matter of data and its representation in a computer memory, and to relate them to one another within a coherent scheme. When these concepts are explicitly distinguished and disentangled from one another, they may be utilized more advantageously in the machine solution of problems. Such distinctions and clarifications become increasingly important as the problems under consideration become increasingly complex, less well understood, and less amenable to familiar algorithms of applied mathematics or of data processing. After certain basic terms have been defined, a method of describing and diagramming data structures and storage structures will be presented, which will make use of these concepts. With the aid of these analytic tools, twelve recently developed programming languages and programmed memory structures will be examined and displayed side by side, so as to facilitate their comparison for the purposes of a potential user. In addition, some tentative conclusions will be drawn regarding the utility of these techniques for the handling of natural language text.
THE DISTINCTION BETWEEN DATA STRUCTURES AND STORAGE STRUCTURES

It is important to distinguish explicitly between computer-independent data elements and relationships of accessibility or ordering among them, on the one hand, and the computer storage slots or registers and their physical or "logical" adjacency in the computer memory as it is organized by a specific hardware or software memory system, on the other hand. While this point may appear obvious to the reader, the blurring of this distinction frequently leads to confusion and loss of efficiency, and denies to the problem solver the precision and flexibility of control he should exercise over his tools and raw materials. There are various ways of representing a given data structure (e.g., a "tree") in storage structures, and the same storage structure may be used for several different data structures. Particular data structures, as defined here, are the products of a particular human interpretation of a problem, expressed in the preliminary preparation and recording of the data and in the choice and ordering of processes that take place in a particular program. They are not invariant, inherent, or necessary parts of the data itself, but are the result of a certain way of looking at the data and the problem solution.* Since data structures are the products of human understanding and purpose in defining a solution to a problem and are often chosen without the explicit awareness of the problem solver, they tend to be taken for granted, and attention is directed solely toward the storage structures in which the data will be arranged and manipulated and which the problem solver must explicitly define and arrange. Hence, properties that belong in the interpretation of the data and the meaning of the data in the problem easily become confused with and unnecessarily tied down to the particular storage structures that were selected to handle them in a programming system for a computer.

The assumption is made throughout this paper that the problems to be solved are of the novel, irregular, and highly complex variety which are not economically soluble by conventional techniques. It is also assumed that the user will operate upon his data within high-speed core memories (or some functionally equivalent memory; i.e., a memory all of whose storage slots are equally accessible and accessible in the same brief time, measured in microseconds).

*The following remarks concerning human perception (E.B. Hunt, Concept Learning, Wiley, New York, 1962, p. 106) are interesting in this connection: "After reviewing several current theories of perception, Allport (1957) concluded that one of the most important aspects of perception is the subject's choice of structure. By 'choosing a structure' Allport meant essentially what we have called 'selecting a description process.' There are many ways in which an object can be described. The choice of a particular description will depend largely on the compatibility of the information that can be obtained with it. . . . Description processes cannot be applied to one object in isolation. Several objects will be in the person's span of attention. Their descriptions must not conflict." Allport (1957) refers to F. H. Allport, Theories of Perception and the Concept of Structure, Wiley, New York, 1955.
This assumption implies that the user will not have to deal with "built-in" storage structures or external file structures, such as that of magnetic tape, which allows only serial access to information, or the structures of other mass storage media (e.g., discs or drums) that may permit access only at relatively large, fixed intervals. It implies, further, that the user has some choice or control over the medium through which the data is to enter the computer, (i.e., he will not be required to accept a specifically formatted card deck or tape with a "built-in" segmentation and structure imposed by considerations external to the problem he wishes to solve). In the circumstances where these assumptions hold, it appears to the problem solver's advantage to analyze his problem by a computer-independent approach such as the "data structure" analysis suggested here, and to produce a number of alternative possible "hand" solutions of the problem, involving alternative ways of looking at the data in different arrangements, before getting down to the details of implementation on a computer.

DATA STRUCTURES - Concepts and Terminology

The data structures chosen by a human being in the solution of a problem are, essentially, sets of possible paths of accessibility among different elementary "atoms" of data. The choice of elementary particles or segments of data useful in a given method of solving a problem is in itself an important initial step in problem definition. These elements will be called data items in this paper. The determination that a certain data item should be directly accessible from certain others, and indirectly accessible in various degrees along paths among several others still, defines a data structure involving these data items. The choice of a data structure makes certain items immediate neighbors, others more distant "acquaintances." It is convenient for graphic purposes to think of a data item as having a right-hand, left-hand, upper, and lower neighbor, as they might be represented in a two-dimensional plane. The "next-door neighbor", adjacency, or immediate accessibility relationship between data is brought about by some activity that is being performed by the problem solver: he is picking up, attending to, or pointing at the items one after another in some order in space or time, or he is producing or putting away the items in some order. When data items along some path of accessibility allowed in a data structure are selected or attended to sequentially for the purpose of applying to them some operation in a problem solution, the resultant sequence of data items is an "operand" as the term is defined in this paper. In fact, the primary purpose in setting up data structures is to make possible the formation of those operands required in the course of a solution.

When the problem solver has decided upon his data items - bits, characters, words, phrases, sentences, numeric values, documents, etc. - and the set of "neighbor" relationships he wishes to define among them to form data structures, he is in a better position to re-examine the computer manuals or descriptions, or the manuals of programming languages or storage techniques available to him, and to choose freely the best way to implement his data structures in storage structures. Delaying the adoption of a specific programming language or storage structure permits choices and insights which the user might not have had if he had analyzed and solved his problem within a particular scheme from the start, even though he may have to revise some of his ideas after he chooses a set of storage structures for the computer implementation. In this connection it is interesting to
note that many of the new "information processing" or "list processing" techniques were developed as a result of attempts to simulate human problem-solving behavior on a computer. Examining the computer-independent data structures set up by human problem solvers, the investigators saw that familiar serially addressed storage structures were not appropriate or economical, and sought to develop new ones that, while more expensive to implement, could be used with greater efficiency and so provided a net saving that afforded a major advantage. It has become increasingly clear that conventional data processing procedures, apparently economical at first sight and customarily used to good advantage in solving problems involving simple, regular data structures, are frequently not good or even feasible procedures for the complex problems under discussion here.

PROPERTIES OR DATA STRUCTURES

Data Ordering. - The property of data ordering specifies the presence or absence of functional order among the elements of a data structure, and the nature of the order if it is present. Order in a data structure means, essentially, that the data items within the structure were produced in a certain sequence which is important in the method of solution and must be preserved, and that the data items must be obtained and acted upon in a certain sequence. Data ordering may be completely absent, in which case data items are batched in unordered collections. If order is present, the data items are listed, and the lists may be one-way or two-way. The term "list", in general English usage, means simply a roster or catalogue of items; it implies an ordered sequence of items recorded in some medium. Each item has a proper "place on the list" where it may be found, but no restrictions are placed on the nature of the ordering principle (alphabetical, by categories, by size, etc.) or on the method by which the ordering is represented or recorded. In the computer field, however, the term "list" has apparently come to mean a very specific kind of ordered sequence, whose order is recorded in one particular way in storage. Primarily as a result of the work of Newell, Simon, and Shaw et al., the term has been given the meaning of a serial arrangement of storage cells, each containing the address of its successor in a chain. It appears somewhat unfortunate to restrict a term of such general utility to so specialized a meaning: in this paper, "list" will be used to refer to any ordered set of data items, regardless of the ordering principle or the mode of its representation in storage. It will be used alone, unmodified by any additional phrase, only in discussing data and not for storage cells, since it is intended that storage arrangements will be described by terms more exact and specific in their reference to the type of arrangement involved.

Batched Data. - There is not functional order among the items that make up a batched data structure. It does not matter in the solution of the problem which item of data is dealt with first or last, or which items occurs before or after some other one. All that matters is the boundaries of the given batch setting it off against other data structures, the number of items it contains, or how to tell which items are members of the batch and which are not. All the items in a batch are dealt with in the same way, like grain being ground in a mill. This way of handling
data is the one overwhelmingly preferred for most machine problem solutions; much ingenuity is expended by programmers upon the effort to redefine problems in such a way that data structures can be batched. It is important to note that the essentially nonordered or interchangeable property of data items in a batch is independent of the way they happen to be stored in a machine memory. In most computers, storage cells must be processed one at a time, so that the data items, or parts of them, will be acted upon sequentially in some order in any case. The point to be emphasized is that with batched data the order in which the items occur is accidental or arbitrary for the purposes of the problem solver; they might be shuffled within the storage slots of the block where they are stored without affecting the problem solution.

**Listed Data.** - A list, as defined above for purposes of the present paper, is an ordered set of data items. The position of an item relative to other data items is important in problem solution and must be recorded and preserved. There are many problems which may be solved more efficiently by defining ordered data structures, or lists, than by batched collections. Even though the bookkeeping required to keep track of the lists and manipulate them may be more demanding, there need be far fewer wasted or unnecessary references to data items. Only those data items that are needed at a particular point in time need be obtained or stored, and a given data item can be unambiguously dealt with in only one way because it occupies a particular position in a meaningful sequence instead of all possible actions always having to be performed on the maximum possible number of data items, as must be the case in batched data structures.

**One-Way Lists.** - Data items are scanned or stored in a line, as the words of a sentence are written or read across a page. Ordered data structures may be considered as having attached to them pointers that move along the data structure and point successively to items in the order embodied in the list. The pointer on a one-way list moves only in one direction, perhaps skipping over several items, but never "back-tracking" in the opposite direction.

**Two-Way Lists.** - Data items are still arranged in single lines, but now the pointer on the list moves in either direction from any point on the list.

**Last-In-First-Out Operand Lists.** - The elements of a data list may be obtained to form an operand in "LIFO" (last-in-first-out) order as well as in the normal order. LIFO ordering is not, strictly speaking, a "property" of data structures, but arises instead in a relationship of inversion between the order of elements in the data structure and the order of elements in the operand obtained from it and used as an input to some later process. For example, if data items are read from left to right in a data list and used to form an operand in LIFO order, they will be treated in any following operation as if they had been read originally from right to left. In this way, a system allowing one-way data lists and both LIFO and normal ordering of operands may be functionally equivalent to one allowing two-way data lists. The pointer on a LIFO list always moves in the opposite direction while reading the list than it did in forming it. While LIFO, or "pushdown" lists
as they are often called, are most usually represented by chained storage structures and read with destructive reading action - to be discussed under "storage structures" below - these are not necessary characteristics of the LIFO order itself. LIFO data lists may just as easily be stored in conventional, sequentially-numbered addresses, and read non-destructively, if this suits the programmer's purpose in a given case, without going to the expense of setting up chained storage structures, available space control, etc., unless these are really needed.

**Data Cross-Referencing.** - Another way in which data can be arranged, in addition to being ordered or non-ordered within data structures, is by setting up relationships between different parts of an ordered data structure, or across different data structures. Data structures may be unrelated and dealt with independently, or they may be cross-referenced in several ways. When a data item has been cross-referenced to another, or a structure to another structure, a pointer set at the first entity will point also to the cross-referenced entity. Sometimes the cross-referenced entity may be obtained only through its primary reference in an ordered structure with a pointer, and sometimes the cross-referenced entity itself is a part of a structure and obtainable in other ways than through the reference. Data cross-referencing may be **collateral**, **hierarchical**, or **multiple**.

**Collateral Cross-Referencing.** - A pointer resting on a data item marks another item as well (for example, an "attribute," label, tag, synonym, etc.). A pointer moving along a list marks items in a parallel list also.

**Hierarchical Cross-Referencing.** - A pointer resting upon a data item points also to a related structure within the same larger structure of which they both form a part. The pointer moving through a structure with this type of cross-referencing may be regarded as following a list up to the referenced point, then jumping to another list, from which it may jump to still another, etc., so long as there are more hierarchical references in any list. Having reached the end of a list without encountering a reference, the pointer goes back to the place where it left off in the immediately preceding list, and continues in this manner until it either reaches the end of the first list in the structure, or is stopped for some other reason. In such data structures, lists may have elements which are themselves lists.

**Multiple Cross-Referencing.** - A data item may be on many different lists simultaneously, or a list may contain the same data item more than once.

Some examples of commonly used data structures that employ various combinations of the properties described above may prove helpful in clarifying the distinctions between them. A "tree," for example, is a structure made up of lists with hierarchical cross-referencing between the component lists; strictly speaking, trees should not have multiple cross-referencing, since this may result in more than one path converging upon a given item or in "cycles" or loops of items across lists in the structure. The term "tree," as used in information retrieval and similar applications of computers, often takes on a broader or looser meaning than the strict definition in mathematical usage (a directed graph with only a single path entering each "node" and without cycles). Many structures called "trees"
in the literature have more complex cross-referencing than the strict definition would permit. A "network" or "lattice" structure is one with both hierarchical and multiple cross-referencing, and the cycles and ambiguous choice situations that may arise in such structures must be dealt with and anticipated in some way by the problem solver.

STORAGE STRUCTURES - Concepts and Terminology

After the problem solver has developed various possible ways of solving his problem, in terms of data structures such as lists with their attached pointers and in terms of actions he wishes performed upon the items in the data structures, he must consider how best to create an image of the data structures in some computer memory. In dealing with data structures, he could be largely free of considerations connected with storage allocation, speed of access, etc. He needed only to define operationally and explicitly his goals, raw materials, and actions, as he would in describing the solution method to a person completely unfamiliar with the problem. Now, however, in choosing storage structures, he must consider the total volume and speed of access of computer memory, the provision of index registers or indirect addressing or chained input-output control, etc. He must consider, also, the size of his data items, the elementary particles he wishes to manipulate, with respect to the individual registers in the computer. These determinations may force him to revise his solution method and abandon some data structures that are too difficult or expensive to realize within the storage mechanisms of a given computer or programming language which he must use, and to fall back upon less sophisticated structures in some compromise solution. Nevertheless, the primary consideration in solving a problem, by machine or otherwise, is the problem itself and the purposes and goals of the human being who is solving the problem. The machine and programming language he uses are his tools, and they should not be allowed to dominate and distort the problem solution to the extent they frequently do when used unimaginatively or hastily, before the problem itself has been thoroughly and creatively analyzed and understood. The attempted machine solution of a complex problem is apt to end up as a solution of a totally different and grossly oversimplified or biased model of the problem, that proves inadequate as soon as an unanticiped or unnoticed source of variety and irregularity shows up in the data. It is suggested here that this type of inadequacy can be avoided more successfully if a thorough and open-minded pencil and paper analysis of a problem is carried out before the details of coding and formatting come to the fore.

PROPERTIES OF STORAGE STRUCTURES

Storage structures, as they will be described here, are not intended to be representations of particular computer hardware memory organizations, or exact memory maps of computer registers as organized by a particular software system. They are intended rather as generalized representations of the relationships of logical accessibility between data items that may be set up in a machine memory (e.g., a magnetic core memory) as it is organized by a software memory system or programming language. A "storage slot" is a cluster of entities which may or may not occupy a single machine
register, but all of which are accessible from other similar clusters
in a single step or cycle of the addressing mechanism of the software
system. The computer object program or programming system which will
implement the problem solution must be able to answer certain crucial
"questions" about those data items which are defined by the problem
solver and are to be represented in storage slots made up of computer
registers. It must know where to put any given data item in storage
and how to obtain it after it has been stored; this is described below
as address assignment. The program must know where to store and how to
get at the "neighbor" item of a given item on an ordered data list, or
the related items cross-referenced to a given item; these questions are
answered by the mode of address coupling in a system. The program must
know also what the act of reading or seeking a data item in a storage slot
will do to that item; will it leave the item in its cell or cells unchanged,
or erase it and render the storage slots available to contain some different
items to be stored later? These points are covered by the mode of reading
action of the storage system. Finally, the program must know how it can
find and recognize an "empty" or "free" slot in which to store a new data
item; this question is dealt with by the mode of writing action in the
storage system.

Address Assignment. - Address assignment is the way in which individual
data items must be obtained or stored in the computer; it may be arbitrary,
content-derived, or associative. By "arbitrary address assignment" will be
meant the single selection of the next available storage slot provided by
the writing action of the storage structure (see "writing action", below).
It may be noted in passing that the terms "associative" and "content-
addressed" seem apt to be interchanged or confused in the literature; per-
haps this confusion reflects an unrecognized divergence of interpretation
whereby some writers emphasize the "association" of an address with the
substantive content of the data element stored in it while others empha-
size the "association" between the substantive contents of two different
data elements, and still others emphasize the "association" between the
storage addresses in which two different data elements are stored. In this
paper "content-derived address assignment" will refer to the formation of
an address from the contents of the item itself; "associative address
assignment" will mean the finding or placing of the address of an item
within the storage slot containing another item. Thus, content-derived
address assignment as defined here means the first of the three alternative
interpretations described above, and associative address assignment refers
to the last of the three. The second alternative, the relationship between
two different data items, is dealt with under data structures, and the
term "associative" will not be used in this paper to refer to such "semantic"
or purposive relationships, since they are intangible, existing only in the
mind of the person who is building the data structure, and are bound up with
that mystery known as "meaning," for which we have as yet no satisfactory
operational definition. Essentially, only two different ordering relations
are dealt with in this discussion. For storage structures, physical or
logical adjacency in computer memory is the relation that defines order
among the component cells or slots. For data structures, the ordering
relation between data items is one of mutual accessibility or temporal
proximity in the course of some action upon the items. Either one of these
relations might serve, for present purposes, as an operational definition of "associations" between data items, but it seems preferable, due to the widely varying connotations the word has for persons with backgrounds in different disciplines, to restrict it here arbitrarily to one specific, operationally definable usage.

Address Coupling. — Address coupling is the way in which ordered data structures or cross-referenced data structures are obtained or stored in memory. Given a storage slot, obtained as a result of some previous action, the mode of address coupling in the storage system tells the program where the next slot to be obtained will be found. Address coupling may be additive, i.e., the program may form the successor address by adding 1 or some constant to the numerical address of the current slot it is scanning. This is the commonest practice in programming, and index registers or counters or "address modification" are the usual ways of carrying it out. Alternatively, address coupling may be chained, in which case the address of the successor slot is physically stored within the current storage slot; a storage structure whose addresses are coupled by chaining may also be regarded as a set of storage slots, each of which is addressed by "associative" address assignment. If the emphasis is upon the structure as a whole rather than the mode of obtaining individual addresses, the term "chained address coupling" will be used. An ordered list of data items represented in a chained storage structure must be scanned serially in such a way that the "ith" item may only be reached by successively obtaining the "i-1" predecessors one at a time. If the data items are too large to fit into single storage slots, some hybrid mixture of additive and chained address coupling must be employed, or cross-references must be stored by one or the other of the address assignment methods, etc. In an additively coupled storage structure, the "ith" address may be obtained by the single operation of adding "i" to the "base address" of the cell sequence; any cell in such a structure may be reached by a single additive operation from any other.

Reading Action. — The way in which stored material in a cell is changed or acted upon by the memory system when it scans a storage slot is the reading action of the system. It may be non-destructive, in which case the item is left in its cell unchanged, or destructive, in which case the item is "popped up" or erased or marked for later deletion, or marked as having been processed so that it cannot be processed again.

Writing Action. — Writing action is the way in which available or empty storage slots are found and recognized by the memory system when it has a new data item to store. It may be programmer-determined, i.e., the programmer has set aside named blocks of storage and clears and fills these in prearranged ways; he must keep track of all the storage allocation for the program in detail. He may allow the program a bit more flexibility of control over storage by placing some flag or marker in used cells, or conversely, in free cells, and by having the program test and manipulate these status flags as it uses and releases storage slots. This device makes possible executive available-space control, allowing the program to obtain or place available addresses on an "available space register," or to examine and manipulate their status flags entirely automatically as appropriate during processing.
AN EXAMPLE OF THE USE OF DATA STRUCTURES IN THE ANALYSIS OF A PROBLEM

In the following pages an artificial problem will be described, along with a number of different possible solution methods and descriptions of the data structures and operations employed to carry out the solutions. Nothing is said about storage structures, since these are somewhat easier to visualize and easier to understand. The problem used as an example is a quite simple series of operations for a human being, indeed almost a trivial problem; it is interesting, therefore, to note the large variety of possible solutions involving different ways of scanning the data and resulting in very different quantities and arrangements of intermediate results. The solutions are not all equally "good" or efficient, as may readily be seen, from the point of view of a human problem solver. A choice among them in a given case, however, would depend upon the storage structures available to the problem solver who must realize them in a computer or by means of a programming system; the choice also depends on the repertoire of operations or tools available in the computer order code or programming language (add, compare, extract, etc.).

THE "CHARACTER-DISTANCE" PROBLEM

Examine an input stream of n characters (letters of the alphabet only), and form a result stream of numeric values, one for each input character. Each position of the result stream will be occupied by a number representing a count of the characters separating the character in the corresponding input position from the nearest similar character to its left in the input stream. No distance larger than 14 will be recorded. Any character not matching anything to its left within 14 positions will have a zero in the result stream.

Sample: input AABCCDEEFFEAGHIJKLMNOPQRSTUVWXYZ
result 0100003001396000000000000000000002

The Solutions and Their Method of Presentation

Nine different solutions to the "character-distance" problem will be described below. These were obtained from three different persons of widely different backgrounds, during a time period or approximately one week. (It is evident that many more might be developed, and the reader is encouraged to find other solutions, either more economical to perform or more interesting and complex in their exploitation and recording of information in the data stream). Each solution will be presented first in a brief verbal description, together with a name or title characterizing it. Following the verbal description, a set of data items, structures, and processes will be suggested in accordance with the analysis provided by this paper. A detailed discussion of the exact steps of the solution in terms of these entities will not be included, since it would be extremely lengthy and tedious to read, and may be derived by the reader from the verbal description without too much difficulty. The analysis of data items, data structures, and processes was not a part of the original solutions provided by our informants but was added by the author; it has, however, been accepted by the originators of the solutions as representing their intentions with reasonable fidelity.
Certain general preliminary comments may help to clarify the analyses of the solutions. A pointer, as has been stated above, is attached to one or more ordered data lists; it may also define "collateral" or "parallel" positions on some other, "cross-referenced" data structure. A pointer always has a value or position or "setting" assigned to it with respect to its list or lists at any given time; this value may be regarded as an integer variable, available to the problem solver as a data item. The value of a pointer may have a constant, a variable such as a "counter" or another pointer value added to it; it may be assigned to another pointer; or it may be placed in the results list, or in a list cross-referenced to another list. By convention, the values assigned to pointers may be positive integers only; the left-most character of a data list may be considered as having the pointer-value "1"; ("zero" is also possible but was not used in the solutions below). If positive integer "n" is added to a pointer value, the pointer may be thought of as moving "n" positions to the right in its list or lists; if a negative value is added, (or a positive value subtracted), the pointer moves that many positions to the left; a value or zero or a negative value (depending upon whether the left-most position was given the value "1" or "0") for a pointer means that the pointer has run off the left end of its list. For values which will be used repeatedly in the course of the problem solution (either numeric or non-numeric), "constants" should be set up as data items; for example, if counting down by 1, or counting up by 3 are operations that will be performed repeatedly, constant data items consisting of the values "-1" and "3" should be provided. A "counter" is a variable data item to which some starting value is assigned (by moving some other data item into it), and which is then counted down or up by adding and subtracting a constant data item to or from it. Other "variables" besides counters are possible as data items: e.g., an item whose value is periodically reset or changed to match data items obtained at various points in a process. Batched data structures do not have pointers but are processed in some arbitrary order by taking the next unused item from an input batch and putting out an item to whatever is the next empty place in a result batch. Names of processes used in the analyses of the solutions are intended to suggest simple hand operations without implying any specific machine methods; "sort" means "rewrite items in some specified order;" "add" and "subtract" have their usual meanings. "Compare" refers to an arithmetic test (greater, equal, less), and "match" refers to a non-numerical, identity comparison or recognition of identical content. Finally, it should be stressed that all processes must have explicit operands in the form of data items or structures; an "add" process must have two operands, and they should be numeric, while a "match" process must have two non-numeric operands which are reasonably conformable for matching (i.e., a single character cannot be "matched" to a list of characters in a single operation).

Solution 1 - Left-Right Parallel Scan.

Fill the result stream with zeroes to start. Match pairs of input characters in the stream at a distance of 14 apart, moving through the stream from left to right. When there is a hit, store the number "14" in the result stream at the same position as the right-most character of the
pair in the input stream. Repeat for intervals of 13, 12, etc., down to 1, using a counter to keep track of the interval, and making a complete pass through the input stream for each interval.

Data Items - Input characters; result numbers; a counter D1 starting at 14 and counting down to 1 by decrements of 1; constants 1, 0, 14, n.

Data Structures - One-way input character list ILIST1; one-way result number list RLIST1; pointers PL1 and PR1 (left and right) on ILIST1; PR1 also defines parallel positions on RLIST1 by cross-referencing.

Processes - Match character pair pointed to by PL1 and PR1; add the constant 1 to both pointers simultaneously; subtract the constant 1 from the counter D1; compare value of PR1 against constant n each time PR1 is changed to see if the right end of the input has been reached on a scan; compare value of D1 to constant 1 to see if all intervals have been completed.

Solution 2 - Left-Right 1-14 Scan With Distance Counter.

Fill result stream with zeroes. Start a counter (for distances between characters) at 1; match pairs of characters with the left-most member chosen from the left-most end of the stream to start, and the right-most member at that position which is to the right of the first member by that number of places specified by the distance counter. While the same left member is used for the pair, the counter counts up by 1, selecting a right member one step farther to the right each time, until either a hit is found or else the distance counter builds up past 14. If a hit is found, the value in the distance counter at that time is placed in the result stream at the position corresponding to that of the right-most member of the matching pair in the input. When either the counter reaches a number greater than 14 (the right-most character of a pair would be off the end of the input stream) or a hit has been found, whichever happens first, select the next input character after the one just used as left member, reset the counter to 1, and repeat the matching process. Continue until every character in the stream except the right-most has been used as a left member in a pair; at this time the process is complete.

Data Items - Input characters; result numbers; a counter C2 starting at 1 and counting up to 14 by increments of 1; constants 0, 1, 14, n.

Data Structures - One-way input character list ILIST2; one-way result number list RLIST2; pointers PL2 and PR2 on ILIST2, with PR2 defining parallel positions on RLIST2 by cross-referencing. Pointer PR2 may be considered as moving along a one-way sub-list of ILIST2 for each position of PL2, instead of both pointers advancing through the list in parallel as for solution 1.

Processes - Match character pair at PL2, PR2, add constant 1 to PL2. Add constant 1 to PR2; add constant 1 to counter D2; compare value of PR2 to constant n each time PR2 changes, to see if it is running off the right end of the input list; compare value of D2 to 14 to see if one scan of PR2 is completed; compare value of PL2 to constant n to see if the solution is complete.
Solution 3 - Left-Right Alphabetic Selection.

Fill Result stream with zeroes. Make a pass left to right through the input stream, finding all cases of any particular letter of the alphabet, e.g., "A," recording their position numbers as they were encountered in the stream and counting the total number of cases for each letter. When all of one letter have been found, examine the list of position numbers by pairs from left to right, subtracting the left-most number from its neighbor to the right, the second from the third, etc., until all the entries have been done. After each subtraction, if the difference is 14 or less, place its value in the result stream at the position marked by the right-most of the position numbers subtracted. Repeat the scanning and subtraction process for each letter of the alphabet; when all letters are done, solution is complete.

Data Items - Input characters; result numbers; letters of the alphabet; counter for cases of each letter, CL3; constants C, 1, 14, n, 26; counter CA3 for letters of the alphabet.

Data Structures - One-way input character list ILIST3; one-way result number list RLST3; batch of letters of the alphabet, whose order among themselves does not matter; one-way letter-position list LLST3; pointer PL3 for ILIST3 and defining parallel positions on RLST3; pointers PLL3 and PLR3 for letter-position list LLST3.

Processes - Match character at PL3 to the letter of the alphabet chosen from the batch of letters; add constant 1 to PL3; add 1 to both pointers PLL3 and PLR3 at the same time; subtract position number at PL3 from that at PLR3; compare difference to the constant 14; compare values of PL3 to n as test for end of input stream during each letter scan; add 1 to counter CA3 for letters of the alphabet; compare value of CA3 to 26 to test for completion of all letter scans.

Solution 4 - Alphabetic Sort.

Make one pass through the input stream, counting the characters from left to right and attaching the count to each character as a position number. Sort the characters with their position numbers alphabetically on the character as major field and the position number as minor field. Make a pass through the sorted stream of characters, matching each character to the one to its right; if they are the same, subtract the position number attached to the left character from that attached to the right character and place differences of 14 or less in the result stream at the position specified by the position number for the right-most character. If the characters are different, place zero in the result stream at the right-most position number of the pair, and if the difference from the subtraction was greater than 14, do likewise. Continue scanning the sorted stream in this way until it is exhausted.

Data Items - Input characters; result numbers; position numbers; unspecified items required for the "sort" process; constants 0, 1, 14, n.

Data Structures - One-way input character list ILIST4, with pointer PL4 for generating position numbers; one-way position number list PLST4;
cross-referenced to ILIST4 by pointer PI4. (This means that the pointer PI4 is used both to generate a sequence of data objects as entries for the position number list PLIST4, and also to select items from that list by parallel cross-referencing to the input list ILIST4.) One-way sorted input character list SCLIST4 with two pointers PSL4 and PSR4 (right and left), which advance side by side; a one-way sorted position number list SPLIST4; pointers PSL4 and PSR4 define parallel positions in SCLIST4 and SPLIST4, selecting both a character from the former and a position number from the latter. The result of this cross-reference is to relate a pair of characters and a parallel pair of position numbers. A one-way list of result numbers, RLIST4, cross-referenced to the input character list ILIST4 by pointer PI4; this reference is represented indirectly by the position numbers which were generated as values of PI4, but are being obtained in their sorted order during the scan of the sorted list of characters.

Processes — Sort ILIST4 and its parallel list of position numbers, PLIST4; (the nature of this process is unspecified; its operand is a list, and some indication of the order desired must be provided. In an actual problem, it might be necessary to analyze the "sort" process in the same way as the "character-distance" problem is being analyzed here, in order to select the method of carrying it out that would be best suited to the entire solution). Add constant 1 to PI4 to generate position numbers for PLIST4; add 1 to PSL4 and PSR4 at the same time; match characters in SCLIST4 pointed to by PSL4 and PSR4; subtract position numbers on SPLIST4 cross-referenced to characters on SCLIST4; compare PI4 to n for end of position number generation, compare PSL4 to n for end of scan through sorted list SCLIST4; compare difference between position numbers to 14.

These rather indirectly and complexly related structures may be clarified by the following illustration, using the sample presented with the problem description:

| ILIST4 | A B C D E F F E A B G .... |
| PLIST4 | 1 2 3 4 5 6 7 8 9 10 11 12 13 .... |
| RLIST4 | 0 1 0 0 0 3 0 0 1 3 9 6 0 .... |
| SCLIST4 | A A A B B B B B C D E E F F .... |
| SPLIST4 | 1 2 1 3 6 1 2 3 0 3 2 4 5 7 1 0 8 9 .... |

When PI4 has the value 5, the character "D" is selected from ILIST4, and the position number "5" is attached to it at the corresponding position of the parallel list PLIST4. After this parallel pair of lists is sorted, they become SCLIST4 and SPLIST4, as illustrated. When PSL4 and PSR4 have the values 4 and 5 respectively, they select two neighboring "B's" on the sorted character list, SCLIST4; by parallel cross-referencing to the sorted position number list SPLIST4, they also select two position numbers, referring to positions on the original input character list, "3" and "6". These are
subtracted, the left from the right, to give the difference "3", which
is placed in the result stream at the position "6" attached to the right-
most "B" of the pair.

Solution 5 - Parallel Scan With Multi-Level Result Record.

Perform the left-right parallel scan of solution 1, but start with an
interval of 1 instead of 14. Instead of immediately placing the interval
in the result stream, form a sequence of the position numbers for all the
right members of matching pairs at a given interval in the order they were
encountered during the scan. Repeat with all intervals up to 14, forming
a new stream of position numbers for each interval. Relate these fourteen
streams in a three-level tree structure, with a "list of lists" containing
the names of the fourteen sublists in LIFO order (i.e., with the name of
the list for interval 14 to be read first and the name of the list for
interval 1 last). Count the entries in each list as they are formed, and
attach the total number of entries in each list to the list name for later
use in scanning the list. Start a counter at 14 and go through the lists
one at a time, taking a list name from the "list of lists" and placing
the value of the counter in the result stream at all the positions on the
list, then counting down the counter by 1 before selecting the next list
name. When the counter has reached zero (or passed 1), stop.

Data Items - Input characters, result numbers, counter D5 starting
at 14 and counting down to 1 by decrements of 1; counter H5 for keeping
track of the number of entries on each sublist, starting at 1 and counting
up by 1; constants 0, 1, 14, n.

Data Structure - One-way input character list ILIST5; one-way result
number list NLIST5; a "tree" structure made up of fourteen one-way lists
of position numbers, and a list of list-names for these in LIFO order,
LLIST5; a LIFO list of numbers telling how many entries are on the sublists,
NLIST5; pointers H5 and PR5 on ILIST5; PR5 defines positions in NLIST5
also by parallel cross-referencing; pointer PT1 on LLIST5, (the list of
lists in the tree), and pointer PT2 for any of the fourteen sublists in the
tree; PT1 also defines positions on NLIST5.

Processes - Same as solution 1 except for the following: add the
constant 1 to counter D5 (the counterpart of D1 in solution 1); compare
value of D5 to 14 to test for end of all the intervals; add 1 to counter
H5 to count entries in a sublist; add 1 to pointer PT1; add 1 to pointer
PT2; compare value of PT1 to constant 1 to test for end of list of lists
(a LIFO list is arbitrarily considered as stored left to right, and read
right to left); compare value of PT2 to the number obtained from NLIST5
(through the current position of PT1) to test for end of the sublist
currently being scanned in the tree.
Solution 6 - Right-Left 1-14 Scan.

Start at the right-most character of the input stream. Match it with each preceding character to its left, with a counter for the number of comparisons starting at 1 and counting up by 1. If there is a hit, place the value of the comparison counter in the result stream at the position of the right-most character of the pair. If the counter has built up past 14, place a zero in the result stream at the position of the right-most member of the last pair tested. In either case, select the next character to the right as right member for the next set of pairs. Continue until the left-most character of the input stream would be selected as the next right member of a pair, at which time the solution is complete.

Data Items - Input characters; result numbers; counter D6 starting at 1 and counting up by 1 to 14; constants 0, 1, 14, n.

Data Structures - One-way input character list ILIST6; one-way result number 1 list RLIST6; pointer PR6 on ILIST6, and defining parallel positions on RLIST6 by cross-referencing; pointer PL6 starting at PR6 minus 1, or one to the left of PR6, and moving a maximum of 14 positions left while PR6 remains in the same place.

Processes - Match character pair at PL6, PR6; subtract constant 1 from PR6; subtract 1 from PL6; compare value of PL6 to 1 to see if it is running off the left end of the stream; compare value of PR6 to see if all input characters have been matched; add 1 to counter D6; compare value of D6 to 14 to see if one scan of PL6 is completed.

Solution 7 - Left-Right 1-14 Scan Without Distance Counter.

Start at the left end of the input stream. Match pairs of characters selected as follows: a fixed right member paired consecutively with a maximum of fourteen characters to its left, unless the left member would be beyond the left end of the stream. If there is a hit, place the difference between their position numbers in the result stream corresponding to the right member of the pair. If the left member is beyond the end of the stream, or if the difference between the position numbers of the characters is greater than 14, place a zero in the result stream in the position corresponding to that of the right member currently selected. In any case, select a new right member farther to the right in the input stream and repeat the entire process until the right member would be chosen beyond the right end of the stream, when the solution is complete.

Data Items - Input characters; result numbers; constants 0, 1, 2, 14, n.

Data Structures - One-way input character list ILIST7; one-way result number 1 list RLIST7; pointers PL7, PR7 on ILIST7; PR7 defines parallel positions in RLIST7.
Processes - Subtract 1 from PR7 to set PL7; match characters at PL7 and PR7; subtract 1 from PL7 to get next left member; subtract PL7 from PR7; compare their difference to L4 to test for end of a scan; compare PL7 to zero to see if it is running off left end of stream; add 1 to PR7 to get next right member; compare PR7 to n for end of processing.

Solution 8 - Alphabetical Count.

Start a counter for counting input characters. Set up twenty-six counters for letters of the alphabet, in no particular order, but accessible in such a way that each letter of the alphabet has a unique and consistent counter. Take the next character from an unordered input batch in arbitrary order, and count the input character counter up by 1. Select the letter counter from the batch of twenty-six that is unique to the letter being scanned in the input. If that counter is zero, place zero in the result batch at the next empty position. If the letter counter is non-zero, subtract its value from the input character counter value. If their difference is more than L4, place zero in the result batch; otherwise place the difference itself in the result batch. In any case, place the current value of the input character counter into the letter counter just selected. Repeat the process with all characters of the input batch.

Data Items - Input characters; result numbers; letter counters, letters of the alphabet, counter C8 for input characters; constants 0, 1, L4, n.

Data Structures - Unordered batches for input and results; a one-way list of letter counters CLIST8; a one-way list of letters of the alphabet ALIST8; pointer P8 on ALIST8 and defining parallel positions of CLIST8.

Processes - Match input character to letter at P8 on ALIST8; add 1 to P8; compare letter counter obtained from CLIST8 to zero; add 1 to C8; compare C8 to n to test for end of input batch; subtract non-zero letter counter value from C8; compare difference to L4.

Solution 9 - Interval Count.

Set up a list of 26 letter counters unique to letters of the alphabet as for solution 8. Flag or mark all the counters with some special symbol, not zero or a positive number, indicating that they have not yet been selected or used. Get a character from the input batch; scan and test all the counters in the letter counter list, adding 1 to each counter that is not flagged with the special symbol (the first time, all will be flagged and none will be incremented; toward the end of the process, if all letters occur in the input, all may be incremented and nonflagged). Select a particular letter counter for the letter being scanned in the input as in solution 1; if it is flagged, place a zero in the result batch at the next empty position. If the counter is not flagged, test to see if it is over L4, and, if so, place zero in the result stream. If the letter counter was L4 or less, place its contents in the result stream. In any case, set the letter counter just selected to zero. Get another character from the input and repeat; continue until all input characters have been processed.
Data Items – Input characters; result numbers; letter counters; letters of the alphabet; constants 0, 1, 14; flag value, 26.

Data Structures – Unordered input and result batches; one-way list of letters of the alphabet ALIST9; one-way list of letter counters CLIST9; pointer PA9 on ALIST9; pointer PC9 on CLIST9; pointer PA9 also defines cross-referenced positions on CLIST9, as for solution 8, even though CLIST9 has its own independent pointer.

Processes – Match input character to letter at PA9 on ALIST9; add 1 to PA9; add 1 to PC9 to scan letter counters; compare PC9 to 26 to test for end of letter counter scan; match letter counter at PC9 on CLIST9 to flag value; add 1 to non-flaged letter counter; compare letter counter at PA9 (obtained by cross-reference to ALIST9) to 14.

EVALUATION OF THE SOLUTIONS

The evaluation of a group of alternative solutions to a problem depends largely upon the specific circumstances within which the problem solver must operate at a given time; an important factor in the decision to use one or another of the alternatives will be the relation of the problem to others which are being solved in conjunction with it. If the problem forms one step within the solution of a larger problem (as the "sort" process does within solution 4 for this problem above), then it will undoubtedly be worthwhile to adopt a solution method that saves and records more information about the data structures even though this may require more time to carry out. On the other hand, if no information need be saved as it is generated during the solution of the problem, and if later steps in a larger problem are independent of what goes on within this step, it is expedient to choose one of the "cheaper" methods that records nothing but its results and is quick to carry out.

The solutions described above for the "character-distance" problem fall into four general classes: 1) linear "1-14" scans; 2) linear parallel scans; 3) alphabetic sorting methods; 4) counting methods. Solutions 2, 6, and 7 are of the "linear" type; solutions 1 and 5, of the "linear parallel" type; solutions 3 and 4, of the "alphabetic sorting" type; and solutions 8 and 9, of the "counting" type. They will be compared briefly in the following paragraphs with respect to two criteria: 1) economy in the number of match operations required, and 2) amount of information saved.

COMPARISON OF THE SOLUTION METHODS

Number of Match Operations Required. – The parallel scans require n times 14 "match" operations, regardless of the number of hits in the input stream. The "1-14" scans require fewer match operations, the more hits there are, with a maximum of (n-1) x 14 for no hits at all, and a minimum of (n-1) matches for an input stream consisting of repetitions of a single letter. The operations required by the alphabetic sort depend on the sorting; (the "selection" method of solution 3 requires 26 x n operations regardless of the number of hits while a sorting method involving a left-right scan along the input stream, with interchanges of characters in "inverted" order with regard to the desired sorted order, would require no "minor" sort on the position numbers and would take relatively few operations). The two counting methods are probably at opposite extremes;
"alphabetic counting" is the cheapest and quickest method to carry out, especially since, in most computers, the "cross-reference" of letter to counter could be done by adding the binary number representing the character in the machine to a "basic address" of the counters, arranged in consecutive registers in storage. This trick would obviate the necessity for a match operation, a one-way list of alphabetic characters, or a pointer like P8 or P49. The "interval count" requires 26 x n "flag" matches as well as those needed to select the counter for a letter; again, there are tricks that make this less expensive in a computer than it is to do by hand (for instance, making the "flag" a minus value which can be recognized by a single machine instruction). The advantages provided by these tricks of the programming trade are, from the point of view adopted in this paper, fortuitous and cannot be assumed routinely as a part of the solution of a problem.

Amount of Information Saved for Later Processes. — While the problem used here as an example was presented in isolation and could be solved as if it stood alone, it is important to remember that any real problem of a degree of complexity sufficient to warrant analysis by these methods at all would consist of many steps. If the input stream is to be scanned, or if information about it is required for several steps in the total problem, it will be advantageous to save information in some orderly way so that the stream need be scanned only once. It is in these circumstances that an advantage may be gained from the apparently more expensive solution methods that exploit and record structural features of the input stream (e.g., linear scanning with tree structures of position numbers like solution 5, or formation of sorted streams with attached position numbers as in solution 4). The elegant and parsimonious "alphabetic count" of solution 8 leaves no trace of its operations on the data stream and no record of characteristics of that stream useful for a later step. Any step following such a process as solution 8 and operating on the same input stream or on the result stream from that earlier process must be carried out as though no earlier process had occurred.

WAYS OF DESCRIBING DATA STRUCTURES

Following the example of Iverson [13], Johnson [14], and others, data structures will be represented in this paper by diagrams similar to directed graphs. The terminology and approach employed will be quite different in many ways from those adopted by other writers, in that the emphasis here will be upon the graphic display and comparison of particular types of data structures embodied in certain programming languages and techniques, and not upon the development of a detailed formal model for such structures in general, as is the intention of most other writers on the subject. The interested reader is invited to examine the works of Iverson and Johnson if he has not already done so; the paper by Johnson is of considerable interest and somewhat less well known than Iverson's contributions. It should also be noted that the usage of the terms data structure and operand in this paper varies from the usage employed by these and other authors. Like so many terms encountered in the literature of machine information processing, these terms are nowhere clearly defined. Some authors apparently use the term "data structure" to refer to structural descriptions of input data which is to be acted upon by an object program being compiled by a compiler program (e.g., [11], p. 402). Others use "data structure" to mean what the present writer prefers to call file structure: the predetermined
linear arrangement of material in sequential external files, and typically in magnetic tape files. Iverson and Johnson use "operand structure" to mean much of what is discussed here under data structure.

The reader's attention should also be directed to a highly interesting and useful paper entitled "A Study of Methods for Representing Data Structure", and written by G. L. Brumm of the MITRE Corporation [22]. This report provides a detailed analysis of the problems involved in the representation of information structures in a computer memory. Of special interest and value is Brumm's description of the steps which a problem-solver may follow in determining a storage structure for use in the solution of a given problem. The examples and comparisons of various methods for representing information in storage are also very useful. Brumm appears to use the term "item" to mean an element of information, as "data item" is used here. His term "atom" appears to correspond roughly to the term "storage slot" as used by the present author, and to refer to a stored element, not further divisible, and accessible as a unit within a storage system.

Data structure in a general sense is the entire set of possible sequences of data items obtainable or desirable in a problem solution, where data items are all constants or variables employed in the solution, whether as "inputs", in the usual sense or as necessary auxiliary information required for some step. A particular data structure is one such set of relationships between data items. The complete set of data structures for a problem solution may include several different types of structure, each of which may be represented by many particular structures involving various grouping of particular data items. An operand, as the term will be used here, is a single actual sequence or selection of data items chosen from within a particular data structure. Examples of data structure in general are "one-way lists" and "hierarchically cross-referenced lists" (trees); numerous examples of particular data structures may be found in the analyses of the character-distance problem provided in Part I of this paper: e.g., "a one-way list of input characters"; an example of input operands for the "match" operation, which requires two, is "the character pointed at by the left pointer on the input list" and "the character pointed to by the right pointer on the input list."

New data structures (but not new types of data structure) may be freely constructed during the course of a problem solution by establishing cross-references between operands derived from previously existing structures. An individual operand, however, will not be considered to have any structure more complex than that of a single element or a simple, one way list of elements. The decision to define the term "operand" in this way was largely arbitrary and was made in order to simplify discussion by assigning all structural considerations of any interest to "data structures." (Although the hypothesis would be difficult to confirm, it seems highly likely that very few of the atomic processes applied to data by human beings in the individual steps of a problem solution deal with actual operands of any higher complexity than simple strings of symbols, however intricate the total process of deriving individual symbols may have been; the human span of immediate recall for perceptual objects such as verbal or visual symbols
and the human capacity for simultaneous attention to several distinct figural objects against a ground are quite limited.) *

If the restrictions on the complexity of structure to be allowed to an operand in this paper are borne in mind, the following description applied by Johnson ([1]) p. 5) to the formation of operands appears appropriate as an introduction to the subject here: "By its very nature a data process selects components from given structural operands, combines components to yield new components, and marshals the created components into new structural operands." Operands arise in a problem solution each time a selection and rearrangement is made among the elements accessible via some path through a data structure; they may be either inputs to or results from a step in the solution. Each step involves some single operation — for example, "match," "add," "compare," "test" — regarded as a unitary, atomic act that may be performed repeatedly during the solution of the problem. These atomic operations constitute the "order code" of the human problem solver for this problem. In a careful statement of the problem definition and solution, each such atomic operation should be attentively chosen and fully defined by the problem solver as regards the number and nature of its input and result operands as well as the exact nature of the operation itself. Operations of any degree of complexity may then be built up by combining atomic operations into "subroutines," or more complex units of action employed repeatedly in the solution, which may themselves form parts of still higher-level units. The result of the application of such a complex or compound operation will frequently be a new data structure, built up from the cross-referenced operands produced by its component atomic operations; an example of such a higher-level result structure is the tree of lists formed in solution five for the character-distance problem (See pp. 75-76).

Diagrams for Data Structures.

Particular data structures may be represented by sets of nodes connected by branches. The nodes may be either data nodes, capable of containing data items, or structural nodes, which are distinct from the data occupying data nodes, and constitute names or tags by which the nodes may be designated. Labels are shown in the diagrams above their nodes and enclosed in parentheses. Branches are shown as arrows connecting nodes, and indicate the adjacency or succession of one node to another.

* cf. G. A. Miller, E. Galanter, K. H. Pribram, Plans and the Structure of Behavior, Holt, New York, 1960, pp. 131-132: "The largest number of digits the average person can remember after one presentation is about seven, and if we want to be sure that he will never fail, we must reduce the number to four or five. Thus it is about four or five symbols (words, elements, items, lists, things, chunks, ideas, thoughts, etc.) that will easily group in consciousness at one time as a new list to which we can attach a new label."
Data nodes are represented by solid circles or circles containing the data characters, and structural nodes by open circles enclosing an "X". For purposes of graphic convenience, certain "directions" have been distinguished in which the arrows between nodes may point: these are up (U), down (D), left (L), right (R), and any number of "collateral" directions (C), which need not be separately named. Three axes or dimensions underlie these "directions": 1. the vertical axis representing hierarchical cross-references in the data structure; 2. the horizontal axis representing succession in a one-way or two-way list in the data structure; and 3. the collateral axis representing collateral cross-references to nodes whose primary place is elsewhere in the data structure or outside of it in some other structure. Collateral successors are often accessible only in limited ways or to a restricted set of operations. The arrows representing them in the diagrams are shown in dashed lines and may extend in any physical direction on the page.

The head of an arrow emanating from a node points to the right, left, up, down, or collateral successor, of that node. Some nodes are linked by double-headed arrows, signifying that they are equivalent nodes, mutually accessible in a single step along the axis involved. If no arrow emerges from a node in some direction allowed in the structure, that node is right terminal, left terminal, up terminal, down terminal, or collateral terminal, as the case may be. Many nodes may be terminal in several directions, and in some structures certain nodes may have no successors in any direction. In structures which allow more than one branch to emanate from a single node in the same direction, the individual branches may be distinguished by numbering them from left to right or downward, whichever is most applicable.

Tables for Data Structures.

Tables may be formed to supplement the diagrams and to provide a more exact and orderly, though somewhat less vivid, representation of particular data structures. These tables may be employed not only as pencil-and-paper aids but also as a foundation or sketch upon which storage structure maps may be built by the assignment of storage slot addresses to node labels; the tables including storage slot addresses may then be placed in the machine memory in adjacent registers and used to control addressing of data items in the computer memory. Johnson[1] and Iverson[3] provide thorough discussions of many tabular or matrix-like representations of data structure (or, as they prefer to call it, "operand structure"). Such data structure tables all contain certain columns of information: 1. a column of ascending row numbers used to name and order rows in the table (and also to assign sequential relative addresses to the storage slots containing successive rows of a table if it is to be used as a memory access control table in the computer); 2. a column of node labels; and 3. a column for data occupying data nodes. There may be a number of other columns in different tables, e.g., a column for the "degree" (number of down successors) of each node, and columns containing row numbers of successor nodes in various directions, most commonly down and right. The tables containing columns for successors are often called "chain-list matrices."
The node labels in the table may be arranged in either of two ways: a "left-list" or "right list" ordering ([14], pp. 8-10). The left list arrangement mentions the nodes of the structure in "subtree" order, i.e., starting at the extreme top left or right (if there is more than one root in the structure) and going downward to a down terminal node, then right (or left) to a terminal node in that direction, then "backing up" to the next unmentioned node on a previous level, and continuing in this manner until all nodes have been mentioned. The right list arrangement mentions the nodes in level order, i.e., starting at the root level (top) and listing all the root nodes in either direction, then descending to the next level and listing all nodes across the structure in the same direction, and continuing until all nodes on the lowest level have been mentioned.

The data structure tables used in this paper will present the following information: 1. the row number column, 2. the node label column, 3. the data column, 4. up successor column, 5. down successor column, 6. left successor column, 7. right successor column, 8. collateral successor column, and 9. a "comment" column for explanatory remarks. Node labels will always be mentioned in subtree order, unless level order is explicitly stated. Labels on the same level within a list will always be mentioned from left to right as they appear on the data structure diagram. The data column will contain a blank for any row standing for a node that contains no data. Successor columns will contain single row numbers or lists of row numbers (depending upon the system being described); these are the numbers of other rows in the table where the successor nodes are to be found. In tables stored in a computer, these successor row numbers serve as chained addresses to other rows of the table. If the system does not allow successors to a node in some direction, a dash will appear in the appropriate column for the row describing that node, and for a node which is terminal in some direction an asterisk will similarly appear. In some cases, where nodes have extensive sets of collateral successors, the labels of the successors may be found in the collateral column instead of row numbers, and they may be assumed to have rows in some other table, not shown in the example. Lists of multiple successors in a column are always read left to right.

WAYS OF DESCRIBING STORAGE STRUCTURES

As defined previously in this paper, storage structures are generalized representations of possible sequences of storage slots composed of registers in a high speed, internal computer memory. Each register of the machine memory has a numerical machine address, by which the built-in addressing mechanism of the computer may reach its contents and which may be treated in many ways like any other number in the machine. Each slot of a storage structure provided by a software memory system similarly has an "address," which may be a number or a name, and is the means whereby the addressing mechanism of the software system reaches the contents of the slot. Slot addresses frequently cannot be treated like numbers; the way in which they are utilized to get at their contents depends upon the memory system involved.
A storage slot is not necessarily or even usually a single machine register in the memory systems to be discussed here. The slot stands for a group of computer registers which are all accessible in a single step or cycle or the memory access mechanism in the software system, and which function as a unit in the system. Mere contiguity of machine registers is not enough to provide common membership in a single storage slot in any system. There must be, in addition, some explicit boundary convention, such as a base address and a length in consecutive registers, or some regularly expected sequence of functionally distinct subfields containing chained addresses of other computer registers. The system must deal with the problem of organizing computer registers into storage slots, and also with the problem of organizing storage slots into storage structures. The former problem pertains to the addressing mechanism of the system and cannot be covered adequately for most systems due to the lack of information provided by the authors concerning the details of storage usage in the computer for their techniques. The organization of storage slots into storage structures will be the primary concern of this discussion. The machine registers which function as a unitary storage slot in a system need not be contiguous in the machine memory. In the SLIP system, for example, some subfields of slots may be pairs of computer registers physically located in a set associated with the "reader" mechanism; other subfields are in special description ("header" and "alias") registers; and still others are in the data storage registers for data lists. Many systems employ flags or markers in machine registers to specify the nature of data, the function of the machine register in the storage slot, or its availability for use to store data. All such details will be considered to pertain to the addressing mechanism of the memory system and will not be discussed or described in detail or represented explicitly in the diagrams.

The address assignment and address coupling methods employed by a memory system, as defined previously in this paper, tell how data structures are entered or searched in storage slot addresses; the reading and writing actions of the system determine the way in which storage slots are made available for use or reuse in storing data. A given memory system usually defines more than one functionally different region within the machine memory, e.g., a raw data storage region, a common work area for general temporary storage of operands, a communication area or "scratchpad" region for recording certain operands for "immediate recall," and specially structured association areas or description lists for auxiliary information. The memory system often makes use of several different storage structures, each appropriate to one of the different regions distinguished in the total memory. Attention will be focused in this paper primarily upon structures in data storage and auxiliary information storage regions, but other regions will be mentioned where information regarding them is provided in the sources.

Diagrams for Storage Structures.

Storage structures will be represented by assemblages of blocks surrounded by rectangular outlines and interconnected by arrows. Each entity within a rectangular outline stands for a single storage slot, and single or double-headed arrows represent one-way or two-way accessibility.
of the slots among themselves. A wavy arrow stands for a special relation between slots whereby the successor slot is accessible only via certain specialized addressing mechanisms in the system, or is within a different functional "region" of memory than the slot of the main storage structure which names it as a successor. Each storage slot has a slot address, written above it in the diagram and encircled. For convenience and consistency, the slot addresses are shown as numbers, but they are to be regarded as names or tags rather than as machine register addresses upon which arithmetic may be performed to get other addresses. The attachment of consecutively numbered addresses to different storage slots in the diagram does not necessarily imply that the slots are located in neighboring machine registers, unless explicitly so stated in the discussion of the system. In most cases it has not been possible to ascertain from the source documents exactly how all computer storage registers are related to one another in the slots of a storage structure. Most authors are reluctant, for a variety of good reasons, to supply the complete concrete details of implementation on a computer in their memory systems. In some cases, the systems are presented only in the form of proposals or abstract sketches of a general method, so that even major considerations regarding storage structure are left to the reader's imagination.

The smaller vertical blocks within a storage slot outline in the diagram are intended to represent subfields where the memory access mechanism of the system expects to find data or addresses of other, successor slots in the storage structure. The uppermost small block is the "Item" (I), and contains either data or an X indicating that it is not expected to store data. It may be blank, signifying that the slot, while capable of storing data, does not happen to contain any at the time. Below the item subfield block are from one to five successor address blocks, depending upon the variety of successors allowed in the system. These have been given the same arbitrary "direction" names as were used for data structures: up (U), down (D), left (L), right (R), and collateral (C). A key is placed near the uppermost storage slot rectangle in each diagram to show the order and nature of the subfields for that system. The successor address subfields may contain either the address of another slot, or an asterisk indicating that the slot whose subfield it occupies has no successor in the direction concerned. In some systems there may be multiple successors in a subfield; such lists are represented by a succession of small blocks on a horizontal axis, perpendicular to the main axis of the slot rectangle. The individual addresses in each list may be considered as numbered from left to right, e.g., for a system having three down successor addresses, these might be designated "D(1), D(2), D(3)." The directions carry no implications with regard to relative physical positions in the computer memory, but they do bear a relationship to the directions in the data structure being represented in the storage structure, as may be seen from a comparison of corresponding diagrams.
Tables for Storage Structures.

Storage structure tables are possible, bearing the same relation to storage structure diagrams as data structure tables do to their diagrams. These will not be discussed further in this paper, as they differ from data structure tables only in that the entries are storage slot labels rather than data item labels. Storage structure tables are likely to be useful as means of describing a storage structure for a program intended to simulate that structure within a different storage structure. They might also aid in the comparison of several different storage structures for some purpose.

THE USE OF DATA STRUCTURES AND STORAGE STRUCTURES IN DESCRIBING SOFTWARE MEMORY SYSTEMS

In the following pages, twelve memory systems will be described; these have been selected from among a large number of proposed or experimentally implemented systems as best representing the scope and nature of the majority of such techniques. A brief verbal characterization will be provided for each system, along with a sketch of the types of data structure it permits and the way these are reflected in storage structures. Following the description, a data structure diagram, one or more data structure tables, and a storage structure diagram will be shown. These are intended to display particular data structures and storage structures that are typical of the facilities in the system, and to exemplify its salient features as fully as possible. The structures used as examples have been reproduced or adapted from the source articles describing the systems. The source of each example is noted under the data structure diagram, and the example is represented in all diagrams and tables for any one system. In some cases, tables or lists used by the author of the source reference to display the example have been reproduced, sometimes in slightly modified form; in other cases, data definition statements or other instructions which serve to set up the structure in a programming language are shown. These lists and statements will be included in the figure containing the data structure diagram. In Fig. 1 may be found a complete key to all symbols used in diagrams and tables.

It should be emphasized that the descriptions of memory systems presented here are not intended to provide full details regarding the systems, or to serve as anything more than introductory sketches or samplings of their facilities. The aim in these descriptions is twofold: first, to exemplify a method of analysis and an approach, and second, to provide a comparative glimpse at a broad sample of programmed memory structures which seem likely to be of special interest or utility in the field of information retrieval and natural language processing. Many details have been glossed over or omitted entirely in the descriptions, diagrams, and tables, and in some cases the interpretation placed by the present writer upon statements in the source articles may well have been faulty. The reader is urged to examine the source references for those languages or systems in which he is interested; they are, with very few exceptions, brief articles in recent issues of the ACM Communications and should be easily accessible to most readers.
Data Structures and Storage Structures in Twelve Recently Developed Programming Languages and Memory Systems.

1. The IPL Programming System. [1], [2]

The well-known "IPL" (Information Processing Language) series of programming systems, through which the "list-processing" concept was first rendered familiar to the computer user and programmer, should require little introduction to readers of this paper. Since 1956, articles have been appearing in considerable number on the subject of IPL and the theoretical and technical considerations underlying its development; IPL has also been widely used to compile object programs in a broad variety of fields of application, notably in artificial intelligence research and in the behavioral sciences, and is available on a variety of computers. Lists in IPL may be data lists or description lists, the description lists being accessible as such only to special instructions and operations in the system. In addition to data lists and description lists, IPL has a special set of "pushdown" communication registers or working registers, accessible via special system mechanisms, in which operands may be temporarily stored and manipulated, and which may be read or loaded to produce operands in LIFO order. (In fact, the authors of IPL are probably the originators of the wide interest at present in LIFO ordering ("pushdown lists"), though individual programmers had undoubtedly made use of the techniques involved before the advent of IPL.) Finally, the system has a memory region known as the "available space list," which is employed for address assignment in writing, and to which vacated addresses are returned for re-use in destructive reading. The working registers and available space list will not be represented in the diagrams and table in Fig. 2, which shows only a data list structure and description list.

All data list elements contain a "symbol" and a "link" field; the symbol may consist of data or of a chained address pertaining to a down successor. Each data list may have only one "description list," which is intended to function as a collateral or auxiliary information appended to the data list, and contains a string of "attributes." Each attribute has a single "value" collaterally appended to it and accessible via special mechanisms from it. The attributes appended to a data node may only be obtained via the special collateral reference from that node, and the value for an attribute may only be obtained via its attribute node. The structure of pushdown lists and the available space list is probably similar to that of data lists, but the system uses them in special ways.

Data Structures. - Lists are one-way in all memory regions. Operands may be formed from lists and stored or obtained from the push-down registers in LIFO order. Cross-referencing is hierarchical or collateral and may be multiple as well, since items may appear repeatedly within a single list or may be referred to by many different lists (note the cyclic reference to list head "L2" in the example, both as root of the entire structure and as the terminal right successor on one of its own sublists). The data lists in IPL have a type of data structure

27
involving only down, right, and collateral successors, which will be called a "double chain with collateral branches." This type of structure, made familiar in IPL has been used repeatedly in other systems developed subsequently, and will be encountered below in the descriptions of LISP, COMIT, and the Frywys-Grey "Multi-List" system. A closely related structure, the "double chain" alone without collateral branches, has been widely employed in memory systems, some of which are discussed under section 5 below. The description list hanging from node "9-1" in the diagram (Fig. 1a.) has a more limited structure.

**Storage Structures.** Address assignment is arbitrary for heads of system lists (e.g., the available space list, communication registers, working registers); these addresses are, from the programmer's point of view, supplied to him as names having an *a priori* storage slot address and computer register address in the system. All successor addresses are assigned associatively, i.e., their slot addresses are obtained from the available space list, which constitutes a list of all vacant storage slots at any given time. Successor slots containing elements of data lists are linked to one another by chained address coupling, through the slot address stored in the "link" portion of the list slot. Collateral (description list) successors are placed in special locations which distinguish them from other successors. There are also special conventions regarding types of symbols, reflecting a tripartite division of computer memory into regions for the purposes of the IPL addressing mechanism ("internal", "regional", and "local"); the system employs this distinction to aid in distinguishing between hierarchical references (down successors) and right successors. Reading action in IPL may be destructive (returning slot addresses to the available space list) or nondestructive. Writing action is accomplished by executive available space control through the available space list.

2. **The LISP Programming System.** [3], [4]

The LISP (List Processor) Language for the IBM 704 computer was developed at MIT in 1958 in conjunction with artificial intelligence research. The "list structures" employed in LISP are, as stated by the author of the system ([3], p. 184), closely similar to those of IPL. The data structures provided include the double chain with collateral branches described for IPL, and a specialized "association list" structure probably like that of IPL description lists. There is a "free space list," a "public pushdown list," and "temporary storage registers" which may be set up within the pushdown list; these are similar to the available space list and working or communication registers of IPL. The author of the reference used as the principal source of information about LISP in this discussion [3] was primarily interested in the formal mathematical properties of LISP rather than in its practical use or actual mode of functioning in a computer, and details of data structures and storage structures are much less clearly presented than was the case for IPL. The data structures allowed by LISP do not appear to differ from those of IPL in any major
respect for purposes of this paper, and accordingly LISP will not be analyzed further here. The language does provide, however, a very extensive and useful set of instructions for operating upon data in the structures, (for example, a Boolean logical operation set, a facility for defining special arithmetic operations similar to "macro-instructions," and a set of operations involving truth values and predicates). These operations render LISP of great interest and value for many applications in artificial intelligence research. Readers interested in LISP should consult(4) which is a complete programmer's manual and contains much explanatory material as well as an example of a complete LISP program.

3. The COMIT Programming System:[5],[6]

COMIT was developed at MIT in, approximately, 1959 to aid in research on the machine translation of language. It is, essentially, a coding language for structural linguistic analysis of a single sentence of natural language text. Its originators have, however, come to feel that it is likely to be of general utility as a symbol-manipulation language, for example, in the field of information storage and retrieval (5). This claim will be discussed further below. COMIT provides a single data manipulation area called the "workspace," containing a single composite structure (which may become quite long) called a "workspace expression," and made up of a one-way list of characters or words called "constituents," each of which may have a specialized collateral structure appended to it and made up of a "numeric subscript" and a set of "logical subscripts." There is a special memory structure called the "list" or "dictionary" which is employed for rapid searching and lookup of specially prepared data sorted automatically into alphabetical order by the system. Operands obtained from the workspace may be stored and retrieved, but not searched or altered, on a set of temporary storage registers called "shelves." Material may be stored either in LIFO or normal order on shelves but is read from them only from left to right and reinserted into specified positions moving from left to right in the workspace. The COMIT instructions (somewhat confusingly called "grammar rules" by the authors of the system) search the workspace from left to right to find given, predetermined patterns of characters, and transform the structure in the workspace by deleting, rearranging, and inserting constituents specified with reference to the recognized patterns. The "numeric subscript" attached to a constituent may be obtained and manipulated only via a special mechanism, and it is subject only to certain numeric operations. There may be any number of "logical subscripts" appended to a constituent, and each of these may have several "values;" these form a two-level hierarchical structure which is, however, only accessible via special mechanisms and to a limited set of non-numeric operations. The workspace structure, while reminiscent of the "double chain with collateral branches" encountered in IPL and LISP, is limited, restricted of access, and incomplete in comparison with the structures allowed by those languages.

COMIT appears to be highly useful in the applications for which it was developed (as a coding language for linguistic problems), but in the opinion of the present writer the wider claims of COMIT's originators...
do not stand up well under careful scrutiny. While COMIT includes many ingenious and useful features, the single level of structure permitted for constituents in the workspace - the only place where data may be scanned in natural order and transformed - is quite insufficient for many problems. It allows only molecular and highly specific transformations to be carried out within single sentences or phrases, and in rather rigid ways. Additional disadvantages are the heavy dependence upon exact, literal specifications of character patterns to be found in the workspace, and the inflexibility of the "constituent" concept which allows the workspace to be organized either in terms of single characters or in terms of words, but not of characters within words, and words within the sentence as a whole, and sentences within larger messages. (A rather awkward pair of operations called "expand" and "compress" are provided to translate parts of the workspace from a word-based constituent structure to a character-based structure and vice versa.) It is extremely unfortunate that the authors of COMIT did not provide it with a fully-developed double-chain structure allowing several levels of reference within the workspace, and "constituents" made up of lower-level constituents - a structure which comes much closer, in the present writer's opinion, to the multi-level structure of natural language as it is usually regarded.

A newer string-manipulation language called SNOBOL, similar in its basic conceptual framework to COMIT but providing a more powerful, hierarchical workspace structure, will be discussed as the eleventh memory system to be described in this paper.

Data Structures. - There is a single one-way list in the workspace, and single one-way lists may be stored on each shelf, in either LIFO or normal order. By clever manipulation of shelves and workspace, something like hierarchical cross-referencing might be achieved in certain operations of the system. No hierarchical cross-references are provided from constituents in the workspace to other constituents in that area, however. Material may be added to the right end of the workspace or put in front of its left end, or inserted into any constituent position within it numbered from left to right. Constituents may not, however, be read from right to left in any list. Hierarchical references are possible only for "logical subscripts." The asymmetrical and truncated possibilities for successors in various parts of the workspace structure may be seen in the data structure table, Fig. 4b.

Storage Structures. - Little or no information is provided by the inventors of COMIT regarding storage structures, even in the programmer's reference manual. One might conjecture, however, that address assignment is arbitrary for heads of major system regions (workspace, shelves, dictionaries), and associative for successors; there is a "free storage control" facility which probably uses a "free storage list" as a source of vacant storage slot addresses, at least for operations in the "workspace." Chained address coupling is the most likely form of address coupling for right successors in lists. Reading action apparently may be either
destructive or nondestructive in both shelves and workspace. Writing action is accomplished via the "free storage control," which keeps track of vacated storage addresses and provides the programmer, on request, with information regarding the number of registers he has used or left unused.


The Multi-List System was described in papers published during 1958 and 1959 by the Moore School of Electrical Engineering of the University of Pennsylvania. The proposed memory system was intended to be applied to MOBIDIC core memory modules and to be employed on military problems. Experiments, whose exact scope and nature are not clear in the sources, were made using a large military personnel file as a data base. The system is, apparently, a special purpose memory system, intended specifically for information storage and retrieval applications involving hierarchically classified files. There are two principal regions for data structure: a "tree structure" of lists, and an "association structure" containing many small multiply-cross-referenced data lists, each having a standard sequence of element types. The data lists in the association structure are cross-referenced to one another and also referred to collaboratively by the nodes on the lowest level of the tree structure. The system is intended to include a "scratch-pad" or working storage area for operand storage, although the reference article gives no further details regarding this region. In general, data is classified ahead of time by human beings and assigned to various categories which have descriptors; the classified data is loaded into the tree and association structures in prearranged ways and then searched; little is said about altering the structure of categories and descriptors in the machine, although this may be possible. This system appears to be a tool for searching and operating upon files previously structured in detail by human beings, rather than a tool for eliciting and forming structures under program control in flexible and dynamic ways.

Data Structure. - In spite of the very different external appearance of this system from those described previously, the data structure it provides in the "tree" region is the familiar "double chain with collateral branches." Lists in the tree structure are one-way, and hierarchical cross-references are allowed for all levels of the tree (except the lowest); collateral references may be found on only its lowest level, and the collateral successors are in the external association area. There is only one level of hierarchical reference, but all branches are "two-way." Multiple cross-references are frequent in this structure; another list in the association area may be referenced repeatedly by elements in one list (e.g., the double mention of "L22" by elements of "L17" in the diagram, Fig. 5a.), and one list may be referenced by many other lists (e.g., "L15", which is mentioned by both "L16" and "L17").

Storage Structure. - Each slot in the structure is a collection of machine registers called "catenas;" these are subfields of the slot which the system obtains either by additive address coupling or, more probably, by hardware arrangements of some kind. The storage structure diagram may
be seen to be somewhat different from the data structure diagram, having one less level in many places; this is a result of the way in which the "catena" registers were utilized to store "descriptors" as data, each linked to a catena in the same slot containing a second descriptor appearing on a lower level in the data structure (e.g., "AGE: 20"). Address assignment is arbitrary for initial slots of major regions, such as the "List of Lists" that forms the root of the tree structure. All other addresses appear to be assigned associatively, and address coupling for all successor slots is chained. Each association area list in the example occupies a single slot containing four "catenas." (The system allows up to ten catenas in a slot for an "average" data record.) Reading action is nondestructive, and writing action apparently is programmer determined. No details are provided regarding the "scratch-pad" memory.

5. Doubly Chained Trees and Filial-Heir Chains, [8], [9], [10], [11], [12], [13], [14], [15]

Under this general heading are included a number of memory techniques; the "doubly chained tree" structure described by E. H. Sussenguth [8] has been selected for discussion here as a typical example. Others are the "symbol tree" of Scidmore and Weinberg [9] and various "chain list matrices" for memory organization [10], [11], [12], [13], [14], [15]. Iverson [13] refers to these at "filial-heir chains," and Johnson [14] as "Sib-path tree lists." All of the structures involve down successors and right successors only, without collateral branches. In doubly chained trees whose nodes are occupied by data, it is frequently assumed that a "match" operation is to be performed between data items in some one-way input or "argument" list and the data items at each node in the tree to be selected along some allowed path. In this case, a successful match causes the "down" successor of the node to be selected for the next match, while an unsuccessful match forces selection of the "right" successor. A problem arises in these systems when the initial portion of an argument happens to match nodes in the tree up to a "down-terminal" node or exit point; failure to rule out this possibility would permit incomplete arguments to be accepted as present in the memory. In some cases, this might well be desirable, as when roots of words are being sought, but in other cases it might give rise to erroneous results. The problem involved here presents several important considerations which warrant fuller discussion.

In the opinion of the present writer, a memory system which is organized into hierarchical structures to be searched for matches to an input or argument string should provide three explicit and distinct modes of operation, which may be called 1. "source control," 2. "comparand control," and 3. "source and comparand control." In source control, the argument would be accepted as a match to some stored operand obtained from the data list structure only if each item matched in the two up to the last item in the argument; it would ignore "left-over" items in the tree structure. In comparand control the reverse would be required; matching items in both until a down-terminal node is reached in the tree, regardless of "left-overs" in the argument. In source and comparand control, the end of the argument
string must coincide with the end of the operand in the tree for a match. 
(The author's own "TEMAC" system, described in section 8 of this paper, 
provides all three modes of search and applies a similar distinction 
between "source" and "target" control to other data manipulation processes 
as well.) The method adopted by Sussenguth for solving the problem of 
Incomplete argument strings which may be erroneously accepted is to 
require that an "end-of-argument" symbol be matched with a similar "end-
of-memory-list" symbol stored at the lowest node of the tree along some 
path, forcing the system to operate only in the "source and comparand" 
control mode as described above. The extra match operation and additional 
level of nodes and branches required by this expedient appear somewhat 
wasteful; a better solution might be to provide a count of the number 
of data items in the argument list, perhaps as a collateral reference 
available to the addressing mechanism of the system. This count may 
then be used to reject any terminating paths in the tree that have 
encountered fewer "down" successor nodes than the count requires. In 
the example used for Fig. 6, however, the practice of providing an 
explicit end-character has been followed, as was the case in the source 
reference; for this purpose, the character "\&" has been used, to agree 
with the example for the TRIE memory to be discussed in section 6.

It is possible in doubly-chained tree structures as in other 
memory structures, to provide either positive recognition of individual 
argument strings being sought in the memory, or mere acceptance of 
strings as being present in the memory. Both modes of operation are 
useful under different circumstances, but of the two, the positive 
acceptance seems to provide the more interesting problems. The examples 
given by Sussenguth for the doubly-chained tree were so arranged that if 
the exit nodes from all paths representing strings of data items recognizable 
in the structure were chained to a common exit, "acceptance" would be possible; 
if each node at the lowest level of the tree were chained to some individual 
location outside the structure, "recognition" would be the result. The 
example chosen for Fig. 6 did not contain explicit information regarding 
action taken by the system on reaching a terminal node, so individual 
exits have been supplied to permit individual recognition of the nine 
words stored in the doubly-chained tree (nodes "LEI" to "LE9"). The 
actual example in Fig. 6 was chosen from Fredkin's article on the TRIE 
memory to facilitate comparisons between that system, discussed in 
section 6 below, and the doubly-chained tree structure.

Data Structures. - Lists are one-way, and cross-referencing is hier-
archical; no mention is made of any other memory structures in the 
system besides the tree structure itself. (It is perfectly possible, 
however, to imagine a system including working registers, pushdown 
lists, or other regions for operand storage combined with the data 
storage area organized as a doubly-chained tree).

Storage Structures. - Assignment of all addresses besides the base 
address of the entire "tree" is probably associative, via an "available 
space control register" of some sort that "remembers" which storage slots 
are available for use. This may take the form of a single slot containing 
the next unused address in a bank of sequential registers. Slots are 
linked to right successors by chained address coupling; for down successors,
additive coupling may be used, so that no subfield need be physically occupied by a chained "down" address. (It may be seen that the data structure provided by the doubly-chained tree might be realized in other storage structures besides the one suggested by Sussenguth, which assumes each node of the data structure to be stored in a single machine register and thereby permits the use of additive address coupling for "down" successors and the reduction of "available space control" to the simple recording of the highest numbered register used up to any given time.) Reading action is nondestructive, and writing action makes use of a highly simplified form of available space control.

6. The TRIE Memory System. [8], [9], [16]

The TRIE memory devised by E. Fredkin [16] is a close relative of the doubly-chained tree system previously described. It differs from the doubly-chained tree essentially in that, instead of providing branches only to nodes actually occupied by data items expected to match some input argument items, it includes all possible branches from any node that was occupied, and might be called a "reduced" tree structure in comparison with the "minimal" tree structure of the doubly-chained tree, standing midway between the latter and a "complete" tree structure including all possible nodes at every level, each node having all possible branches ([8], p. 278). Comparisons of the TRIE with other tree-like systems are provided in ([8], pp. 278-279), and in ([9], p. 31), and quantitative analyses of the comparative speed and economy of various tree structures are presented in [8] and [16]. The TRIE is made up of a "portal" or common entry "register" and a set of successor "registers," where the term "register" has a meaning identical with that of the term "storage slot" in this paper, rather than the meaning of "computer register." Each such "register" has a subfield for every possible distinct element that might occur in an input argument; for example, if the input arguments were strings of characters drawn from the English alphabet, terminated by a special "end-of-argument" character, every "register" of the TRIE would require twenty-seven subfields. In the example shown in Fig. 7, six subfields are utilized, accommodating the characters A, B, C, D, E, and the special end-character "V". The TRIE subfields do not contain any actual data items but instead are occupied either by chained down-successor addresses for other "registers" or by a special "null" address that provides a common exit to a "no-match" or "non-acceptance" node outside of the TRIE.

The subfields of the TRIE "register" are arranged in a predetermined order, (e.g., alphabetically by character), or else they are selected by content-derived address assignment. The presence of a chained down-successor address in the appropriate subfield of the "register" being examined constitutes a match, or acceptance of the input data item. In the example in Fig. 7, distinct terminal nodes "EL" to "FG" have been provided for "recognition" of the nine different words that may be found in the structure as operands, while in the original example as shown in Fredkin's article all down-terminal nodes were linked in common to the same "portal register" that was used as a common entry, thus providing
only "acceptance" or argument strings as present in the memory. In
the data structure diagram, Fig. 7a, the branches to nodes not
occupied on the next level are suggested by short lines emanating
from the parent node, to avoid confusing and cluttering the diagram.
In Fredkin's table, reproduced in modified form in Fig. 7a, the
disadvantage of using an end-of-argument character instead of a count
of characters to terminate matching becomes apparent. The number of
different words consisting of letters A, B, C, D, E in any sequence that
may be stored in the TRIE of Fig. 7 before more registers must be added
is restricted to 16 (the number of registers present in addition to the
"portal register"), since each register has space for only one end-
character subfield. Since words may end in any character, the registers
of the TRIE are not being utilized as efficiently as they might be. By
the use of a "count" for argument data items to be sought, this
inefficiency might be avoided.

Data Structures. - Lists may be one-way or two-way in various
kinds of searches that might be made in the TRIE (only one search,
the character-matching of "down" successors against input characters
strings, is discussed by the system's author, but other ways of utilizing
the structure may easily be imagined). Cross-referencing is hierarchical
only.

Storage Structures. - The assignment of a storage slot address to
the portal "register" is arbitrary (i.e., from the system's point of
view, the "portal" address is given). The successor slot addresses
for other registers are probably assigned sequentially in blocks
of "k" successive computer registers, where "k" is the number of
subfields required for a single TRIE "register". Within each register,
the assignment of computer cells to subfields is apparently content-
derived, or based on some prearranged ordering that is uniform throughout
the TRIE. Address coupling to down successors is chained, via the sub-
fields of the register. Reading action is nondestructive, and writing
action is probably programmer-determined; (since it is apparently a simple
matter of adding a constant to the last-used machine address to get another
block of "k" cells, and testing to be certain that the total number of
machine cells allowed for the TRIE has not been exceeded, writing action
does not require an available space control facility).

7. The TEMAC Programming System [17]

The TEMAC (Text Macro Compiler) system is a problem-oriented language
for text scanning and editing, developed for experimental purposes during
early 1961 by the author of this paper. It ran on a 704 computer from
July 1961 to August 1962, producing various successful object programs
and an expanded and improved version of itself, TEMAC II. TEMAC II ran
until June 1963, when the 704 was no longer available; the object programs
continued to run in simulated form on a 7090 and 7094. In January 1966,
a new and greatly expanded version of TEMAC went into operation on the 7094,
and is now successfully running under the IBDJOB monitor, producing and
executing many object programs. TEMAC provides three meaningfully distinct
higher levels of reference in data structures for language text, in addition to the level of atomic characters, and three corresponding levels of reference in data structures for other kinds of data (e.g., numeric values or symbolic names). The three levels in language structures are called "paragraphs," "sentences," and "words," and those in non-language structures are called "master-blocks," "cell-lists," and "cells."

The BRAID system, to be described below, provides a somewhat similar organization of data items into meaningful hierarchical levels, although the organization is based upon entirely different technique and philosophy. The levels of structure in TEMAC and BRAID have a different meaning than do the levels in "list-processing" languages like IPL, LISP, and SLIP. This difference reflects a special property of natural language text and similarly organized material - a property which the present writer calls the "meaningful ordering" of data items. In IPL and similar systems, anything may stand on a list, and lists may contain any element; there may be any number of hierarchical levels of reference, and any item may be found at any level in a structure. In a system based upon natural language, there are qualitative distinctions between levels (character, word, sentence, paragraph), and the levels must be occupied only by elements of the appropriate type; the resulting structure is not a general symbol-manipulation structure by means of which widely different aims may be realized, but a special-purpose, problem-oriented structure for a certain class of problems. The "level-order" arrangement of the data structure table has been shown for both TEMAC and BRAID in figures 8c and 9c; those relationships between data items most likely to be useful in problem-solving are displayed most effectively in such a table for systems of this kind.

A frequent use of the non-textual structures in TEMAC is to act as collateral cross-references to the multi-level textual structure, i.e., to contain weights, tags, codes, descriptors, or special lists called "pointer sets" made up of reference numbers standing for elements at certain positions within other structures. The non-textual data structures of TEMAC are, however, accessible entirely independently of the textual ones, and neither type of structure is subsidiary to the other, offering a decided advantage over the limited "description list" or "logical subscript" apparatus of COMIT. For both textual and non-textual structures, any number of structural descriptions, or "aliases," may be set up to provide various data structures for the same stream of raw data, and the structural characteristics of a stream may be investigated and manipulated as a problem-solving tool. Any structure may be cross-referenced on any level to the corresponding level of any other structure, of either type in the entire memory for a given problem, through a general cross-referencing mechanism based upon the relative positions of elements within structures on the next higher level, scanning from left to right in those structures.

The example in figure 8 shows a three-level textual structure "P," consisting of a "paragraph" of natural language text, which contains three "sentences" ("S1", "S2", "S3"), each of which in turn contains a number of "words" ("W1", "W2", etc.). The sentences and words may all be obtained either directly by name, or by selecting the appropriate
component from a larger structure; the paragraph "Fm" may, of course, only be obtained by name. Only the first sentence structure has been shown in detail. Individual characters or arbitrary sequences of characters may be obtained either by referring to the "word" names via a special "word-related character-string" operand, or else by referring directly to the data area in which the characters are stored (this latter structure has been omitted from the example). The TEMAC system includes some other memory regions with different structures, and alternative structures for textual data (e.g., one in terms of single characters scanned left to right), but these will not be described here.

**Data Structures.** - Lists in the textual and non-textual structures are two-way. Cross-referencing may be collateral, hierarchical, or multiple, but there are no "up" successors anywhere in the structures. In TEMAC, data structure is rendered more fully independent both of external file structure for data as "read in" to the computer, and of internal storage in computer registers than is the case in most other memory systems.

**Storage Structures.** - Address assignment is arbitrary for heads of major lists or memory regions, and associative for successors on lists. In textual structures, both "down" successors and "right" successors are linked by chained address coupling via structural description tables that function like "chain list matrices" in storage. In non-textual structures, whose atomic elements are single computer registers, only "down" successors are chained, and "right" successors are stored in neighboring computer registers so as to be obtainable by additive address coupling. Reading action is nondestructive, and writing action is programmer determined.

Storage is organized by TEMAC into four main areas: the "system subroutine" area, the "compiled program" area, the "descriptor" area, and the "data" area. The descriptor area contains tables providing access to the data objects (paragraphs, sentences, master-blocks, etc.). There are five different descriptor tables: the paragraph table, the sentence table, the word table, the master-block table, and the call-list table. Each of these tables has a reserved name which may be referred to by the programmer in the same way as any other program symbol, and by which he may directly access table entries, either by machine instructions or by TEMAC language statements, in addition to the "built-in" automatic references to them provided by the system. Only as much space is allotted to the different tables as is needed for any given compiled program. A "paragraph-descriptor" contains two parts: the address of the first register of a group on the sentence-descriptor table, and the total number of sentence-descriptors in the group. A "sentence-descriptor" contains the address of the first register of a "word-descriptor" group in the word table, and the total number of entries in the group. A "word-descriptor" has three parts: the address of the cell within the data area which contains the beginning of the word, the character position (1 to 6) occupied by the left-most character of the word within that cell, and the length of the word in characters. The text itself is stored in the data area exactly as it was read or moved in; packed six BCD characters to a register.
to occupy an irreducible minimum of space; the text always remains unaltered unless explicitly rewritten by the programmer.

Another programming language for text-handling, developed at the University of Pittsburgh and called PENLOPE, will be described as the twelfth memory system analyzed in this paper. PENLOPE bears a startlingly and highly suggestive similarity to TEMAC, and, as will be shown, makes use of an almost identical method of representing data structures in storage. The independent choice of this method and the kind of structure it provides to the user by two different compiler designers, without any knowledge of the other's efforts, appears to be a good indication that this structure is an appropriate and useful one for practical text-handling problems.

8. The BRAID Memory System. [18]

The BRAID (Bidirectional Reference Array Internally Derived) system was developed in 1963 and tested on the IBM 7090 by M. R. Seidel of the Datatrol Corporation. It provides a single homogeneous data structure preserving ordered relationships between elements on several levels, e.g., characters within words, words within phrases or sentences, or indexing terms within clusters applied to different documents. The storage structure is so arranged as to make maximum use of redundancy in the data at all levels, and to provide highly symmetrical relationships between storage slots in all "directions". The examples provided in the paper which formed the primary source of information for this description were obviously chosen to demonstrate the economy provided by BRAID for highly repetitive material, notably the one reproduced in Fig. 9: the phrase "to be or not to be." The BRAID system is intended to be capable of creating a multi-level structure based on frequently repeated character groups and word-groups automatically from a "raw" stream of text; no information is provided, however, regarding the scope and methodology of the techniques by which this scanning is accomplished. The system is oriented primarily toward applications involving storage and retrieval of text or text-like material. In some ways, as mentioned above, the data structure provided by BRAID is similar to that provided by TEMAC; BRAID, however, is a memory loading and searching scheme aimed at the compact storage of text, whereas TEMAC is a programming language with a wide range of instructions and facilities, based on a rather different scheme for storage and searching of text-like material, and providing the means of exploiting and extending multi-level data structure relationships simultaneously over several other types of data objects besides text. TEMAC is well adapted for editing of text as well as for searching and storing it. An additional difference may be found in the importance of repeated phrases and character groups in BRAID, which stores these only once, while TEMAC records the natural character-word-sentence-paragraph structure of the text as it stands without regard to repetitions within it.
Data Structures. - All lists are two-way, and hierarchical cross-referencing is also two-way, so that paths may be taken up, down, left, or right from any node in the structure. In the data structure diagram of Fig. 9a, the letters T, O, B, E, R, N are the only data items stored at any node, and are at the down-terminal nodes of the structure; all nodes above these are structural junction points only. For purposes of clarity, the diagram has been drawn with "subtrees" referred to more than once and represented repeatedly in their entirety. In the actual data structure, as may be seen from the data structure tables (Figs. 9b and 9c), multiple cross-referencing permits each substructure to be represented once only and its root to be referred to repeatedly on lists of successors on the appropriate level of the structure. Because the levels have so important a meaning in the BRAID memory, a data structure table in "level order" is included in addition to that in "subtree order."

Storage Structures. - Address assignment is arbitrary for initial slots of lists and associative for successors. Address coupling is additive for left or right successors and is chained for up or down successors. The successively numbered slot addresses in the example (Fig. 9d), which are reproduced from the source, are evidently intended to imply physical contiguity of the slots in storage, and probably to imply their being contained also within single computer registers, so that additive address coupling may be used to provide many of the symmetries in the data structure. Content-derived address assignment may be used for data items consisting of single letters of the alphabet or small numbers, but this is not explicitly stated to be the case in the source. Reading action is probably nondestructive and writing action probably programmer-determined, but little information is provided in the source regarding either.


SLIP (Symmetric List Processor) was developed by J. Weizenbaum during 1962 and 1963 from his own earlier KLS (Knotted List System) [23] and made use of concepts introduced by IPL, Perlis' threaded list system to be described below, and the "FLPL" (FORTRAN List Processing Language) devised by R. Gelernter [21]. SLIP, like FLPL, is embedded in the FORTRAN algebraic programming system; it is available on the 360 and 1604 computers. The system includes an "available space list," "public pushdown lists" for working storage, data list structures, special list-tracing mechanisms called "readers," and structural descriptions called "headers" and "aliases" that function in the hierarchical linking of lists. Each list head may have a description list similar to the description lists of IPL; the description list is accessible as such only via special operations, and may contain any number of "attributes," each one of which may have a single "value" appended to it. SLIP seems to hold out the promise of greater convenience, at least for non-numerical purposes, than any of the other "list-processing" languages (IPL, LISP, threaded lists, COMIT). Its one major defect, in the opinion of the present
writer, is the fact that it was implemented by forcibly wedging a list processor (as a set of subroutines) into the midst of a problem-oriented language, the FORTRAN algebraic compiler system. It would have seemed more logical to embed an algebraic compiler providing the means of operating on numeric data within a general symbol-manipulation language like SLIP, rather than to follow the opposite procedure. The practical reasons for the course of action followed by Weizenbaum may be imagined (e.g., the wide familiarity of FORTRAN among computer users and its wide availability; the historical and economic primacy of numerical over non-numerical uses of computers; the apparent technical difficulties involved in attempting to combine any other facility with FORTRAN on any other terms but those of FORTRAN itself; and last but not least, the apparent fact that Weizenbaum is himself a mathematician to whom mathematical problems are of central interest). Nevertheless, from the point of view of a user primarily concerned with processing language text, and only marginally, if at all, concerned with numerical problems, the requirement of operating within an unneeded and inappropriate mathematical language in order to use SLIP for symbol manipulation is an annoying inconvenience.

Data Structures. - Lists are two-way and may be read and stored from either their right or left ends. Hierarchical references are provided to the leftmost ("top") and rightmost ("bottom") elements of each list only via special structural list name nodes (e.g., nodes "L1", "L4", "L6", and "L42" in Fig.10a). Multiple cross-references are possible.

Storage Structures. - Address assignment is arbitrary for the head of the "available space list" and associative for all other storage slots. Address coupling is chained, using "left link" and "right link" subfields for left and right successors, and obtaining down and up successors through the "reader" mechanism and the "headers" and "aliases". The "reader" memory structures are themselves "pushdown stacks" which record the history of movement through the storage structure storing a given data structure whose name is in a "header" and "alias" reference, and which keep track of hierarchical references as they are encountered. Reading action may be destructive or nondestructive, and writing action is by executive available space control.

10. Threaded List Structures. [20]

The threaded list structures were developed as an addition to the IPL list structures by A. J. Perlis of the Carnegie Institute of Technology, Pittsburgh, Pennsylvania, during the course of work for the Office of Naval Research and the U. S. Army Signal Corps. The contribution of "threading" the lists consists mainly in providing the rightmost or "tail" element of each list in the IPL structure with an "up" branch returning to its parent node on the level above. The article from which the example in Fig. 11 was drawn provides very little information about the practical workings of the programming system. In addition to the data structure diagram (which seems quite straightforward and adds little to the doubly-chained tree that forms the basis of the IPL list structure),
the table provided by Perlis to describe this simple structure in his article is reproduced in Fig. 11a, and the descriptions of fields within this table given by Perlis have been collected, rearranged, and rephrased as the present writer understands them. The reader is invited to compare the explanatory material accompanying Perlis' table on page 196 of the reference; it appears somewhat indirect in its style of expression and easily able to be misunderstood. It serves as an excellent example of the obstacles interposed in the path of prospective users of special memory systems - i.e., those persons whose intentions in reading such an article in the literature are primarily pragmatic and not formal or theoretical. The additional obstacles such an "explanation" as that on p. 196 of the article places before the prospective user who is also a non-mathematician may well be imagined.

Data Structures. - Lists are one-way only, and "down" hierarchical reference are possible from structural nodes as in IPL. "Up" references are allowed only from the right-terminal node on a list to its parent node on the next higher level. Collateral references similar to the "description lists" of IPL are presumably allowed, but no mention of these is included in the source article.

Storage Structures. - The "table" provided by Perlis is apparently intended to function as a chain list matrix in storage, and the flag "f" is used to govern interpretation of the chained addresses by the addressing mechanism of the system. Aside from the "flag," there is little explicitly stated in the article about storage structure which differs in any important way from the storage structure of IPL.

11. The SNOBOL Programming System. [24], [25]

SNOBOL was developed at the Bell Telephone Laboratories in 1962. It was based to a considerable extent upon concepts embodied in the COMIT language described above in Section 3, but provides a more powerful and flexible means of handling character-strings. Operands consisting of word-strings and sentence-strings may be built up in an implicit "workspace" area which may be entirely re-built (from the user's point of view, at least) by each instruction that is executed. Word-strings may be manipulated within this area by scanning "rules" (instructions) analogous to those which COMIT provided for simple character-strings. A set of auxiliary "functions" and "predicates" allows access to numerical items and useful information about strings in the workspace, such as their "sizes". SNOBOL does not make use of any explicit auxiliary mechanisms such as the "shelves" of COMIT, nor does it employ any collateral structures like the COMIT "logical subscripts" and "numeric subscripts". The effects of these COMIT facilities are all obtained in SNOBOL more simply, as a result of the multi-level structure of the workspace itself, and by the action of hidden mechanisms implementing the scanning and data-handling activities of the system.

The data structure provided by the SNOBOL workspace appears much more regular and flexible than that of COMIT. The only item collaterally cross-referenced to data items in the workspace is the "size" or string
length value, which is obtainable by special operations. Numeric
and textual data may be manipulated with no apparent distinction from
the user's point of view. SNOBOL appears to remedy most of the defects
of COMIT, and to provide a means of handling textual strings which many
users have found highly satisfactory. Some programmers, however, have
considerable difficulty in learning to state and solve their own
problems in SNOBOL, as well as in COMIT, even though they have solved
pre-arranged "toy" problems with little difficulty in training classes.
Several programmers of the author's acquaintance who have attempted to
use SNOBOL have commented on the difficulty of learning to think in the
special fashion that SNOBOL requires. In the opinion of the present
author, a language like PENELOPE, to be described in the following
section, is a much more comfortable and natural vehicle for textual
problem-solving where real-life problems are involved. This preference,
while admittedly subjective and personal, is based upon important
characteristics of the languages and the problems upon which they are
to be used; an attempt will be made below, after PENELOPE has been
described, to compare SNOBOL and COMIT, on the one hand, against TEMAC
and PENELOPE on the other, and to analyze more fully some of the features
of what appears to be a fundamental difference in approach to text-handling
problems.

Data Structures. - There is a single one-way list in the workspace,
whose elements may have one-way sublists hierarchically cross-referenced
to them. A "size" value is collaterally cross-referenced to each named
element, and is obtainable by a special operation. The ends of the main
list and its sublists are immediately accessible, as are their beginnings,
so that elements or strings may be appended after strings. If something
is to be inserted or deleted in the middle of a list, however, it can apparently
only be done as a result of some scanning operation in the "left-half" of
a particular SNOBOL "rule", which locates a substring having certain
literal features, breaks the list into two at that point, performs the
insertion or deletion, then again rejoins the broken parts. In other words,
from the user's point of view, an explicit, literal scanning operation
must be performed in order to find any non-first or non-last element in
the workspace. It should also be pointed out that SNOBOL apparently
permits words and sentences to be obtained and manipulated within a larger
character stream only so long as they, or their surrounding elements, can
be inserted or called for as literal character strings. The SNOBOL workspace
still consists, essentially, of a single one-way list of characters, as was
the case in COMIT.

Storage Structures. - No information whatsoever is supplied, in the
published description of SNOBOL, regarding the actual mapping of data
structures into computer storage registers. The present author has
attempted to investigate SNOBOL storage structures by making a careful
study of a machine listing of the SNOBOL system in its MAMOS version
(under the University of Maryland 7094 operating system). This study was
a matter of considerable difficulty, since there are no explicit data
declaration or definition routines which might help to clarify the methods
used for storing data. The general outlines can be discerned, however,
and will be sketched briefly below. Address assignment is predominantly associative. Address coupling is chained, and reading action may be destructive or nondestructive. Writing action is by executive space control, which periodically collects "garbage" (i.e., finds and releases storage blocks for re-use when they are known to be no longer needed.

12. The PENNELOPE Programming System. [26]

PENNELOPE was developed for the IBM 7070 in 1963 at the University of Pittsburgh. Its primary purpose is the handling of natural language text, and it appears to be, like TEMAC, a true problem-oriented language for text-handling rather than a generalized string-manipulator or symbol-manipulator. PENNELOPE resembles TEMAC closely in many ways, and a general comparison of the two languages will be presented below, with special reference to the way in which both differ from COMIT and SNOBOL. Four different kinds of data items are allowed by PENNELOPE: "integer", "real", "Boolean", and "alphanumeric". There is apparently a three-level data structure for the integer, real, and Boolean items, and a four-level structure ("sentence-list", "word-list", "word", "character") for the alphanumeric items. Once a decision has been made concerning which are to be word-boundary and which are to be word-forming characters in the alphabet, any text word is accessible via a correctly written operand involving a data object name and a "component selector" or index expression. The boundary characters are also available through the use of a special form of component selector, as if they were word-like objects collaterally cross-referenced to the primary text words. These word-like boundary objects, called the "left punctuation set" and "right punctuation set" of a text word, may be obtained as wholes, or any individual character within them may be obtained alone. Component selectors point to individual components, and more complex operands are built up by "do-indexing" or "for-statements" which iterate the statement containing the operand while stepping the component selector over its string.

Some features which PENNELOPE and TEMAC have in common, and which present a contrast to the basic philosophy of COMIT and SNOBOL, will be discussed below. In the opinion of the present author, PENNELOPE is a much more natural and convenient language in which to solve text-handling problems than either SNOBOL or COMIT. The form of PENNELOPE, which it shares with TEMAC, with verb-like operation words and noun-phrase-like operand expressions related by a syntax resembling that of natural language, seems very helpful to a person whose approach to problem-solving is primarily verbal and discursive rather than mathematical or abstract. It seems far easier to keep track of the problem-solving steps when there are verbs and operands to serve as mnemonic aids. PENNELOPE, like TEMAC, provides facilities for making declarations of structure and content for data objects, mapping them out ahead of time so that components may be selected from them by operand expressions in program statements later on. This predefinition facility provides an advantage for some persons over the approach used in COMIT and SNOBOL, where literal patterns are spun out of thin air, as it were, with no previous establishment of structure, and where little or no structural information is saved from one "rule" to the next.
The facility provided by PENLOPE and TEMAC for specifying and manipulating data items embedded in the structural environment where they occur, and preserving all their relationships with surrounding structures, offers a great advantage in problem solution. The relative position of an element within a structure, and its relation to elements at corresponding positions in other structures, can provide a very powerful problem-solving tool; it is, at the very least, a convenient and generally useful mnemonic aid to the programmer in his efforts to find his way through complex structures being dynamically transformed in complicated ways. In COMIT and SNOBOL, by contrast, much reliance is placed upon the insertion of artificial markers into the data stream to flag sub-strings singled out for further attention; these languages also rely heavily upon the extraction of strings out of their context. A list-internal element can apparently only be operated upon in these languages by scanning through the workspace for some literal cue in the left-half of a matching "rule", and the main, if not the only ways in which the programmer may act upon strings is by rewriting them or moving them or parts of them; the data itself must be altered, or all relationship to the original context where it occurred must be lost. A person who is accustomed to using indexing expressions to select any desired sub-structure within a data object independently of its literal contents, and to record, without necessarily moving or changing it, the position of a desired sub-structure singled out for later attention, may find the constant artificial re-scanning and rewriting of data forced upon the programmer by COMIT and SNOBOL awkward and exasperating.

In addition to the above, data-structure features held in common by PENLOPE and TEMAC, both languages provide a very similar storage structure for representing data structures. This storage structure involves a very simple and powerful hybrid chained-additive address assignment mechanism, which is capable of economically providing direct, one-step access to an optimally large number of components on all levels of the data structure, with a minimum drain on memory space. Both PENLOPE and TEMAC provide textual (alphameric) variables and constants, and both emphasize structure and structural information explicitly separated from literal content of data, as a problem-solving tool accessible to the programmer. SNOBOL and COMIT, on the other hand, emphasize literal content and leave structure inside the "black box", out of the user's reach. Finally, both PENLOPE and TEMAC permit and, indeed, encourage the programmer to exercise a fine control over the way in which operations will be performed; (TEMAC apparently goes rather farther than PENLOPE in this regard, in that it permits 7094 machine instructions and other IBMAP assembler statements to be inter-mixed with TEMAC compiler-level statements at will, and routinely lists all system material, including all data object storage structure tables, as a part of the compiled program, so that they are directly accessible to the programmer). This readiness to invite the user to look "under the cover", and to tinker with any of the system's inner mechanisms at his own risk whenever he wishes to do so, is strongly at variance with the "closed cover" philosophy of COMIT and SNOBOL.
TEMAC and PENELOPE differ in a few relatively minor ways. TEMAC allows complex operand structures ("sets", "sequences", "left-boundary operands", etc.) in addition to single component selection, and provides access to characters via a special type of operand structure, the "character-stream" operand. PENELOPE allows more levels of indexing (up to three for a "sentence-string") and arithmetic expressions within indexing expressions, and addresses individual characters by the same mechanism as higher-level objects. TEMAC allows any number of "alphabet" definitions, each of which involves, among other features, the setting aside of a different "boundary character" selection for word boundaries, sentence boundaries, and paragraph boundaries; the alphabet containing the desired sets of boundary characters is then called upon in a special "segment" statement which forms the data structure in a predefined, "dummy" paragraph or sentence variable; any number of different "segmentations", or word-structures, may be made concurrently available for the same data stream by calling upon different alphabet names and variable names. The boundary characters themselves may be obtained by means of a "word-related character-stream operand". In PENELOPE, "segmenting" is not an explicitly controllable process, but is done automatically while reading or moving data; boundary characters may be specified by the programmer, and the specification may be changed at will in the program. Boundary characters in the data may be obtained as if they were words before and after a given word of the text, as described above.

Data Structures.-In PENELOPE, lists are two way; cross-referencing may be collateral, hierarchical, or multiple; all successor possibilities are allowed with the apparent exception of "up" successors. The data structure is highly similar to that provided by TEMAC (described above in section 7), aside from a slight difference in the handling of the boundary characters between text words.

Storage Structures. - Address assignment is arbitrary for heads of major memory regions and associative for successors within these. "Down" successors are linked by chained address coupling via special "pointer" tables to be described more fully below. "Right" successors are stored in neighboring computer registers so as to be accessible by additive address coupling, taking advantage of the simple "address modification" facilities available in the hardware and machine instructions. Reading action is essentially nondestructive, and writing action is by a simple and economical form of available space control. The PENELOPE programming manual includes an excellent, detailed explanation of the storage structures employed by the system, and some highly interesting comments on the advantages and disadvantages of the structures [26, pp. 97ff]. The author of the manual is indeed to be commended on his readiness to provide this information, which offers an invaluable aid to the prospective user in making the best possible use of the system, over and above its great intrinsic interest. The following description, based upon this material, may be compared to the information regarding TEMAC storage structures in section 7.
Storage in PENELOPE is organized into five major areas: a "system subroutine" area, "compiled program" area, "text bin", "pointer" area, and "variable heads" area. Text is stored in the text bin area in a semi-packed form, with each text word left-justified within as many adjacent computer registers as it requires (five characters per register), and "punctuation sets" or boundary character groups similarly arranged in separate registers. The registers holding the words and boundary groups are kept in the same relative order as the material originally displayed before unpacking. The pointer area contains groups of word-pointers, each word-pointer containing the starting address of the register or registers occupied by the relevant text chunk in the text bin. There are also sentence-pointers, which contain the first and last address of a group of adjacent word-pointers providing access to the words and punctuation sets for the sentence in the proper order. Finally, there are "variable heads" in the variable head area, which provide a pointer to the highest level of structure for each named data object; if the object is a single integer variable, the value itself is stored in the variable head, and no further pointers are needed. The different types of pointers in the pointer area are distinguished by flags within their registers, rather than by storage in different sub-regions as in TEMAC. There is a "free list" which keeps track of available space by a simple chained addressing mechanism linking chunks of adjacent registers.

The variable head, pointer, and text bin areas have slightly different sets of successor possibilities, as may be seen from the storage structure diagram (Fig. 12c). A data object has, apparently, only one name at its highest level, which provides access to the entire structure via a single variable-head slot. This slot may contain data (for a single value) or two "down" successors - the first and last address of a series of adjacent slots in the pointer area. In the pointer area, left and right successors and one or two down successors are provided; a word-pointer also provides access to two cross-referenced "punctuation sets" by additive addressing within the text bin, and contains a cross-referenced value telling the character-length of the text word. In the text bin area, left-justified chunks of text are arranged in consecutive registers with registers within the chunk and registers storing left and right "punctuation sets" accessible via additive addressing.

Some Tentative Conclusions Regarding Various Systems.

The twelve systems described and analyzed above fall into the following five major classes: (1) IRL, LISP, SLIP, and the Perlis threaded list structure may be called generalized symbol-processing languages; (2) COBOL and SNOBOL may be called string-manipulation languages; (3) TEMAC and PENELOPE may be called problem-oriented languages for text-processing; (4) The BRAID and Multi-List systems are specialized memory systems for information storage and retrieval (BRAID for natural language text and the Multi-List system for hierarchically categorized files); and (5) doubly-chained trees and the TRIE memory system are generalized memory systems. A "memory system" may be distinguished from a programming language in that, while the memory system involves only a way of organizing data in memory, the programming language...
involves a memory structure and also an extensive set of operations or instructions that operate upon the data within the memory structure and exchange information with the external world as well. It would be essentially meaningless to attempt any comparisons between languages in different classes (e.g., a symbol–processor over against a problem-oriented text processor) in the absence of any particular problem or set of circumstances to provide a frame of reference. The field of interest which first lead the present author to make the investigations described in this paper was that of text processing, especially within the context of information storage and retrieval problems. The comparisons between software systems to be presented below were all made with reference to this field of application.

The "memory systems" can be very helpful in certain cases, but they seem somewhat limited in scope, and require a rather rigid recasting of problem structures into specific arrangements in memory. It seems likely that a programming language, general enough in its field of applicability to handle a large set of problems, is preferable both to a memory system alone, such as BRAID or TRIE, and to a narrowly specialized programming language as well. It would appear desirable to have a language which on the one hand, provides a data structure permitting the manipulation of text in natural units, such as words, sentences and paragraphs, and on the other hand, makes an optimum number of data items directly and individually accessible on several levels of structure, without requiring long sequences of searches through predecessor elements. A hybrid storage structure of the type utilized by TEMAC and PENEOPE appears to provide direct access to many useful areas in data structures, as well as permitting very economical use of space in storing text and time in handling it in the machine. SNOBOL, while presenting some disadvantages to some users as suggested above, also appears to be a very useful language for many text-handling applications. COMIT appears excessively limited in the data structures it allows and overly specialized in its applications (despite the claims of wider applicability made by its inventors). Neither COMIT nor SNOBOL seems to have the full complement of closely controllable textual data definition and manipulation facilities required for the language-handling problems of interest to us. Both COMIT and SNOBOL share the disadvantages, apparently endemic to systems based on chained addressing and available space control, of making excessively high demands on computer memory space for storing text. In SNOBOL, for instance, according to the statistics provided by the interpreter for a program run made by the author as a test, eighty-one "strings" had to be stored, occupying one-hundred and sixteen registers, and requiring in addition three hundred "reference assignment" registers for a total of four hundred and sixteen registers, simply to create and access a structure involving one paragraph containing two short sentences, and four additional individual short sentences. These objects would require, in TEMAC, only a total of seventy-two registers, thirty for storage of text and forty-seven for word, sentence, and paragraph descriptors. In PENEOPE, a total of one hundred and thirty-one registers would probably be needed: eighty-four for text and forty-seven for "pointers" and "variable heads".

It should be pointed out that, of SNOBOL, COMIT, TEMAC and PENEOPE, none allows "up" successors anywhere in its data structure; it might well be fruitful to design and experiment with a data structure similar to that of TEMAC or PENEOPE but including a provision for "backing up" from sub-strings into higher-level strings, as is possible in the SLIP "list-processor".
This facility might pay for the extra expense it would entail by permitting more efficient searches of complex structures in storage. The symbol-processing languages and in particular IPL and SLIP, seem to provide convenient access to the structure of data objects under the programmer's control, and powerful means of manipulating complex data structures with relative ease. Unfortunately, however, these systems all appear to lack the means of dealing with text in natural units; they are best suited for handling single, brief names (words of five or six characters at most), or with numeric values. They are weak in their ability to describe and handle textual data, in contrast to their power in describing structure. COMIT and SNOBOL, while much more convenient in their data handling facilities, and employing flexible "list" structures for implementation of their operations "behind the scenes", do not allow the user explicit access to data structure as a controllable analytic tool. Finally, it is by no means certain that the chained, rigidly sequential one-way operand structures favored by "list" languages are really best suited for text-processing problems. We have seen earlier that these structures force the programmer to resort to an excessive amount of rewriting and re-scanning of data streams in order to preserve and regain information about relationships among elements. Relative position of elements within structures appears to be an inherently important feature of text, broadly useful for describing and analyzing textual material at all levels of ordering - character, word, and sentence. Information about the relationship of an element to surrounding elements, and its place in the structure of which it forms a part, is frequently lost in the destructive reading ("popping up" and "pushing down") and chained addressing methods favored by the symbol-processors in order to facilitate rapid insertion and deletion of elements.

In conclusion, it seems essential that further investigation and experiments of a practical nature be undertaken to develop a widely applicable and convenient problem-oriented language for text-handling. The present author feels strongly that more effort should be directed towards the study of human text-processing and problem-solving activities, with a view to emulating these activities efficiently on computers. An ever-increasing variety of valuable investigations are in progress in the behavioral sciences which can be of considerable aid to the computer scientist in this field. There is a great need for closer cooperation and more active communication between behavioral scientists and computer scientists; structural linguists and information specialists in the Library sciences should also cooperate in the research. All of these specialists are looking at different pieces of the problem, in the light of widely different basic assumptions, and at times duplicating the work of others in other specialties due to their narrow view of their problems.

It is greatly to be hoped that this vitally necessary research and experimentation, centered around the nature of text-handling problems and the operations and data groupings most useful for problem solution, will not be slighted as a result of the current trend towards universal "algorithmic" computer languages. The understandable but, in the author's view mistaken, efforts to press ALGOL upon programmers as a universal language for the solution of all problems, may result in the diversion of skilled data systems analysts and programmers from the important work of understanding the
problems themselves so as to state and solve them more effectively. There is a very real danger that natural human text-handling methods will now be artificially forced into conformity with the way of looking at problems required by ALGOL, just as they have, in the past, been forced into other highly unnatural and contorted channels by the requirements of machine languages and generalized "data processing" or file processing procedures. We need to know a great deal more about human text-processing activities and requirements before we try to fit them into any universal "algorithmic" language.
REFERENCES


**Figure 1a: Symbols in Data Structure Diagrams**

- **U, D, L, R, C**
  - Columns for up, down, left, right, collateral successors
  - Structure allows no data, or no successors

- *** (L)**
  - Node is terminal
  - Node labeled "L", not in table, is successor

**Figure 1b: Symbols in Data Structure Tables**

- Address "5"
  - Subfields
  - List of elements in subfield

- Accessibility in one cycle of addressing in software memory system

**Figure 1c: Symbols in Storage Structure Diagrams**

- Slot does not store data
- Data "A" in slot
- Structure allows no successors
- Slot has no successor
### Figure 2a: IPL Data Structure Diagram (L, p. 207, fig. 3)

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2</td>
<td>9-1</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>9-7</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>ø</td>
</tr>
<tr>
<td>9-1</td>
<td>ø</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>A1</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>V1</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>A5</td>
<td>ø</td>
</tr>
<tr>
<td>9-7</td>
<td>ø</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>9-5</td>
<td>ø</td>
</tr>
<tr>
<td>9-10</td>
<td>15</td>
<td>ø</td>
</tr>
<tr>
<td>9-5</td>
<td>ø</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>2L</td>
<td>ø</td>
</tr>
<tr>
<td></td>
<td>2H</td>
<td>ø</td>
</tr>
</tbody>
</table>

### Figure 2b: IPL Data Structure Table

<table>
<thead>
<tr>
<th>Row</th>
<th>Label</th>
<th>Data</th>
<th>H</th>
<th>D</th>
<th>L</th>
<th>R</th>
<th>C</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>9-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>3</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>9-7</td>
<td></td>
<td></td>
<td>*</td>
<td>4</td>
<td></td>
<td>7</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td></td>
<td></td>
<td>*</td>
<td>6</td>
<td></td>
<td>5</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>9-5</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td>Z1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>8</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>10</td>
<td>&quot;attributes&quot;</td>
</tr>
<tr>
<td>9</td>
<td>A5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>&quot;values&quot;</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>V1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>9-10</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FIGURE 2C: IPL STORAGE STRUCTURE DIAGRAM
Figure 3a: LISP Data Structure Diagram (cf. 3, p 191, fig. 4b)

Figure 3c: LISP Storage Structure Diagram

Figure 3b: LISP Data Structure Table
"Workspace Expression": W/140, NAMEA AA AB AC, NAMEB BA + Y/152, NAMEC CA CB CC, NAME D A, NAMEE EA EB + Z

**Figure 4a: Comit Data Structure Diagram (cf. 5)**

<table>
<thead>
<tr>
<th>Row</th>
<th>Label</th>
<th>Data</th>
<th>U</th>
<th>D</th>
<th>L</th>
<th>R</th>
<th>C</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Workspace</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[left and right boundary]</td>
</tr>
<tr>
<td>2</td>
<td>L1</td>
<td>Y</td>
<td></td>
<td>2A</td>
<td>4</td>
<td>3</td>
<td>5, 6</td>
<td>&quot;Workspace Constituents&quot;</td>
</tr>
<tr>
<td>3</td>
<td>L9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12, 13,</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>L20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>L2</td>
<td>140</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>7</td>
<td>&quot;numeric subscript&quot;</td>
</tr>
<tr>
<td>6</td>
<td>L3</td>
<td>NAMEA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[&quot;logical subscripts&quot;]</td>
</tr>
<tr>
<td>7</td>
<td>L4</td>
<td>NAMEB</td>
<td></td>
<td></td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L5</td>
<td>AA</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>10</td>
<td>&quot;Values&quot;</td>
</tr>
<tr>
<td>9</td>
<td>L6</td>
<td>AB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>L7</td>
<td>AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>L8</td>
<td>BA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;value&quot;</td>
</tr>
<tr>
<td>12</td>
<td>L10</td>
<td>152</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;numeric subscript&quot;</td>
</tr>
<tr>
<td>13</td>
<td>L11</td>
<td>NAMEC</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>14</td>
<td>&quot;logical subscripts&quot;</td>
</tr>
<tr>
<td>14</td>
<td>L12</td>
<td>NAMED</td>
<td></td>
<td></td>
<td></td>
<td>19</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>L13</td>
<td>NAMEE</td>
<td></td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>L14</td>
<td>CB</td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td></td>
<td>&quot;Values&quot;</td>
</tr>
<tr>
<td>17</td>
<td>L15</td>
<td>CB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td>&quot;value&quot;</td>
</tr>
<tr>
<td>18</td>
<td>L16</td>
<td>CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>L17</td>
<td>DA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;value&quot;</td>
</tr>
<tr>
<td>20</td>
<td>L18</td>
<td>EA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>21</td>
<td>&quot;Values&quot;</td>
</tr>
<tr>
<td>21</td>
<td>L19</td>
<td>EB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4b: Comit Data Structure Table**
FIGURE 4C: COMIT STORAGE STRUCTURE DIAGRAM
**FIGURE 5a: MULTI-LIST DATA STRUCTURE DIAGRAM (7, Table 1)**

<table>
<thead>
<tr>
<th>ROW</th>
<th>LABEL</th>
<th>DATA</th>
<th>L</th>
<th>D</th>
<th>L</th>
<th>R</th>
<th>C</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L1</td>
<td>LIST OF LISTS</td>
<td>20</td>
<td>25</td>
<td>29</td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L11</td>
<td>AGE</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>L12</td>
<td>RANK</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>L13</td>
<td>HEIGHT</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>L111</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>L112</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>L113</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L114</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>L115</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>L121</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>L122</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>L123</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>L13</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>L131</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>L132</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>L133</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>L134</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>L171</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>L172</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>L173</td>
<td></td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>29</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 5b: MULTI-LIST DATA STRUCTURE TABLE**
FIGURE 5C: MULTI-LIST STORAGE STRUCTURE DIAGRAM
Figure 6a: Doubly-chaired tree data structure diagram (cf. [6, p.490, fig. 1])

Figure 6b: Doubly-chaired tree data structure table
Figure 7a: Trie memory data structure diagram (16, p990, fig. 1)

Figure 7b: Trie memory data structure table
Figure 7c: True Memory Storage Structure Program
Figure 8a: Temac Data Structure Diagram (cf. [17])

Hello, said John.

Figure 8b: Temac Data Structure Table (Subtree Order)
### Figure 8c: TIMAC Data Structure Table (Level Order)

<table>
<thead>
<tr>
<th>Row</th>
<th>Label</th>
<th>Data</th>
<th>U</th>
<th>D</th>
<th>L</th>
<th>R</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>P0</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>P1</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S1</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S2</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>S3</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S4</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>S5</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>W1</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>W2</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>W3</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>W4</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>S10</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>W5</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>S6</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>S7</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>S8</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>S9</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>S10</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>S11</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 8d: TIMAC Storage Structure Diagram

```
1
   / \                  / \  
 2   3 4            2   4
     / \            / \  
    5   6 7        5   6
       \   \        \   \  
         10 11 13    10 11 13
```

**Data Area**:
- **HELLO**
- **NAME**
- **DAJHN**
- **...**
Figure 9a: Braid data structure diagram (18, p12)

Figure 9b: Braid data structure table (subtree order)

Figure 9c: Braid data structure table (level order)
FIGURE 9d: BRAID STORAGE STRUCTURE DIAGRAM
**Figure 10a: SLIP Data Structure Diagram**

**Figure 10b: SLIP Data Structure Table**
FIGURE 10c: SLIP STORAGE STRUCTURE DIAGRAM
A = "address" or label

f = "flag"

1 = head of threaded list
2 = end of list
φ = other element

L
if "f" was 1,
L is down successor
if "f" was φ or 2,
L is data

R
if "f" was φ or 1,
R is right successor
if "f" was 2,
R is "up" successor

FIGURE 11a: THREADED LIST DATA STRUCTURE DIAGRAM (29) P.196

<table>
<thead>
<tr>
<th>Row</th>
<th>Label</th>
<th>Data</th>
<th>L</th>
<th>D</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L1</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>L2</td>
<td>b</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>L3</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>L4</td>
<td>f</td>
<td>-</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>L5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>L6</td>
<td>j</td>
<td>5</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>L7</td>
<td>n</td>
<td>3</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>8</td>
<td>L8</td>
<td>-</td>
<td>9</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>L9</td>
<td>h</td>
<td>-</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>L10</td>
<td>e</td>
<td>8</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>11</td>
<td>L11</td>
<td>i</td>
<td>1</td>
<td>-</td>
<td>*</td>
</tr>
</tbody>
</table>

FIGURE 11b: THREADED LIST DATA STRUCTURE TABLE
Figure 11c: Threaded List Storage Structure Diagram
**Figure 12a: Snobol Data Structure Diagram**

- **Line 1**: "Around, around the sun we go!
- **Line 2**: "The moon goes round the earth"
- **Text**: Line 1 + Line 2

**Figure 12b: Snobol Data Structure Table**

<table>
<thead>
<tr>
<th>Row</th>
<th>Label</th>
<th>Data</th>
<th>U</th>
<th>D</th>
<th>L</th>
<th>R</th>
<th>C</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Text</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>2</td>
<td>NAME/SIZE OF &quot;TEXT&quot;</td>
</tr>
<tr>
<td>2</td>
<td>LINE 1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>NAME/SIZE OF &quot;LINE 1&quot;</td>
</tr>
<tr>
<td>3</td>
<td>L10</td>
<td>Text</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>L11</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>L12</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>6</td>
<td>L13</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>7</td>
<td>L14</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8</td>
<td>L15</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>9</td>
<td>L16</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>10</td>
<td>L17</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>11</td>
<td>L18</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>12</td>
<td>L19</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>13</td>
<td>L20</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>14</td>
<td>L21</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>15</td>
<td>L22</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>16</td>
<td>L23</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>17</td>
<td>L24</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>18</td>
<td>L25</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>19</td>
<td>L26</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>20</td>
<td>L27</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>21</td>
<td>L28</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>22</td>
<td>L29</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>23</td>
<td>L30</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>24</td>
<td>L31</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>25</td>
<td>L32</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>26</td>
<td>L33</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>27</td>
<td>L34</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>28</td>
<td>L35</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>29</td>
<td>L36</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>30</td>
<td>L37</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>31</td>
<td>L38</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>32</td>
<td>L39</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>33</td>
<td>L40</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>34</td>
<td>L41</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>35</td>
<td>L42</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>36</td>
<td>L43</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>37</td>
<td>L44</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>38</td>
<td>L45</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>39</td>
<td>L46</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>40</td>
<td>L47</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>41</td>
<td>L48</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>42</td>
<td>L49</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>43</td>
<td>L50</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>44</td>
<td>L51</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>45</td>
<td>L52</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>46</td>
<td>L53</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>47</td>
<td>L54</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>48</td>
<td>L55</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>49</td>
<td>L56</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>50</td>
<td>L57</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>51</td>
<td>L58</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>52</td>
<td>L59</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>53</td>
<td>L60</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>54</td>
<td>L61</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>55</td>
<td>L62</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>56</td>
<td>L63</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>57</td>
<td>L64</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>58</td>
<td>L65</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>59</td>
<td>L66</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>60</td>
<td>L67</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>61</td>
<td>L68</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>62</td>
<td>L69</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>63</td>
<td>L70</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
FIGURE 12C: SNOBOL STORAGE STRUCTURE DIAGRAM
**Figure 13a: Penelope Data Structure Diagram (C [26])**

**Figure 13b: Penelope Data Structure Table**

<table>
<thead>
<tr>
<th>Row</th>
<th>Label</th>
<th>Data</th>
<th>U</th>
<th>D</th>
<th>L</th>
<th>R</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p</td>
<td></td>
<td></td>
<td>3,33,39</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>p</td>
<td></td>
<td></td>
<td>4,9,17,25</td>
<td>*</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s1</td>
<td></td>
<td>9</td>
<td>8,6,7,15</td>
<td>*</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s2</td>
<td></td>
<td>9</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>c1</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>c2</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>c3</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>c4</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>c5</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>c6</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>c7</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>c8</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>c9</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>c10</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>c11</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>c12</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>c13</td>
<td></td>
<td>9</td>
<td>14,15,16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FIGURE 13c: PEMEOPE STORAGE STRUCTURE DIAGRAM