4. This area is a fruitful area for standardization activities at some levels, but standardization in the classical sense is not the simple full answer. The development of a collection of tools and techniques, means of evaluating them, means of selecting the appropriate ones for an environment, means of describing the environment all must be developed. Included in this collection will be standards (as there are already – USASCII for example).

The technical areas require considerably more work before any meaningful approach can be taken. This may be accomplished in many ways. A standardization activity to monitor developments and initiate activity whenever standardization within an area becomes potentially fruitful is desirable. The problem of separating the problem into areas more amenable to attack must be addressed by a collection of people conversant in the area. The services of the newly formed ACM SIC File Description and Transformation (SICFIDET) have been offered to perform developmental work, and this is one area in which it might fruitfully pursue activities. Well defined developmental activities, which are not being done other places, should be initiated, but I cannot suggest how within the standards framework. The problems of communication should be pursued by the establishment of journals, newsletters, symposia, and conferences – all aimed at creating an atmosphere in which exchange of ideas and information can take place readily, rapidly, and without prejudice. This is also an area in which professional bodies, such as the new ACM SICFIET, will be active. Ways of encouraging and supporting those activities should be pursued. Enough of a standards framework should be established to monitor activities when necessary and to provide guidance to the computing community in areas related to potential development of standards.

REFERENCES

1. Knuth, D.E. *The Art of Computer Programming, Vol. I: Fundamental Algorithms* (Addison-Wesley Publishing Company, Reading, Mass., 1968).

INFORMATION STRUCTURES: TOOLS IN PROBLEM SOLVING

M.E. D'Imperio

I. BASIC APPROACH AND FRAME OF REFERENCE

A. Computer-Oriented Problem-Solving as a Subset of General Human Problem-Solving

The basic approach, in this paper, to the topic of information structures is through the phrase *Tools in Problem Solving*. Human problem-solving activity is the crucial context, within which information structures are regarded as *tools*, actively chosen or designed by a human problem-solver in pursuit of a goal. Computer-oriented problem-solving is a restricted subset of human

problem-solving behavior (in which we are especially interested) but differs in no *essential* respect from other human problem-solving; in fact, it invariably is found embedded within a larger sequence of activities not directly involving the computer. Many of our most stubborn and obstructive difficulties in the application of computers stem from our failure to pay sufficient attention to the manual, human environment of noncomputerized procedures which feed into and emerge from the machine procedures; computer specialists too often tend to approach the machined portions of a total effort as if those portions existed in isolation.

While it is true that all features of a problem for computer solution must be stated more fully, more precisely, and more concretely than is customary for purely manual solution methods, the essential point is that computer problem-solving is performed, ultimately, *by people* and *for people*. The primary considerations which must apply to the problem-solving situation as a whole are those relevant to the operating characteristics of the human mind — its limitations, its requirements, and its preferred modes of operation — rather than to the operating characteristics of computers. Instead of starting with the computer, software, etc., and working back to the problem as an *application* of these, I start with the problem and the human plans for solving it and then work forward to the computer and its associated techniques for information organization and representation, regarding them as means for implementing portions of problem-solving plans.

Most writers seem to have said, "Here are a lot of techniques, and here are some kinds of problems you can solve with them." I prefer: "Here are some kinds of information patterns useful to people in solving problems, and here are some of the ways those patterns can be realized in computer memories and software."

B. Computer-Oriented Information Structures as a Subet of General Human Methods of Information Representation

As a subset of general human methods of information representation, computer register arrangements, file structures, data-types, and file descriptions are not isolated, specialized techniques, invented just for the world of the computer and unrelated to anything else. Viewed in this light, they exhibit a close kinship to other, much older, pencil-and-paper techniques for organizing information, exploited by reasoning men long before computers were thought of. Computer programmers and systems analysts make liberal use of these informal techniques at various stages in their efforts to develop, design, debug, and evaluate a program. Flow charts, narrative procedures and descriptions, classification tables, tree diagrams, organization charts, etc., are used and taken for granted by computer specialists every day. These pencil-and-paper forms of information structures are of great interest in their own right and deserve far more attention and respect than they customarily receive.

We have much to gain by viewing computer hardware and software data-structures, file structures, and the like as a special class of problem-solving models, which are built up from a small set of elementary human psychological patterns, used over and over again by reasoning man in organizing information and plans during any goal-directed activity. Just as computer problem-solving forms a subset within all human problem-solving, so do computer data-organization techniques form a subset within human patterns of information representation. By looking at software and hardware storage structures in this way, we may effect a considerable clarification of the field so cluttered at present with apparently disconnected jargon and gadgetry.

II. A CLASSIFICATION FOR INFORMATION STRUCTURES

Based on the concepts outlined previously, a classification of information structures is presented in the following Sections.* Reflecting a distinction which has great practical importance, the classification divides information structures into two main classes:

1. *Modelling structures* consist of a set of abstract psychological (or heuristic) plans, arrangements, and descriptions. The modelling structures are generally applicable to any problem, take an implicit or explicit part in everyday human reasoning and planning behavior, and are largely independent of the characteristics of any particular language, device, or recording medium. The emphasis in modelling structures lies on the characteristics of the data and of the problem and on the general characteristics of methods for solving the problem.

2. *Implementation structures* comprise concrete, specific kinds of arrangements of data elements in units of storage in some recording or memory medium. They are relatively machine- or language-dependent and are the realization of a model in a given machine, a specific kind of memory, and a particular software system. At their most general, they realize a problem-solving plan in a particular class of machines, memories, or software systems. The emphasis in implementation structures lies on the characteristics of the medium, language, memory, or machine.

The diagrams in Figs. 1 – 9 are for some elementary modelling structures and some useful structures built up from them. The diagrams in Fig. 10 use slightly different conventions and are for some storage structures and some modelling structures which they are capable of realizing.

III. MODELLING STRUCTURES

Modelling structures are detailed arrangements which the problem-solver plans to build up, stepwise, as operands and patterns of operands, from some set of elementary units into which he has analyzed the data for the problem and from some set of elementary operations he has chosen to apply to those units. A simple form of graph diagram has been employed in Figs. 1 – 9 to represent some elementary modelling structures and some useful structures built up from them; in a sense, these graphic diagrams serve as pencil-and-paper *storage structures* for the modelling structures they show. The two dimensions of the plane in a sheet of paper are utilized with special meanings: Paths in the left–right dimension show *sibling* relations within a single elementary structure, while up–down paths stand for relations between nodes in different elementary structures. Nodes where elements may occur are shown as open circles, within or near which a character-string (representing a data item) is written when data is present at the node.

---

*Only a very brief characterization of the main terms and concepts are presented in this paper. The author is in the process of completing a text which develops, describes, and illustrates the classification of information structures in much fuller detail.

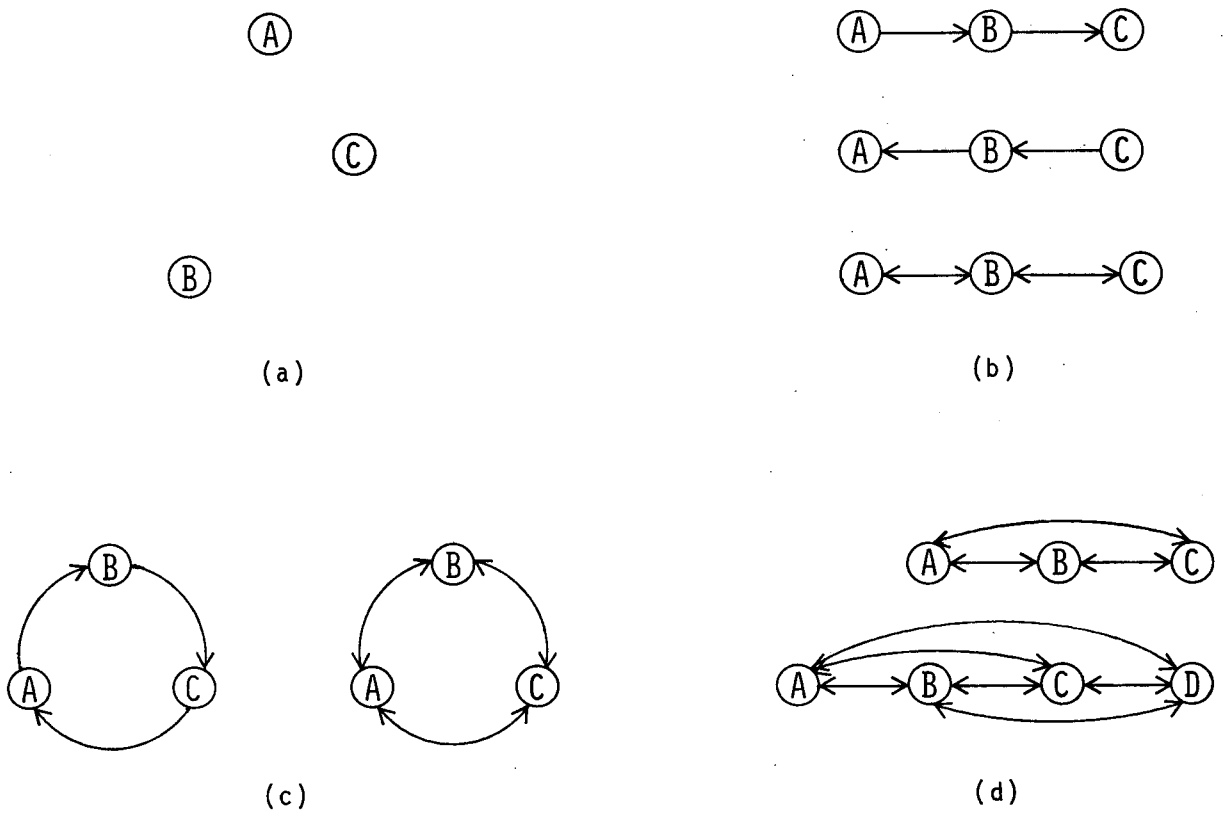

FIG. 1. Varieties of Internal Ordering: (a) Unconnected Set (Unary Lists); (b) Linear Lists; (c) Circular Lists; (d) Fully Connected Lists.

Access paths are represented by lines between the nodes; an arrowhead on the line touches any element to which the path provides access; a solid line is used for neighbor paths within an elementary structure; wavy lines stand for *external* access paths to an element from outside of its own structure; and dashed lines are used to show reference paths linking different structures as wholes. The left-to-right order in which the nodal elements are shown represents the order of priority which the problem-solver has accorded to them within their elementary structure (before the action of the step or set of steps he is about to consider has taken place) so that the currently produced operand structure will be created with reference to this arrangement. Thus, a right-to-left arrow in a diagram implies an inversion of the previously established ordering in the operands produced by following that path. Similary, a top-to-bottom arrow between nodes means that the upper element is the one which initiates a call upon, or leads into, the lower element. An arror pointing upward, then, implies a reversion or backing up to a higher-order element.

It must be reemphasized that the modelling structures are not intended to represent any concrete set of computer registers or bytes in any actual hardware or software memory. Neither are these structures intended to show actual configurations of data items from some particular problem (e.g., characters of text, coefficients of a polynomial, uniterms applied to documents for indexing,

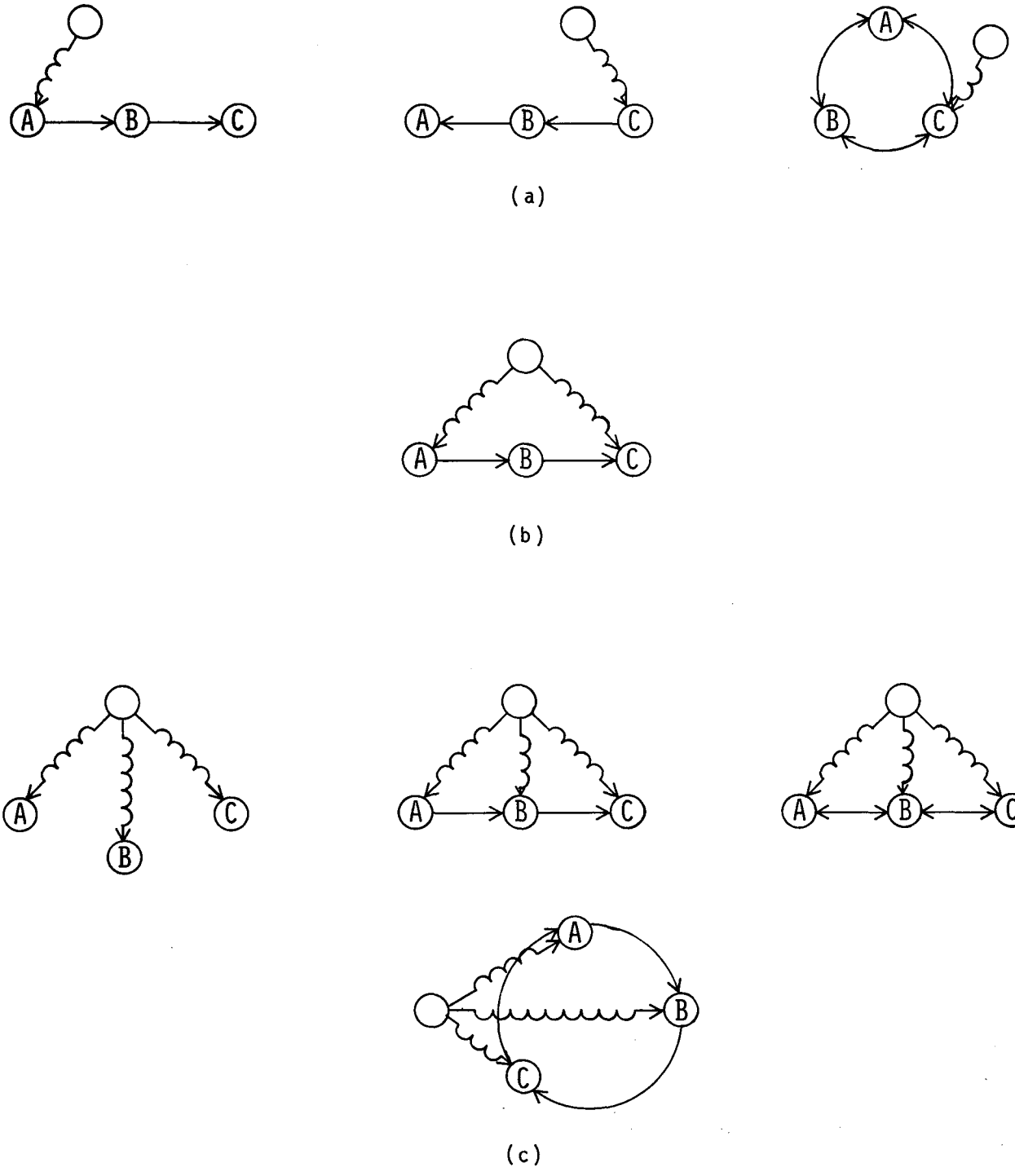

FIG. 2. Varieties of External Access: (a) Single-Point Access; (b) Double-Ended Access; (c) Equal Access to Set, Lists, Ring.

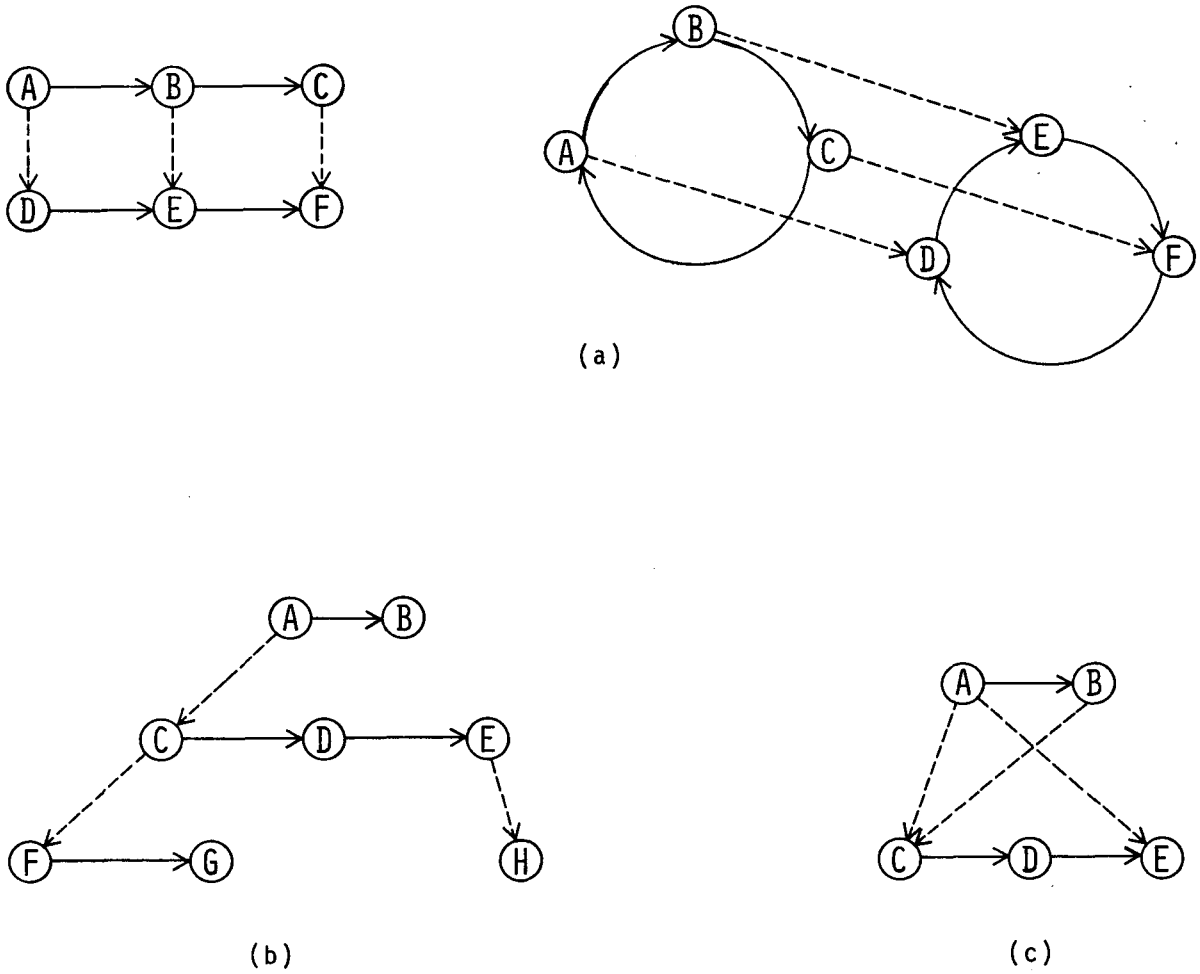

FIG. 3. Varieties of Reference: (a) Collateral Lists; (b) Hierarchical Lists; (c) Overlapped lists.

storage and retrieval, etc.). These diagrams represent an attempt to express purely structural (or psychological) groupings of items or of positions where items may be found, generally applicable to any problem and data. As he creates modelling structures, the problem-solver considers the utility of arranging his character-strings, coefficients, uniterms, etc., in the general *shapes* of linear lists, rings, trees, or other modelling patterns, to permit him to generate and transform the operands of his algorithm.

A. Basic Properties of Modelling Structures

The kinds of elementary structures that enter into the problem-solver's plans may be regarded as varying in three important ways: in *internal ordering* of nodes within an elementary structural unit; in *external access* to a node within

a structural unit from outside of that unit; and in different kinds of *reference* among structural units as wholes making up a higher-order structure. By various combinations of the different possibilities of ordering, access, and reference, a wide variety of useful structural patterns can be described.

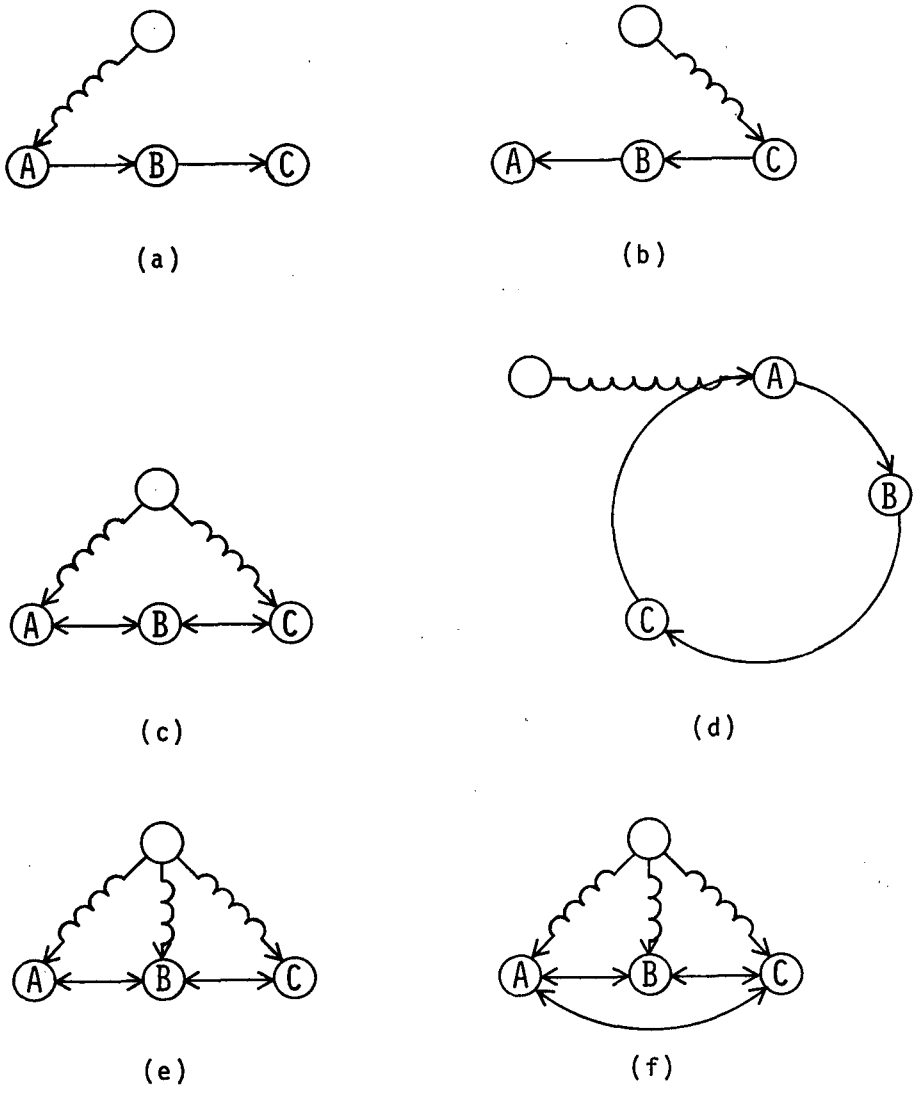

FIG. 4. Simple Modelling Structures: (a) Chain; (b) Lifo List; (c) Shelf; (d) Forward Ring; (e) Two-Way Fan; (f) Array.

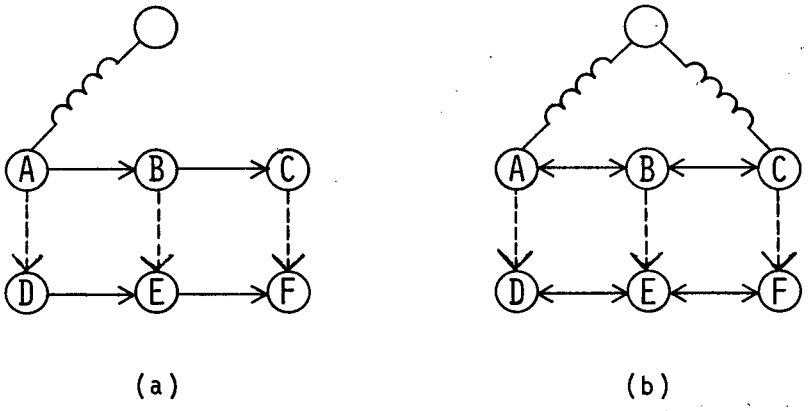

FIG. 5. Some Collateral Structures: (a) Chain Table; (b) Shelf Table.

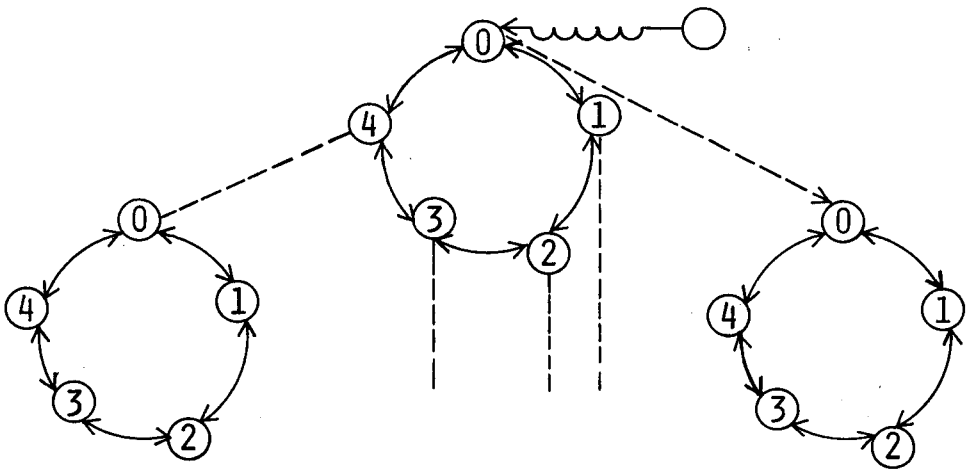

FIG. 6. A Ring Tree.

1. Varieties of internal ordering

The elementary structural unit of modelling structures is the list. The word *list* is not employed here in the sense that computer specialists usually impute to it, that of the *storage structure* involving linked address coupling and made familiar in the IPL series of languages. Instead, it is used, in a sense closer to its everyday speech meaning, to refer to any ordered sequence

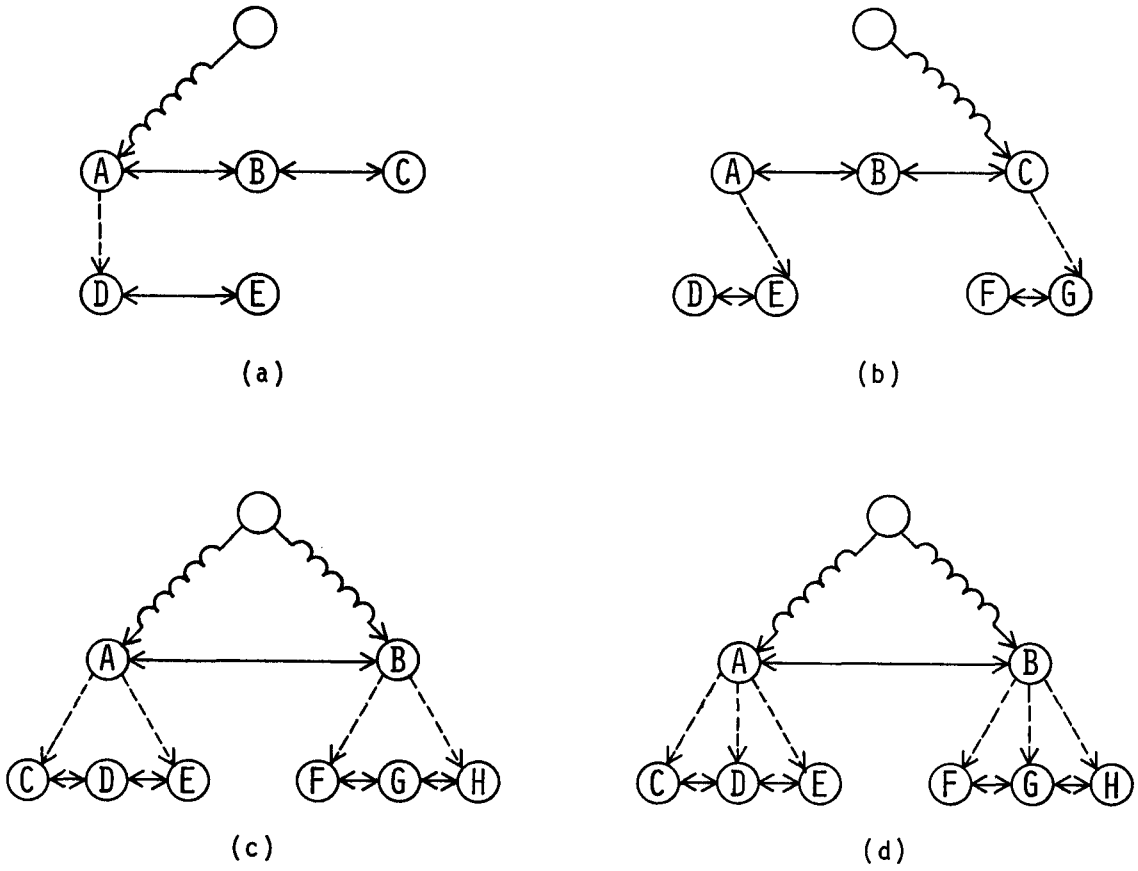

FIG. 7. Some Hierarchical Structures: (a) Chain Tree; (b) Lifo Tree; (c) Shelf Tree; (d) Fan Tree.

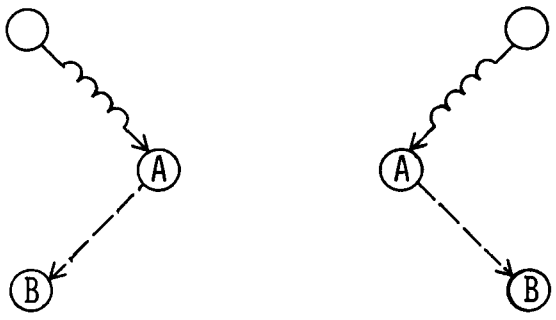

FIG. 8. Binary or General Trees.

of items, with no restrictions upon, or implications concerning, the way in which the order is recorded or brought about. A shopping list of items in a column on a sheet of paper, a vector of values stored in a FORTRAN *array*, a sequence of English words stored in a chain of registers linked by pointer addresses, a row of numbers written across a page – all are examples of lists (in the author's sense) implemented in a wide variety of storage structures. The convention followed here requires that, within any single elementary list, the same ordering

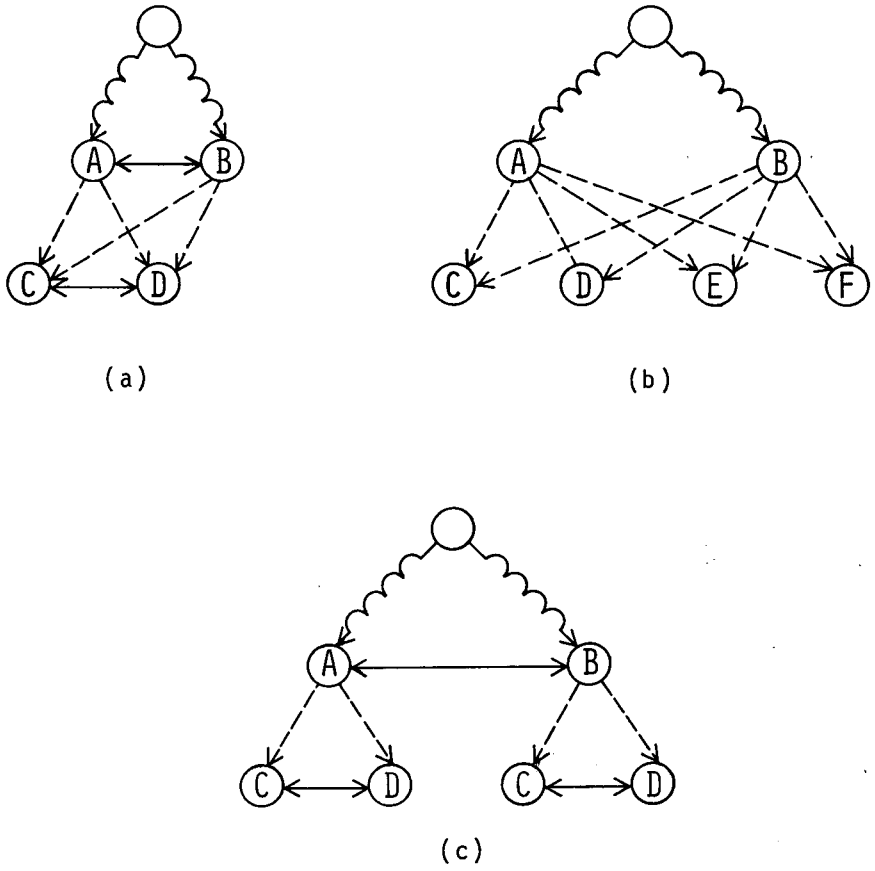

FIG. 9. Overlapped Structures: (a) Shelf Web; (b) Fan Web; (c) Shelf Tree Equivalent to (a).

pattern is to be used uniformly over all the nodes; otherwise a set of structures so chaotic as to be inaccessible to any interpretation rule would result. Similarly, compound structures (made up of elementary structures) require that all component lists have the same ordering patterns; they may, of course, vary in length and content. Higher-order structures made up of components having heterogeneous ordering patterns will be called *complex structures* and will not be mentioned further.

There are four principal varieties of internal ordering: *unary lists*, *linear lists*, *circular lists*, and *fully connected lists*. A unary list is a minimal or degenerate case of a list; it consists of a single item which has a place in the psychological space upon which the modelling structures are plotted but has no immediate siblings. It is the simplest entity that can occur in a modelling structure and constitutes a one-element list. *Linear Lists* may be one-way forward (left to right) or one-way backward (right to left) and permit paths between siblings in one direction only. *Circular lists* are elementary structures which have no first or last nodes but provide sibling paths forward (clockwise), backward (counterclockwise), or in both directions: forward, and two-way circular lists. *Two-way lists* permit movement in either

direction from any node to a sibling. Operands in circular lists may be generated by following paths from node to node in a given direction, continuing until the movement is halted by some arbitrary specification (e.g., length of the generated operand string, or encountering a particular data item a given number of times); the operand string may contain any number of repetitions of the data items in their sequence. *Fully connected* lists allow direct access from any node to any other node.

2. Varieties of external access

Single elements, linear lists, and circular lists may be reached as units, *by name* as it were, from some other region of the psychological space by a convention specifying a particular node (or nodes) as entry point. The problem-solver may need a structure which permits access via the right- or left-most node; these possibilities are called *single-point access* and, more specifically, right access and left access. *Double-ended access* allows the problem-solver to get at the list through either end, and *equal access* allows him to get at any node on the list from outside. In general, one can think of external access as a more distant path to an element in a list, requiring a bigger psychological jump than paths between siblings require.

3. Varieties of reference

Structures may be arranged in patterns to form higher-order structures by establishing *reference paths* between them; the higher-order structures thus produced may, as stated above, be compound (all having the same ordering and access patterns) or complex (having heterogeneous patterns). Varieties of reference are *collateral reference*, *hierarchical reference*, and *overlapped reference*. *Collateral* paths between lists provide access across the component structures in such a way that elements in each path are associated with corresponding elements in a collateral structure or structures; the paths may be one-way or mutual. *Hierarchical* paths lead down from a node in one structure to another subordinate structure via allowed external access nodes of the latter. Paths in an upward direction lead from the external access node or nodes of the sublist back to a single node on the higher-order list. *Overlapped* reference paths permit more than one element in a list to refer downward to the same subordinate element in a lower-order structure and, likewise, permit one subordinate element to refer upward to multiple elements on a superordinate list. These paths are still restricted to the access nodes of the component elementary structures.

B. Examples of Some Useful Structures

The basic concepts and diagrams described above can now be applied to the task of classifying and comparing a number of modelling structures that have been utilized over and over again, under various names and guises, by programmers and other computer specialists.

1. Simple structures

Some simple modelling structures frequently encountered in computer problem-solutions may be formed by combining internal ordering and access conventions so as to specify an elementary list capable of providing certain sets of

operand strings; in the diagrams of Fig. 4, these are shown as if they were entered from a single external element

a. Chain (or left-access, forward linear list)

Internal elements are accessible only from left to right, and the list as a whole is entered from without only at its left-most element. This structure may be called a FIFO (First-In-First-Out) list as well, since the first element read must always be the first element that was stored.

b. LIFO (Last-In-First-Out) list

This structure may be seen as a special case of a right-access two-way list in which storing or putting of elements is performed from left to right (starting from the right-most node), and reading or getting of elements is performed from right to left (also starting with the right-most node) and crossing out or erasing each node as it is read [see Fig. 4(b)].

c. Shelf (or double-ended access, two-way linear list)

See Fig. 4(c).

d. Ring (or circular list)

Figure 4(d) is a single-point access, forward circular list, or *ring*, such as might provide operands for modular addition. Forward, backward, and two-way rings are possible, and they are useful in many applications.

e. Fan (or equal-access list)

Figure 4(e) is a list in which every element is equally accessible from outside the structure, and internal paths may move in either direction. One-way, two-way, and circular fans are possible.

f. Array (or fully connected list)

Figure 4(f) is a structure which, while very familiar to computer specialists and universally utilized, turns out to have a surprisingly complex diagram. The *array* is a list whose elements are equally accessible from outside and also from one another; it is not really a *linear* list, strictly speaking.

2. Compound structures

A compound structure is made up of a group of simple structures, all of the same type, the whole having at least one external access point. Compound structures may be classified as *collateral*, *hierarchical*, and *overlapped*, in accordance with the type of reference paths employed to bind together their component structures.

a. Collateral structures (or tables)

Tables are made up of two or more simple lists, related by collateral reference paths, so that elements in corresponding positions across all the lists may be obtained to form an operand string. Other possible names for such structures are *matrices*, *parallel lists*, or *orthogonal lists* (see Fig. 5).

b. Hierarchical structures (or trees)

The term *tree* is used here in a special way to mean a hierarchically referenced modelling structure; it is not intended to coincide with formal definitions of trees in graph theory. Here, a tree always means a structure having the following characteristics: It is made up of two or more simple lists (in our sense of *list*) bound together by hierarchical reference paths. Thus, trees may be made up of rings, as well as linear lists and single elements; an example of a ring tree may be seen in Fig. 6, and other types of trees are shown in Figs. 7 and 8.

The subject of trees has been discussed at great length in Computer Science literature by writers preferring various definitions of the term *tree* and emphasizing a wide variety of practical applications for the structures they discuss. The most complete treatment from the point of view of the computer programmer is probably that of Knuth [*3*]. Figure 8 shows two trees, which may be regarded either as binary or unary trees (*general* trees, as Knuth says); if the diagrams are considered as representing a pair of binary trees, they are not equivalent since one has only a left subtree and the other only a right subtree. If they are considered as *general*, they are representations of the same unary tree structure. As Knuth points out, a binary tree is not simply a general tree having no more than two nodes under any given node; it is a special kind of hierarchical modelling structure, with distinct functional characteristics, and is of particular importance to the computer specialist because general trees are often reshaped into binary trees to render them more manageable in machine programs. A binary tree may be regarded as always providing a two-way choice at each node, leading to two distinct outcomes on the level below.

3. Overlapped structures (or webs)

An overlapped structure may be made up of any simple structures; thus there may be webs made up of chains, shelves, LIFO lists, rings, fans, and arrays. Figure 9 shows a *shelf web* and a *fan web* and also demonstrates the equivalence of a shelf web containing two elementary structures and a shelf tree containing three, one of which is a repetition of another [Fig. 9(c)].

IV. IMPLEMENTATION STRUCTURES (STORAGE STRUCTURES)

A. Basic Concepts and Properties

1. Hardware and software storage structures

The most obvious classification of implementation structures at the outset is as hardware or software structures. *Hardware* storage structures are

engineered into the operation of a machine – in its logic, electronic addressing, fetching, storing mechanisms, etc. *Software* storage structures are accomplished by programming and are always superimposed upon a hardware memory which may or may not have similar characteristics. The B5500 stack memory and the core memory of the 7094 are examples of hardware storage structures; a SNOBOL4 work space, an ALGOL array, and a formatted card or magnetic tape file are examples of software storage structures – all requiring programming to reach and manipulate the data fields they contain.

2. Substrate and superimposed storage structures

A useful distinction may be made between the *superimposed* memory structures ("on the surface") which the problem-solver is consciously using or creating and the substrate structures underlying these as building blocks, and in terms of which the superimposed structure is formed. In describing any memory system as it presents itself to a prospective user, we can consider the most explicit higher-level storage arrangements (type declarations, grouping of fields in data description statements of file description statements, etc.) as superimposed features while all other bit, register, or field groupings, formats, etc., at more elementary levels, can be regarded as substrate, in that the user may disregard them in his normal use of the system.

3. Activity levels in a memory

For expository purposes, another convenient distinction is that among three levels of activity in a storage system as viewed by a user: (1) the *window registers*, or most active and basic level, involving a relatively small number of working registers (hardware or software) through which all addressing and manipulation must take place; (2) an *active memory* from which the window registers' activities are returned; and (3) a *backup memory* which provides and receives data for the active memory. The active memory is typically considerably larger than the number of window registers, and each of its storage units is usually of the same size as the largest or most powerful of the window registers; active memory units are so structured within themselves as to provide all or most of the subfields appropriate to certain window registers. The backup memory is larger again by at least an order of magnitude, slower of access, and less finely divided; each of its functional units is typically made up of a string of units in the active memory, and structures in it are formed on the assumption that they will be processed in the active memory via the window registers and their subfields. There may be more than one level within the backup memory, or more than one alternative choice among kinds of backup memory with different characteristics or for different purposes in the system as a whole.

4. Cell-oriented and byte-oriented memories

The smallest units of data underlying other structures in a memory may be of two basic types: (1) sequences of bits filling all or some major portion of a window register, or (2) sequences of bits constituting small modules, all of equal lengths, called *bytes*, which the window registers are capable of processing individually or in groups. A memory in which most window registers operate upon a cell-full of bits at a time, such as the 7094, is a cell-oriented memory (often called a *word-machine*, where *word* is used to describe a register, or cell, in what seems to me a very unfortunate accident of terminology). By contrast, a memory which has window registers capable of handling individual bytes, called *positions* or *characters* in many applications, is a byte-oriented

memory; machines having such memory structures (e.g., the IBM 1401, 705, and 7080) are often called *character-machines*. Some machines, like the 360 systems and the Burroughs 5500, provide a parallel pair of structures permitting character-mode (byte-oriented) and word-mode (cell-oriented) operations. In the 360, cells are built up of integral numbers of bytes, so that a wide variety of fixed-length and variable-length operands may be formed in the system's window registers (accumulator, index registers, etc.).

5. Access characteristics of memories

Another important basic characteristic of memories is concerned with the number and distribution of external access points for reading and writing data. Common varieties of access in memory devices may be classified as *random*, *modular*, *cyclic*, *linear*, and *parallel*. These terms are intended to describe the behavior of the memory as it presents itself to the user and not to label an inherent property of the device; it should be remembered that any device may be so used, or so modified, as to alter its usual or expected external access characteristics for the problem-solver's purposes. Certain hardware storage devices, however, are typical of each of the above access patterns and are used as examples in the following Sections.

a. Random access

Every individual storage slot in the memory is equally accessible within a relatively brief unit of time. The memory acts as if each slot had its own independent read/write head. Magnetic core memory is an example of a random access memory.

b. Modular access

A large number of randomly accessible entry points are provided, but they are distributed at intervals throughout the memory so that each services a group of single slots. The memory acts as if it were divided into a large number of separate small modules, each having its own read/write arm or head. Examples of modular access are some data-cell (or magnetic-card) memories, multiarm disc devices, and drums.

c. Cyclic access

The memory has a relatively limited number of entry points which are sampled at intervals in a repeating cycle. It acts as if one or a few read/write heads were revolving about a fixed surface or were fixed at intervals above a revolving surface. Some drums and single-arm discs, or discs having very few arms, behave in this way.

d. Linear access

The memory has only one read/write head or entry point which traverses the memory spaces from beginning to end. Reverse movement is permitted only under restricted circumstances, usually for purposes of repositioning the access point only and not for reading or writing, which can typically occur only in a forward direction. The memory acts like a long thin ribbon with a single read/write head. Magnetic tapes, punched-paper tapes, and punched cards stacked in a card reader are examples of linear access memories.

e. Parallel access

Portions of a number of different memory spaces are simultaneously accessible to the outside world. The memory acts as if a number of read/write heads were side by side, slicing through it at some point. The parallel behavior is usually a mode provided to a memory in addition to a basically modular or random access mode, and it operates in a direction perpendicular to that in which material is read or written in the modular or random mode. Thus, data stored in the random mode may be read *in parallel*. Memories of this kind are still largely experimental and are quite expensive to build; they are commonly called *content-addressed* (or *associative*) memories as well as parallel memories.

B. A General Model for Storage Structures

The abstract model presented here is applicable to hardware and software, substrate and superimposed levels, various activity levels and access characteristics, and cell-orientation or byte-orientation in memories. It may be used as a classification for storage arrangements the problem-solver would like to create, or for features of a given memory system which the problem-solver would like to evaluate or to use. Other writers have devised similar abstract models or embodied them in the philosophy of some programming language. Notable among these more-or-less explicit theories of storage structure are those of Standish [*4*] and Knowlton [*5*]. Some points of correspondence between these models and the one to be presented here are described in the following Sections.

1. Basic storage elements

a. Storables and addresses

The basic elements in a storage structure are its uniquely addressable *storage slots*, each capable of accommodating one storable or data element arising in a problem-solution process. Each slot may be considered to have a habitation and a name, as it were, in that each possesses a physical location in the memory with respect to some external access point, and a key or symbol by which it may be called up or which may stand for it; together, the location and the key constitute the *address* of the slot in the memory system. Access paths between these slots may be diagrammed along the horizontal and vertical dimensions of a sheet of paper, with the same meanings as in diagrams for modelling structures. Left and right paths lead to close neighbors of siblings, while down and up paths lead to alternative, subordinate, or superordinate elements and link together elementary sequences to form compounds. Diagrams for representing storage structures are given and described in Sec. V, and some comparisons are made between modelling structures and storage structures for implementing them.

b. The storage slot

The slot is the smallest individually addressable element in the memory; it is the minimum element having a unique identifying sign or expression by which it may be called up and acted upon. The identifying sign of a slot is its

address, consisting of a distinct location and key or of a single expression that functions both as location and key.

The slot concept presented here corresponds closely to the concept of a *constructed object* described by Standish [4] and to the storage block created by the L-6 language as a unit of data structure. The window registers of L-6 are its "bugs", or base registers, through which it reads and writes in the storage blocks.

c. Slot fields

A slot is defined within a memory system as consisting of one or more fields, each of which may in turn be further defined in terms of bits (bytes, cells), in a given memory. Each field has a length, defined as that number of bits (bytes, cells) sufficient to hold all data configurations which the field must accommodate; and each field has a relative position within its slot, defined in terms of other fields preceding it or in terms of relative bit (byte, cell) positions from the beginning of the slot. A field within a slot is not individually addressable but must be obtained through the window registers via the slot address; for mnemonic convenience, fields may be given names such as *address field, left link field, data field, a, flat,* etc. As a part of their definition, restrictions upon their allowable contents in terms of bit (byte, cell) contents and frequently have a specification of a null or empty flag or value which must be recognized by the system's field-interpreter operations. The fields in the model presented here are closely similar to the elements called *components* within constructed objects by Standish and are almost identical to the fields within an L-6 storage block.

d. The slot type

Each slot has a slot type, classifying it in one of a small number of formats or configurations of fields provided by the system's window registers. Slot types are similar in function to the data types, data descriptions, type declarations, or word-formats of software and hardware systems. Standish uses the term *data type* for a concept closely allied to the slot type; in L-6, a string of field definitions referring to one block sets up a structural convention with a somewhat similar function; but there is no explicit name for such a format, and it differs from our slot type in that blocks having different field patterns may be combined in one structure in L-6, whereas, in our model, all slots in a single string to be treated as a slot sequence must be of the same type and have the same minimal field configuration.

e. Slot sequences

The problem-solver, in building his storage structures, groups slots together to form larger structures that can contain lists, trees, tables, and the like, establishing access paths to a group of slots as a whole and from each slot to one or more different kinds of successors within the grouping. These larger structures, here called *slot sequences*, are not given special names by Standish or in the L-6 language other than *data structures*.

2. Basic storage operations

As stated above, a memory may be regarded as a means for relating two basic sets of primitive entities: a set of elementary storables and a set of

elementary addresses. In carrying out its task of associating storables and addresses, a memory system must be able to perform several important operations. It must be able to obtain and interpret the fields within a given slot type so that it can read and write storables in data fields and obtain successor slots in slot sequences and structures made up of sequences. Space must be found and allocated to new storables, and existing storables must be retrieved from the spaces previously allotted to them. Access must be gained to a given structure in some region of the memory from outside or from some other region. To provide the memory system with these capabilities, the following operations must exist: *field-interpreter*, *address-assignment*, *address-coupling*, *data-fetching*, and *data-storing*.

a. Field-interpreter operations

A memory must have a set of primitive operations, each designed to act upon one of the fields in a given slot type, which will find a field, transfer information bits to or from the field via an appropriate window register field, and check its contents for a null value or for some other specified set of values included in the field definition. The field-interpreter operations in this model include the operations which Standish calls *component selectors*, *predicates*, and *constructors*; they are closely equivalent to the *basic subroutines* of Knowlton's L-6 language.

b. Address-assignment operations

By means of these primitive operations, a memory system is able to relate address-expressions involving keys, locations, or both, to storable data items being put into or taken out of the memory. Given a unit of storable data, whose content or whose key is known, the memory assigns to it an access point and slot location where it may be stored if it is being entered into the memory, or where it may be found it it is being retrieved. Given an address (key, location, or both together), the memory must seek out the corresponding slot and discover what storable, if any, it contains. Different storage systems employ various kinds of address-assignment operations, and some systems make use of a number of different kinds in combination. Some of these kinds of address-assignment operations are: *arbitary assignment*, *content-derived assignment* (direct or computer), and *space control* (space listing or status flagging).

Arbitary (or user-determined assignment): The memory system employs a basic address-expression generator to create an address, or to select one of a set of allowable addresses, and then attempts to assign the address thus produced to a given storable or to find a desired storable in it. The system acts as if it had one or more location counters or address registers which are stepped and read each time an address is required for any purpose. This procedure results in the reading or storing of data in whatever *next address* the system provides. The user must remember where he has caused stored data to be placed, and the relation between a storable and its address is purely conventional or addidental. The arrangement of successive program instructions in 7094 memory, by the programmer's writing them one after another and punching them in successive cards which are then loaded in successive cells by software, may serve as an example of this familiar address-assignement procedure.

Content- (data or key) derived assignment: The memory system makes use of the storable itself, or its key, in generating an address-expression for a storage slot to hold it. Content-derived assignment may be *direct* or *computed*.

If there are enough addressable slots in the memory or some memory region to hold all the different storables and if the address-expressions which the system can generate are long enough to uniquely represent every storable or every key, then content-derived assignment can be direct, i.e., made by a simple one-to-one lookup, in a table or with the aid of a base register. If, however, the number of storables is too large or is unpredictable, or the allowable address-expressions in the system are insufficient to uniquely represent every item or if it is desirable to distribute the storables as evenly as possible over the available set of locations, addresses may be computed as some function of the storable or its key. This kind of computed, content-derived address-assignment has been called *hashing, hash-addressing, randomizing,* and *scatter-storage* by programmers and writers. A single address-expression may be assigned to more than one storable by the hashing function, so that the procedure must be supplemented in every case by other means of locating a given storable among several collisions, repeats, chains, overflow items, etc., which are all reached by one computed address.

Space control: Some memories have address-assignment operations involving the organization of the entire set of allowable storage slot-addresses into some more-or-less complete structure, or the classification of the set of slot-addresses into two or more subsets based on characteristics important to the addressing operations. The set of addresses may be partitioned into a *free* (available) subset and a *used* (nonavailable) subset, and there may be several different used subsets, based upon their use to store data or program entities, their accessibility to different users, and the need to save their contents under various circumstances. Systems of this kind are of major importance in list-processing programming languages, and for multiprogramming and multiprocessing systems. For the latter two systems, an elaborate and rigorously inclusive set of space-control operations is a vital necessity; the memory must, in effect, keep an up-to-date internal model of all regions within itself and of all events occurring within those regions that are pertinent to address-assignment at any time. Two important kinds of operations for space-control activities are *space listing* and *status flagging*:

1. Space listing involves the recording of used or free addresses, or addresses of some other important classes, as data within a special subregion in the memory, or within special fields in certain slots. A frequent technique for accomplishing space listing involves the linking together of all the slots in a region by means of the *linked address-coupling* method discussed below.

2. Status flagging causes a flag value to be stored in some field within each slot, to mark it as free, used, or belonging to some other class important to the system. Special field-interpreter operations serve to test and manipulate the flag fields and their contents.

Through the exploitation of various space-listing and status-flagging operations and one or both of the address-coupling methods below, many list-processing languages (e.g., LISP, SLIP, SNOBOL) carry out an activity commonly called *garbage-collecting*. This procedure involves periodically scanning the memory by following paths through its various linked structures on space lists and in data storage areas, and reorganizing the entire set of addresses for reclaiming slots no longer needed and making them available for new uses.

Address-coupling operations: Given a storage slot address, the memory uses its address-coupling operations to generate or obtain the address of a successor slot, so that paths of access may be followed through sequences of slots. Two principal varieties of address-coupling operations may be distinguished: *additive* and *linked*.

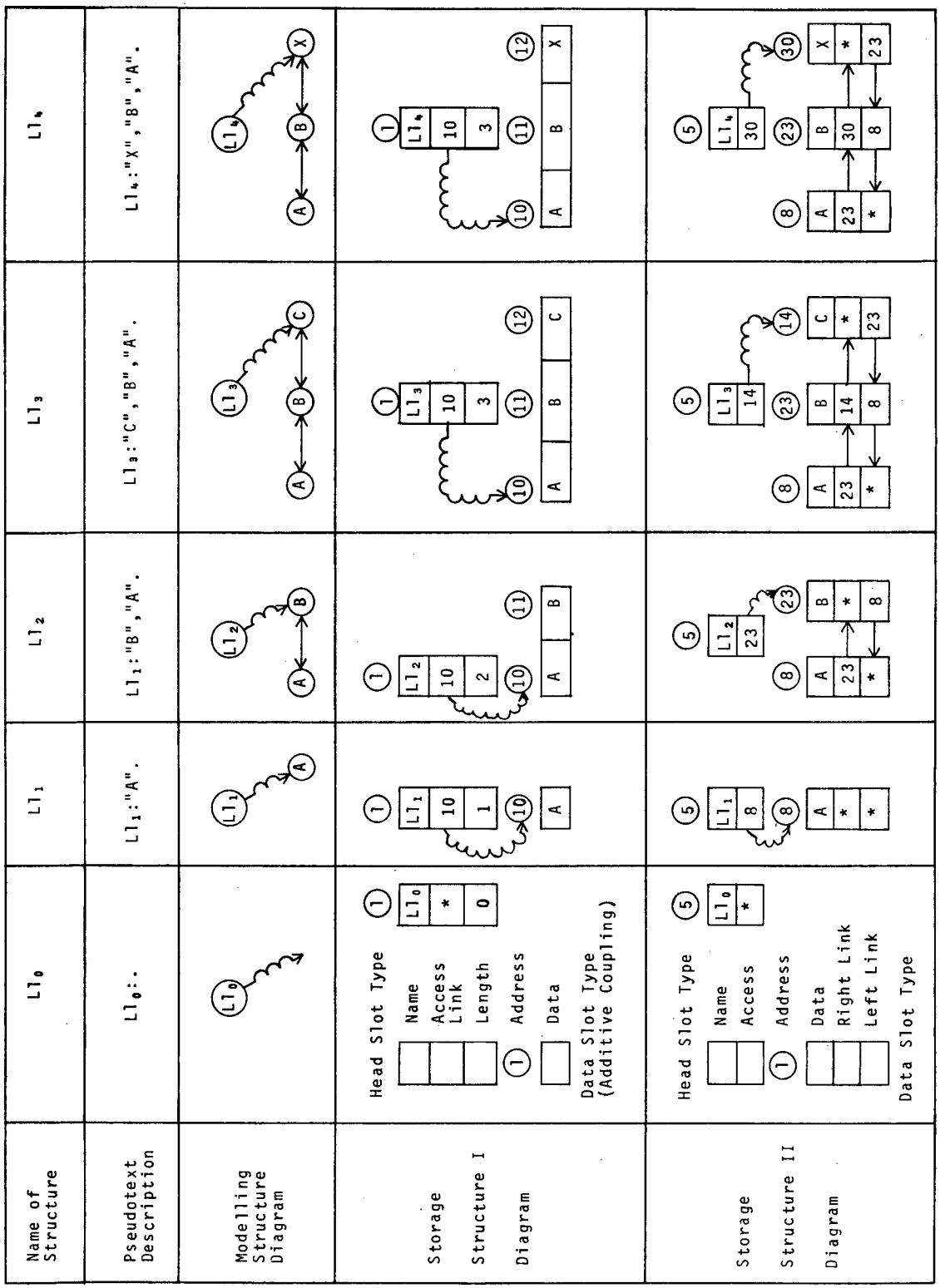

FIG. 10. A LIFO-List Modelling Structure and Two Storage Structures.

1. In additive address-coupling, the memory treats a given slot address as if it were a number like a location and creates from it a successor address by counting it up or down by some numeric value. In systems based on additive coupling, addresses are closely allied with (if not identical with) physical locations in the memory. This method of obtaining successor addresses as physical next-door neighbors is the one most familiar to programmers; Knuth calls it *sequential allocation*. The problem-solving model called an *array* earlier in this paper [Sec. III-B-1(f)], in which any element is accessible directly (in one jump) from outside and from any sibling, is typically implemented by additive address-coupling.

2. In linked address-coupling, the memory uses its field-interpreters to reach certain link fields within a slot and to obtain from them the actual stored address of a successor slot. A slot type may include a number of different link fields, providing links to left or right siblings, and reference links to other structures as well. Knuth refers to this form of address-coupling as *linked allocation*, and the designers of IPL called it *nonsequential addressing* to contrast it with the sequential additive method.

Data fetching: Given a slot address, the memory uses a field-interpreter operation to get at the data field in the slot, transfer its contents to the appropriate window register, and check it for null or disallowed values. Data fetching may be *destructive* or *nondestructive*. In destructive data fetching, after a data field has been read into the window-register field, the source slot is altered so as to erase its contents in the data field, or otherwise make it available for reuse. The operation by which this alteration is accomplished depends upon the definition of fields within the slot type, and the space-control actions employed by the memory. A status flag may be turned off or on, addresses in various link fields may be changed, or a null value may be written over the data field. Destructive fetching forms an important part of the pop-up action used by some memories in implementing LIFO or push-down lists. Nondestructive data fetching leaves the data unaltered in the slot data field.

Data storing: Data fields in slots are obtained by field-interpreter operations and storable data is transferred to them from a window-register field. As was true in the case of destructive data fetching, this storing action may involve changes to status flags or link fields when a previously free slot becomes used in some system having a form of space control. By various combinations of data fetching and storing operations, the memory system is able to copy storables from one slot to another, or from slot sequences and higher-order patterns to others, and to insert, delete, or erase simple and compound operand patterns.

IV. SOME EXAMPLES OF MODELLING STRUCTURES AND STORAGE STRUCTURES TO IMPLEMENT THEM

This Section attempts (1) to provide some illustrations for clarifying the distinction between modelling structures and storage structures and (2) to demonstrate the way in which structures of data are designed to go with a given set of operations to supply operands in a problem-solving plan. The two different graphic representations in Fig. 10 distinguish between *modelling-structure and storage-structure diagrams;* a third, narrative representation in the form of a pseudotext (an artifical language text), using a simple notation, is provided to specify operand strings or patterns of events

$$PUT(STRUCTURE_i, DATA) \longrightarrow STRUCTURE_j$$

Step 1(a). If there is no data node at end of external access path, make a data node there, store data in it, and stop.

1(b). If there is a data node at end of external access path, go to that node, and then erase the access path just used.

2. Create a new two-way sibling path and a new data node to the right of currently selected node, go to the new mode, and store data in it.

3. Create a new external access path to the node now selected, and stop.

$$TAKE(STRUCTURE_i) \longrightarrow STRUCTURE_j, DATA$$

Step 1(a). If there is no data node at end of external access path, stop with empty message "*".

1(b). If there is a data node at end of external access path, go to that node and then erase the access path just used.

2. Read out and save data from node now selected; erase the node; follow sibling path to left neighbor; erase sibling path just used.

3. Create a new external access path to the node now selected, and stop.

(a)

Step 1. $PUT(Ll_0, "A") \longrightarrow Ll_1$

2. $PUT(Ll_1, "B") \longrightarrow Ll_2$

3. $PUT(Ll_2, "C") \longrightarrow Ll_3$

4. $TAKE(Ll_3) \longrightarrow Ll_2, "C"$

5. $PUT(Ll_2, "X") \longrightarrow Ll_4$

6. $TAKE(Ll_4) \longrightarrow Ll_2, "X"$

7. $TAKE(Ll_2) \longrightarrow Ll_1, "B"$

8. $TAKE(Ll_1) \longrightarrow Ll_0, "A"$

9. $TAKE(Ll_0) \longrightarrow Ll_0, "*"$

(b)

FIG. 11. (a) Two Modelling Operations, PUT and TAKE, for the LIFO Structure Ll; (b) A "Program" Made Up of PUT and TAKE Operations.

encountered in the structures. We may say that, given a structure capable of providing a certain set of operand strings in which we are interested, a second structure is a functional equivalent of the first if it provides the same set of operand strings, regardless of any additional strings it may also provide. The symbols employed in the diagrams and in the pseudotext statements are explained below.

Consider a simple *modelling structure* of the type called a LIFO List; according to the definition in this paper of the LIFO list, it consists of a two-way linear list with a single external access point at its right-most node; let this structure be named L1. Consider also two simple *modelling operations* named PUT and TAKE, which are to be applied to data structure L1 to store one item at a time into it, and to read one item at a time from it (Fig. 11). We might also call these operations PUSHDOWN and POP-UP. The action of PUT can be described in another kind of pseudotext notation, as if it were a verb in an artificial language, and a PUT statement can be specified which can form one step in a program deriving its operands from L1 and other structures like L1. The PUT statement may have the form:

$$\text{PUT}(\text{STRUCTURE}_i, \text{DATA}) \longrightarrow \text{STRUCTURE}_j.$$

This statement shows not only the verb PUT and its two operands, a modelling structure and a data item, but also the new transformed structure resulting from the action of the statement. Subscripts i and j on the structures do not stand for ordinal or cardinal numbers in any real sense but are only two different designations for two distinct forms of L1; in some statements, STRUCTURE_i and STRUCTURE_j turn out to be identical. Two structures are identical if their diagrams and pseudotext descriptions are identical in every respect.

The action of TAKE may similarly be described in a statement:

$$\text{TAKE}(\text{STRUCTURE}_i) \longrightarrow \text{STRUCTURE}_j, \text{DATA}.$$

In this statement, the next accessible single data item is read out, or popped up, from STRUCTURE_i to produce a result structure STRUCTURE_j and the data item itself.

Finally, consider the small program made up entirely of PUT and TAKE statements [see Fig. 11(b)] and the set of more detailed specifications, each in the form of a verbal flowchart or narrative description, describing exactly how the modelling operations PUT and TAKE are to be carried out, step by step [Fig. 11(a)]. The modelling structure diagrams in Fig. 12 may be applied by a pencil-and-paper procedure following these steps "blindly", like a computer program, to produce the structures for each statement of the sequence in Fig. 11(b). This sequence begins with an empty structure $L1_0$, builds it up by successive PUT operations until it contains data items A, B, and C, then TAKES item C, PUTS a new item X, and finally TAKES all the data items untl L1 is again empty.

The initial, empty form of L1 is only a potential list; it contains only an access point from which a list may be hung, but now leading to no data nodes. This potential list $L1_0$ is shown in the modelling structure diagram of Fig. 10 as a node containing the name $L1_0$ and an access arrow which leads down to an empty spot where a data node could be. The pseudotext representation of events is $L1_0$:., where the colon stands for an external access path and the period stands for *stop*. This expression may be read "L1, access path, stop." The statement $\text{PUT}(L1_0, \text{"A"}) \longrightarrow L1_1$" describes the storing of data item A in $L1_0$. After this statement has been executed, $L1_0$ becomes $L1_1$, which is diagrammed

$PUT(STRUCTURE_i, DATA) \longrightarrow STRUCTURE_j$

Step 1(a). If head length field contains "0", get a free data-slot address from space control, store it in head access field, and go to that address.

1(b). If head length field contains something other than "0", get next address = contents of head access field + contents of head length field, and go to that address.

2. Write data into data field of slot now selected.

3. Add "1" to contents of head length field and stop.

$TAKE(STRUCTURE_i) \longrightarrow STRUCTURE_j, DATA$

Step 1(a). If head length field contains "0", stop with empty message "*".

1(b). If head length field contains something other than "0", get next address = contents of head access field + contents of head length field - 1, and go to that address.

2. Read data from data field of slot now selected.

3. Subtract "1" from contents of head length field.

4. If head length field now contains "0", write "*" into head access field, otherwise change nothing.

5. Stop.

FIG. 12. An Implementation of Modelling Operations PUT and TAKE in Storage Structure I.

with a new node containing "A" at the end of its access arrow; the description of events is "Ll_1:"A".", which may be read "Ll_1, access path, data "A", stop". If we then PUT data items "B" and "C" into Ll, it becomes a structure Ll_3, having three data nodes with two-way sibling paths, and its description becomes "Ll_3:"C","B","A".", read as "Ll_3, access path, data "C", sibling path left, data "B", sibling path left, data "A", stop.", and where comma stands for the left sibling path. (Other descriptions are also possible which use the right sibling paths, but the present one suffices for this example.)

The application of TAKE to Ll_3 produces a modelling structure indistinguishable in its diagram and in its description from Ll_2, since data item "C" has been popped up and the structure reorganized to omit "C" and all paths leading to or from it, leaving the structure just as it stood before "C" was stored. In the last step of the program diagrammed in Fig. 11(b), it can be seen that an attempt to TAKE data out of Ll_0 results in the identical structure Ll_0, and an empty or failure signal "*" instead of data. In this case, $STRUCTURE_i$ is identical with $STRUCTURE_j$. Figure 10 presents modelling structure diagrams and descriptions of all five distinct forms of Ll resulting from various steps of the "program" in Fig. 11(b), and the reader may verify the results by

$$PUT(STRUCTURE_i, DATA) \longrightarrow STRUCTURE_j$$

Step 1(a). If head access field contains "*", get a free data-slot address from space control, store that address in head access field, go to that address, and skip to Step 3.

1(b). If head access field contains something other than "*", use it as an address and go to that address.

2. Get a free data-slot address from space control, store that address in right link field of data slot now selected; go to that address.

3. Write data into data field of slot now selected.

4. Copy the contents of head access field into left link field and write "*" into right link field of slot now selected.

5. Write address of currently selected slot in head access field and stop.

$$TAKE(STRUCTURE_i) \longrightarrow STRUCTURE_j DATA$$

Step 1(a). If head access field contains "*", stop with empty message "*".

1(b). If head access field contains something other than "*", use it as an address and go to that address.

2. Read and save data from data field of slot now selected.

3. Copy the address in the left link field of this data slot into head access field.

4. Return address of this data slot to space control as free.

5. If head access field now contains "*", stop.

6. If head access field now contains something other than "*", use it as an address and go to that address.

7. Store "*" in right link field of data slot now selected, and stop.

FIG. 13. An Implementation of Modelling Operations PUT and TAKE in Storage Structure II.

stepping through the subcommands for each PUT and TAKE statement as applied to each $L1_1$. The simple operations PUT and TAKE may be extended or combined in various ways to produce higher-order operations; for instance, more than one data item can be stored or read at once by statements of the form:

$$PUT(STRUCTURE_i, n, STRUCTURE_k) \longrightarrow STRUCTURE_j$$

and

$$\text{TAKE}(\text{STRUCTURE}_i, n) \longrightarrow \text{STRUCTURE}_j, \text{STRUCTURE}_k.$$

where n specifies a count of the number of data items in STRUCTURE_k.

Two of the many possible storage structures for implementing L1 are diagrammed in Fig. 10. Figures 12 and 13 provide two possible redefinitions or translations of the detailed specifications for PUT and TAKE into specifications in terms of the basic storage operations for each of these two implementation schemes (field interpreters, address assignment and coupling operations, data-fetch and data-store operations). Storage Structure I is a hybrid linked-additive structure, since it uses linked address-coupling to provide the external access path, and additive address-coupling for sibling paths between data nodes. Storage Structure II uses linked address-coupling for all its paths. Storage slots are shown in the diagrams as rectangular, heavy outlines, and the fields within them are shown with lighter separating walls. Linked addressing paths are shown by arrows (wave-like arrows for external access and straight arrows for sibling paths), and additive addressing paths are shown by placing the slot rectangles together in a horizontal row with no intervening space or arrows. The address of each slot is enclosed in a circle above the slot rectangle. (In Fig. 10, all addresses are numbers, representing locations.) Two different slot-types are used in the diagrams for each structure: (1) a head slot-type for the names of structures, access links, and other information pertaining to the structure as a whole, and (2) a data slot-type for data items and any paths or other information pertaining to each data item individually.

The storage structure diagrams in Fig. 10 are purely abstract, in that they are not supplied with any specification of fields, etc., in terms of primitives such as bits, bytes, or cells; and this ommission was for purposes of clarity and simplicity in this illustration only. In any real case, information mapping slots and fields onto the features of a particular hardware or software memory would be included. In Storage Structure II, for example, the head slot might consist of a single 7094 36-bit active memory register with the access link in its address portion and the name in its decrement portion. The data slot might be made up of two 7094 active memory registers: the first could contain 36 bits or six 6-bit bytes of data, and the second could contain the right and left links in decrement and address, respectively.

It should be evident that the diagrams and notations presented here may be expanded and extended in a number of directions and used in a number of ways. Actual problems may be solved in terms of these representations, and the solutions may be stepped through by hand to debug them just as if they were programs. The diagrams and notations may also be used for evaluating and comparing alternative choices of models (involving different choices of basic modelling structures and modelling operations) and implementations (involving different memory characteristics); this may be done by assigning an estimate of cost (in terms of time, space, money, man hours, etc.) to each path or node of a diagram or to each symbol of a pseudotext expression. Finally, the diagrams and notations may function as clear and complete descriptions of a problem-solution method and its implementation in documenting or reporting the work.

It is hoped that the presentation in this paper will suffice at least to suggest some of the possible uses and advantages of the definitions, descriptions, and diagrams available to the computer-oriented problem-solver under the classification scheme proposed in this paper. It is also hoped that the representations for information structures which have been described will serve to aid others in designing their own tools for problem-solving.

NOTES AND REFERENCES

1. M.E. D'Imperio, "Data Structures and Their Representation in Storage," in M. Halpern and C.J. Shaw (Eds.), *Annual Review in Automatic Programming* (Pergamon Press, New York, 1969), pp. 1–75.

2. Some of the things that have been called *data structures* by various writers are: numbers, strings, vectors, character-strings, matrices, arrays, records, files, symbol-tables, instructions, tetrahedral arrays, functions, trees, lists, stacks, list-structures, associative structures, decision tables, rectangular arrays.

3. D.E. Knuth. *The Art of Computer Programming, Vol. I: Fundamental Algorithms* (Addison-Wesley, Reading, Mass., 1968), p. 309 for binary trees.

4. T.A. Standish, "A Data Definition Facility for Programming Languages," Carnegie Institute of Technology, Pittsburgh, Pa. (May 1967).

5. K.C. Knowlton, "A Programmer's Description of L6, Bell Telephone Laboratories Low-Level Linked List Language," Bell Telephone Laboratories, Murray Hill, N.J. (Mar. 1965).