ARPA ORDER NO. 189-1

MEMORANDUM
RM-6145-ARPA
APRIL 1970

A SURVEY OF DATA STRUCTURES FOR INTERACTIVE GRAPHICS

J. A. Hamilton

PREPARED FOR:
ADVANCED RESEARCH PROJECTS AGENCY

The RAND Corporation
SANTA MONICA • CALIFORNIA

52

ARPA ORDER NO. 189-1

MEMORANDUM
RM-6145-ARPA
APRIL 1970

A SURVEY OF DATA STRUCTURES FOR INTERACTIVE GRAPHICS

J. A. Hamilton

This research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 67 C 0141. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of Rand or of ARPA.

DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

The RAND Corporation
1700 MAIN ST • SANTA MONICA • CALIFORNIA • 90406

PREFACE

Current activity in interactive computer graphics has exposed several basic problems. One of the more difficult of these problems is organizing data within the computer to allow sufficient flexibility for a wide class of applications.

The data structuring and manipulation methods are of particular interest to the designers of large-scale data-retrieval systems and management-information systems including the USAF Advanced Logistics System and USAF Rome Air Development Center's work on DM-1 and follow-on systems.

This Memorandum describes and compares the salient features of several important research efforts in the field. Among these are Sketchpad, CORAL, APL, ASP, LEAP, TRAMP, AED, and L^6.

The results of the research (especially the software associative-store techniques in LEAP and TRAMP) bear on a wide variety of fields in which data relationships are important.

James Hamilton, a RAND Corporation Summer Student, is presently a student at the University of Michigan.

SUMMARY

A Survey of Data Structures for Interactive Graphics analyzes and compares several methods of organizing and manipulating data within a computer to allow sufficient flexibility for many interactive graphic applications. These structuring methods are particularly useful in large data-retrieval and information-management systems, including the USAF Advanced Logistics System. A data structure is defined in terms of:

1) A *data item*--a bit of string;
2) A *pointer*--a data item that contains address information;
3) A *structure space*--a subset of the computer's memory;
4) A *bead*--a contiguous block of machine words that is not contained in any other such block;
5) A *component*--a block of contiguous machine words contained in a bead.

A *data structure* is a collection of beads within a structure space.

A program for processing such a structure must be able to create and destroy beads, and reference data items. The first ability is provided by a storage-allocation, associative-memory system in the ALGOL-like LEAP language, by which one data item is used as the address of a block of several data items. The structure requires excess storage, but it keeps related information together and uses secondary storage efficiently. Thus, LEAP structures are valuable as relational data stores for large information retrieval problems, but they are less so for the normally small interactive graphics problems.

The referencing ability provided by several procedural languages, including AED, BCPL, L^6, and PL/1, enables the

programmer to design his own structures. However, several predesigned structures, whose components are arranged into *rings*, are useful in interactive graphics--e.g., Sketchpad or CORAL.

How is graphical information represented in a data structure? In a Sketchpad ring, a drawing is a collection of entities (e.g., points, lines, and circles), each represented by a bead. Line drawings in TRAMP (structurally similar to LEAP) are shown by an associative-memory structure--a tree whose nodes represent positions, lines, points, and pictures. The Sketchpad display *includes* topological information (i.e., the connections between parts) but CSMP represents topology *exclusively*.

Graphical input is usually related directly to a data-display structure, then indirectly to the problem structure because 1) most hardware provides input via light pen, thus automatically relating input to the display structure, and 2) transformations have been applied to the problem structure so that only the display structure knows what is being shown.

CONTENTS

FIGURES

I. INTRODUCTION

This Memorandum provides a useful and reasonably thorough description and analysis of data structures for interactive graphics. Familiarity with the field is assumed, and no attempt is made to define "interactive graphics" (dealt with only in Sec. III). The goal is to *describe* and *compare* salient features of the most important research efforts in data structures.

Comparison requires descriptions in a common terminology so that essential similarities and differences become apparent. Many properties attributed to data structures are really properties of description, particularly when pictorial. Good descriptions are valuable to programmers using data-structure facilities, but should be independent of the structures themselves (and therefore lie outside the purview of this paper). The most important description of any data structure is the language used to reference and modify it. (As discussed in Sec. IV, many linguistic constructs are independent of the structure to which they are applied.)

The information in this Memorandum derives almost entirely from the published literature. Unfortunately, most authors mention data structure only briefly. Thus, potentially interesting features of many systems are not adequately enough described for inclusion. The primary basis for discussion of any particular work is its author's claim of applicability to interactive graphics. This claim is rarely supported, because the terms are not well defined. But some agreement exists among authors in the field as to which papers are of primary importance, and all of these are considered.

II. STRUCTURES

Terms used in the description of data structures are: *structure space*, *bead*, *component*, *data item*, and *pointer*. Although these correspond to familiar objects, here they are generalized, and the reader is cautioned against assuming properties not explicitly stated in the following:

A *data item* is any bit string that represents an object of interest. A data item typically represents an integer, a floating-point number, or a character string. A *pointer* is a data item that contains address information. Specifically, if f(x) is a function whose value is a machine address, then any data item in the domain of f is a pointer. In any given structure, there is only one such function; but if there is more than one, pointers are referred to by the name of the function.

The most common function is

$$f(x) = x + \text{constant}$$

with the constant generally zero. The choice of a function may be machine-dependent; it is always determined on the basis of efficiency. The furction may have two arguments, in which case both are pointers. This generality is needed to describe the so-called software-associative memory structures. The range of the function is generally restricted to some subset of the machine's available memory. This subset, which may include all types of storage as well as core, is called the *structure space*. Restricting the range restricts the domain.

Bead is a term borrowed from the Automated Engineering Design (AED) Project [1]. It means n-component element, but is defined here as a contiguous block of machine words that is not contained in any other such block (except the structure space).

A *component* is a block of contiguous machine words contained in a bead, and is also the smallest object that can be pointed to by a pointer. A component contains at least one data item, which may be a pointer or any other type. It may contain any number of pointers, and is not fixed in size or internal structure. A bead may contain any number of components of varying size and internal structure. Components are ordered within a bead, and may be referred to by number.

A standard pictorial representation for structures is used: beads are shown as rectangles, components are separated by horizontal lines, and data items within components are separated by vertical lines. If a data item is not a pointer, it is blank or contains a name for the data item. If it is a pointer, it contains a dot, which is the tail of an arrow, that points to the component referenced by the pointer (see Fig. 1).

A data structure is defined as a collection of beads within a structure space. A program for processing such a structure requires the basic abilities to create and destroy beads, and to reference data items. The first ability is provided by a storage-allocation system (discussed in Sec. V).

The referencing ability provided by several procedural languages, including AED, BCPL, L^6, and PL/1 [1-4] (discussed in Sec. IV), gives the programmer the ability to generate and manipulate structures of his own design. Whether the necessity of designing one's own structure is a useless burden remains to be answered. Large classes of problems exist that have considerable structural similarity, and data structures have been designed that match such problem structures. But there remains the necessity of learning the intricacies of these structures, as well as those of maintaining and interfacing with the associated software.

However, several structures have been designed that are useful in interactive graphics. Designing a data structure

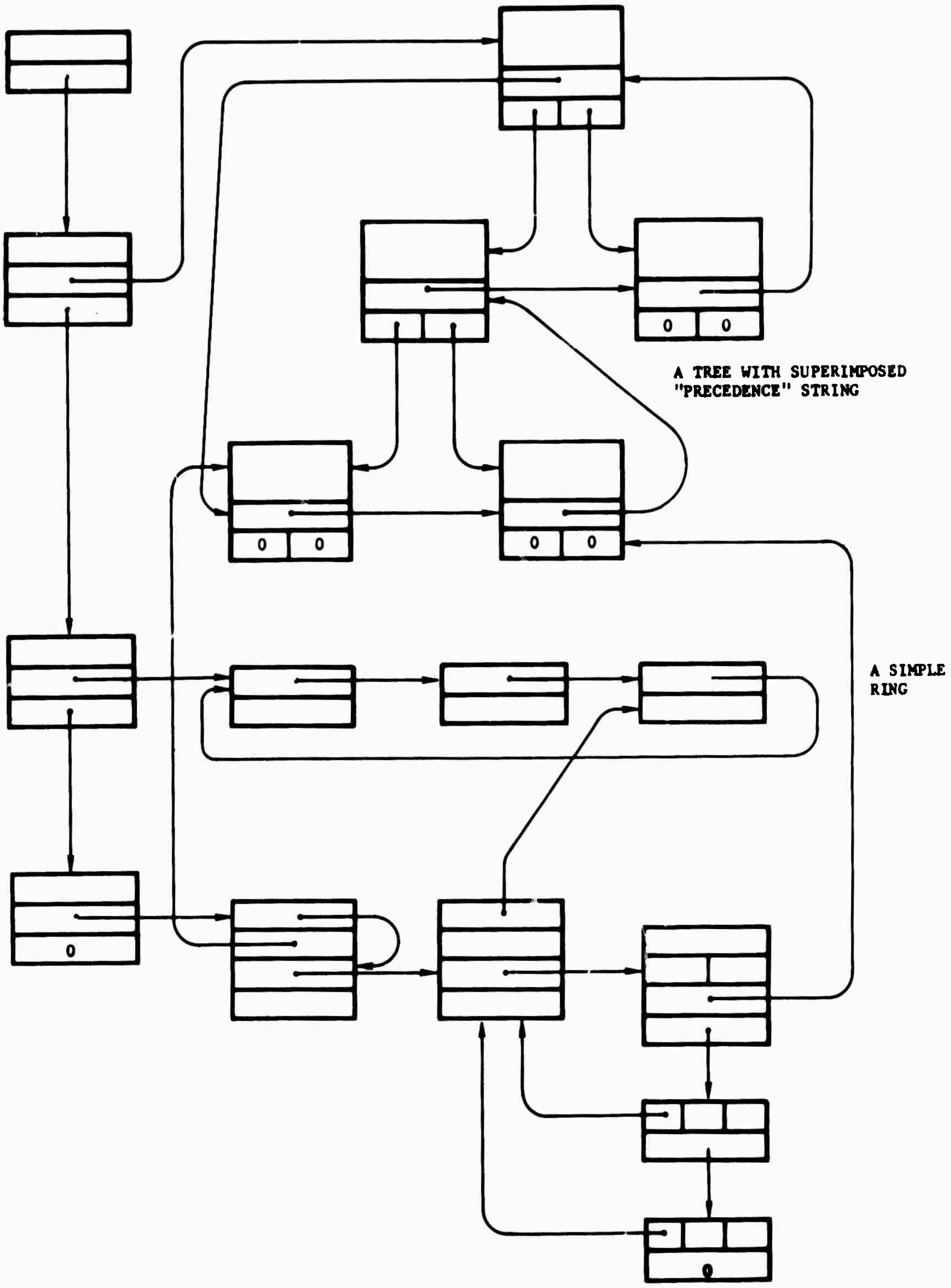


Fig. 1--A Generalized Structure

consists of making restrictions on the internal structure of beads and components. These restrictions take the form of declarations about classes of beads and their components. These declarations allow the construction of primitive functions that manipulate the structure in predefined ways. The terms used below were introduced by the original authors.

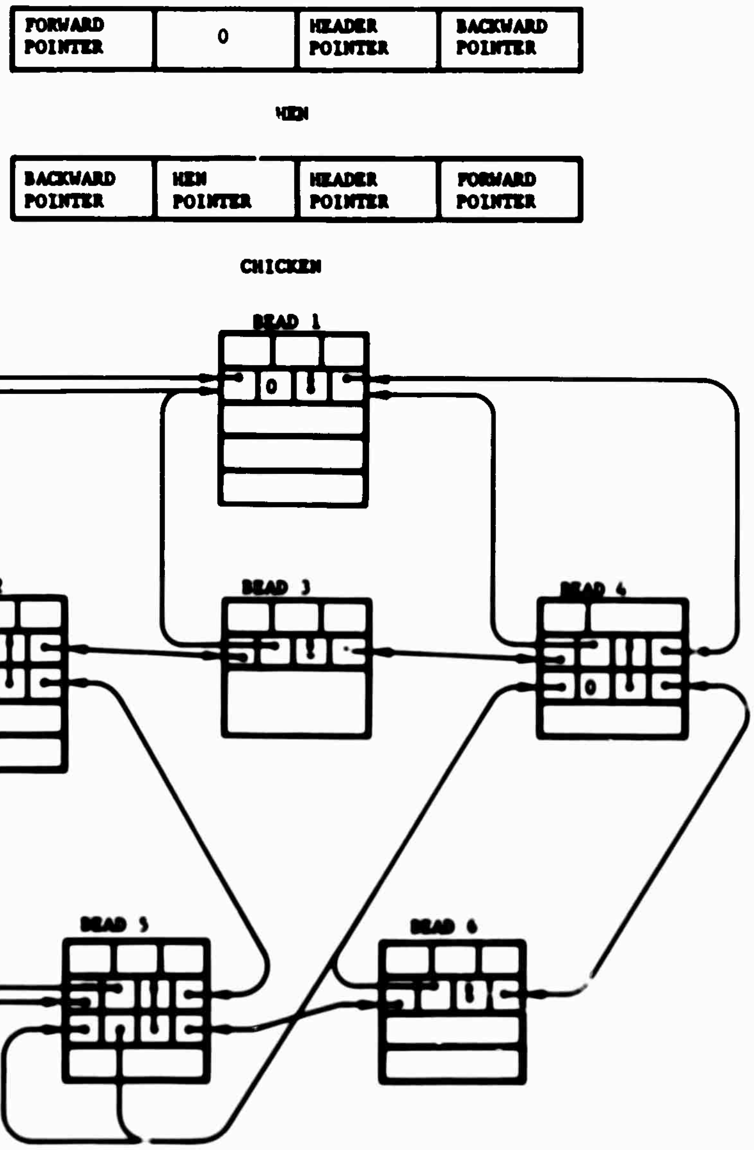
THE RING STRUCTURES

Perhaps the best-known of the structures are the Sketchpad rings [5]. Sketchpad contains four types of components: *header components*, *value components*, *hens*, and *chickens*. Every bead has one header component, its first component, followed by an arbitrary number of hens and chickens, collectively called ring components. Following the ring components are an arbitrary number of value components.

Value components contain one data item that is not a pointer and has no structural interpretation. A header component contains three data items: 1) the number of components in the bead, 2) the number of ring components, and 3) the type, used for identification by the processing program.

The ring components are the interesting ones. A chicken contains four pointers: the hen pointer, the forward pointer, the backward pointer, and the header pointer. A hen contains the same four pointers. However, the hen pointer is given a value not in the domain of the pointer function (viz., zero), to distinguish hens from chickens.

As shown in Fig. 2, the ring components are organized into *rings*. Each ring contains exactly one hen, and an arbitrary number (possibly zero) cf chickens. The ring components are ordered in the ring, with forward pointer pointing to the next component; backward pointer pointing to the preceding component; hen pointer pointing to the hen; and header pointer pointing to the header component of its bead.



RINGS: BEADS 1, 2, 3, 4
2, 5
4, 5, 6

Fig. 2--A Sketchpad Structure

The pointer function in Sketchpad is the standard

$$f(x) = x + c \ ,$$

with non-zero c that allows x to be smaller than the full address space of the machine. The header pointer, however, uses a different function, viz.

$$f(x) = x + \text{address of } x \ .$$

The above representation is structurally irrelevant, but important for efficiency since it allows x to be a very small negative number, needing only 4 or 5 bits.

Most follow-ups to the original Sketchpad work [6-11] have been either inadequately described or not significantly different. One nearly identical structure is CORAL (Class Oriented Ring Associative Language) [7], in which the backward pointer and hen pointer are replaced by one pointer plus a one-bit data item that tells whether it is a backward pointer or a hen pointer. CORAL rings are built with backward pointers and hen pointers alternating. Also, the header pointer is eliminated without mention. The author can only assume that an additional one-bit data item is used to mark the header component so that it may be found by a short search.

These differences primarily affect efficiency, cutting in half the space requirement of Sketchpad ring components, which requires two machine words. CORAL achieves this improvement at the cost of a small increase in processing time due to the necessity of moving, for example, through more than one component to reach the hen. This cost is dependent on problem characteristics: if these are such that rings are generally short (less than 10 components), then hen and backward pointers could be eliminated without great loss. CORAL is considered a good compromise.

A more important difference is the addition in CORAL of a small bead called a *nub*. When a bead is created, it contains a fixed number of ring components that cannot be changed because of storage-allocation methods. A bead is made a member of a ring by adjusting one or more of its ring components to point to the appropriate ring members. However, it may happen that all of the ring components are used; in the majority of cases, this will never happen because of problem-structure constraints. In a Sketchpad structure one must be careful of this possibility when designing bead types; but in CORAL the "nub" overcomes this problem. A nub is a bead containing two ring components (used as shown in Fig. 3), which serves as a general example of a CORAL structure.

Both Sketchpad and CORAL structures are manipulated by a set of primitive functions discussed in Sec. IV). Another language, APL (Associative Processing Language), extends PL/1 and provides structural capabilities identical to CORAL, with the exception of a data item called an "associative data attribute" [12]. Although its implementation is not described (and this author is unwilling to guess), the language itself is discussed in Sec. IV.

Another ring-type structure, ASP (Associative Structure Package) [13], is logically equivalent to CORAL but with the CORAL nub carried to its extreme. In CORAL terms, the ASP structure is obtained by restricting the ring components in each normal bead (called an element in ASP) to two (one hen and one chicken). A second bead (a ringstart) expands hens in the same way that the CORAL nub (an associator in ASP) expands chickens.

Although this description of ASP differs from the original, it permits a direct comparison that indicates ASP is equivalent to CORAL from the viewpoint of the programmer equipped with the same set of primitive functions--except that he no longer has to determine the number of ring components in a bead. Processing of an ASP structure is slower

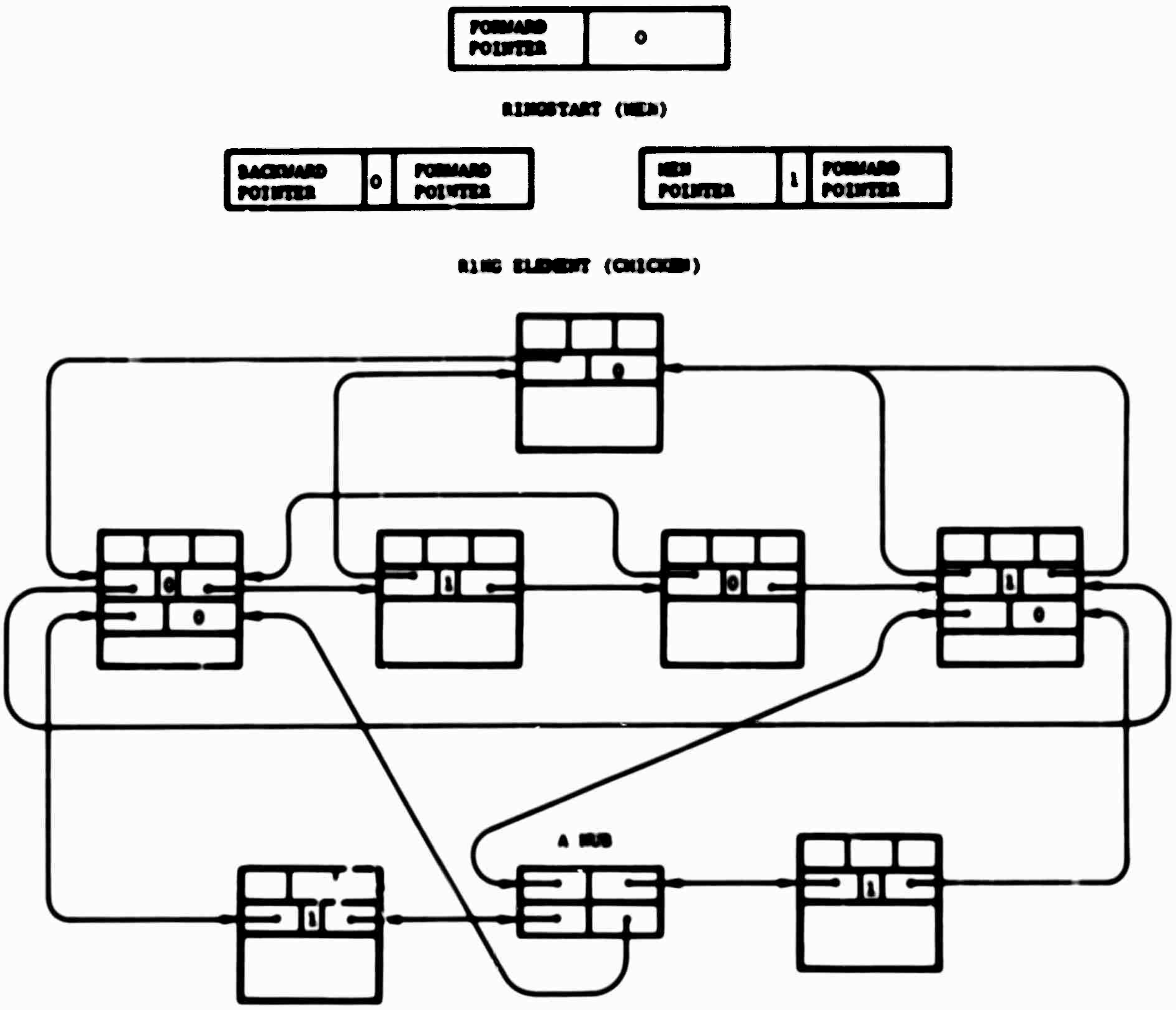


Fig. 3--A CORAL Structure

than CORAL, but part of the ASP work is the development of an elegant pictorial description for structures. Although the pictorial description specifically represents ASP structures it could also be used for other structures.

THE SOFTWARE ASSOCIATIVE MEMORY APPROACH

This section concludes with a description of a software-simulated associative memory. Several systems, nearly structurally equivalent, use the associative memory approach [14-17]. The system described here is part of the LEAP language [14-15].

LEAP is designed to store triples. A *triple* is an ordered set of three data items--attribute, object, and value. Because the structural description alone might be meaningless, retrieval requests that LEAP is intended to support are discussed before describing the representation of triples in terms of beads and components. A LEAP structure is a collection of triples. Besides determining the presence or absence of a particular triple, LEAP can retrieve the following sets of triples:

1) All triples having attribute A and object O;
2) All triples having object O and value V;
3) All triples having attribute A and value V;
4) All triples having attribute A;
5) All triples having object O;
6) All triples having value V.

Retrieval is accomplished by providing three separate structure spaces: attribute space, object space, and value space. There are two retrieval methods: 1) for the first three types and 2) for the second three. Each triple is stored in all three structure spaces. To retrieve a set of type 1), for example, method 1 is applied to the attribute space.

Each space is identical in structure, the only difference being a permutation of the triple before it is stored. The attribute space stores the triple (A, O, V), the object space (O, V, A), and the value space (V, A, O).

In the attribute space, each data item in a triple has three pointer functions. The first is the *item-function* that yields the address of the external description (external to the data structure and its processor in the sense that it is interpreted elsewhere). This external description is usually outside the structure space, but also may be inside (i.e., another triple).

The second function is the *hash-function* that requires two arguments, the attribute and the object of a triple. These arguments may be two pairs such that

$$f(A1,O1) = f(A2,O2) .$$

This situation, called a conflict, must be resolved. In Sec. V the hash function, which yields the address of a hash bead (see below), and implements retrieval method 1, is further discussed.

The third pointer function--the *use-function*--applies to the attribute (for the attribute space), and yields the address of a use-header bead (p. 13). It implements retrieval method 2.

A fourth and final pointer function--the *link-function*--applies to other data items found in the structure space, not to triple items. It applies to pointers called links and yields the address of any bead.

Two bead types have been mentioned above. All beads listed below contain exactly one component (the distinction is not relevant here) and one data item that indicates the type (see Fig. 4):

1) The hash bead contains five data items:
   a) Attribute;
   b) Object--for comparison with originals to resolve conflicts;

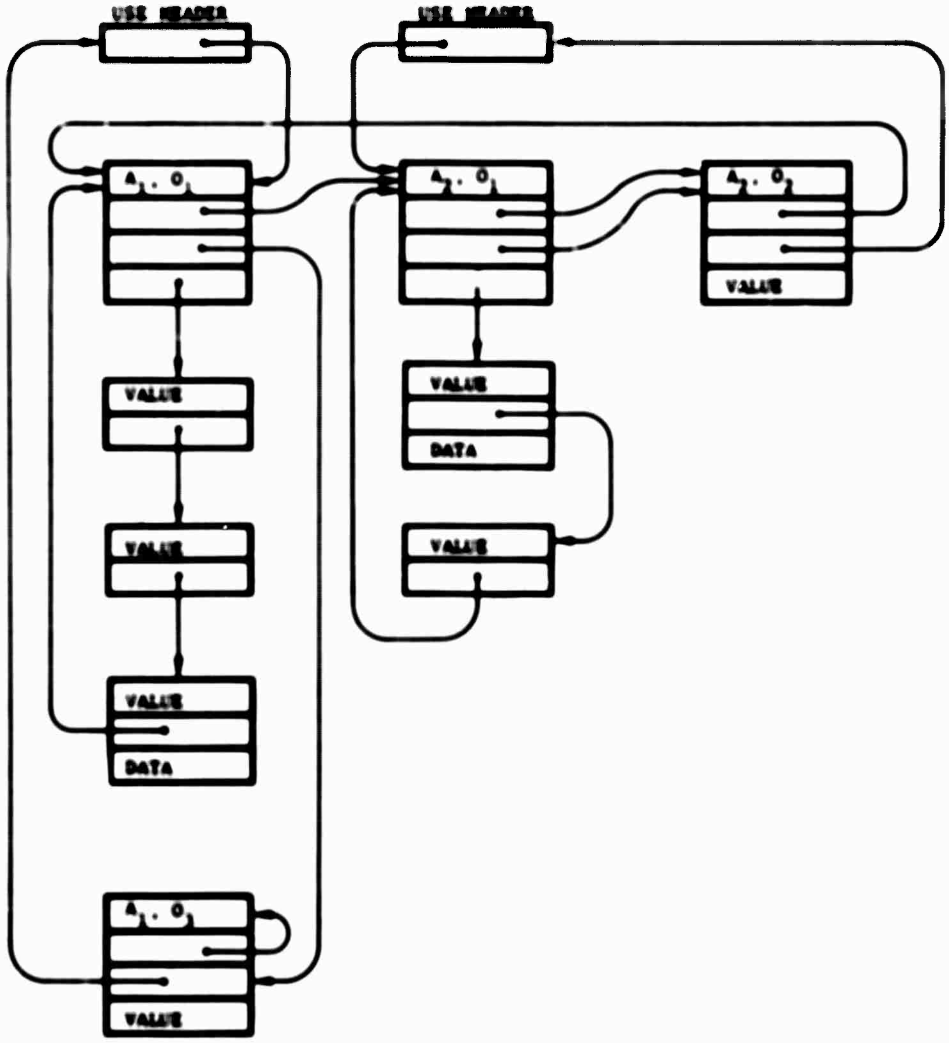


Fig. 4--A LEAP Structure

c) Use link--see use header;

d) Conflict link--in conflicts, additional hash beads are chained to the first. To retrieve, one searches the list for the bead with the correct attribute and object;

e) Value, or value link--if there is more than one value for the same attribute-object pair, additional value beads are chained to the hash bead.

2) Value bead--two data items:

a) Value;

b) Value link or zero.

3) The use header contains one data item that is a link to the first hash bead in a use ring. There is one use ring for each attribute that collects all triples having that attribute. It is accessed by applying the use-function to the attribute.

Hash beads and value beads may have one additional non-pointer data item if the triple is used as an ITEM (pointed to by a triple member via the item-function).

Now, compare this organization to other structures, considering only the attribute space. Let attributes correspond to components, objects to beads, and values to data items. Although only one of many ways of viewing a LEAP structure, the entire generalized structuring ability is there, plus much more. Of course, one must assume the burden of designing one's own structure, which is even worse in the simulation of associative memories, because of its greater complexity and its relative unfamiliarity.

In any case, there are several differences between generalized structures and a LEAP structure with the above correspondences: beads (objects) are of varying length; the use ring automatically ties together objects that have the same attribute; the object and value spaces are intentionally redundant (although it is clear that a ring structure could be duplicated in a smiliar way by using permutations). Note that a ring structure can store exactly the information required by the problem, whereas a LEAP structure stores the same information in several forms for reasons of search efficiency.

Finally, in considering the costs involved, the quantity of storage required is vastly increased, so much so that all but the smallest structures require secondary storage [18]. On the other hand, LEAP structures lend themselves to a rather efficient use of secondary storage; in fact, speed is relatively independent of the size of the structure, which makes it look good for large problems. But it is not clear that graphics problems generate structures large enough to require secondary storage. As for processing speed, assuming the referenced triple is in core, the hash function requires three or four instructions, and conflict resolution even more, which means that referencing is probably less than a factor of ten slower.

LEAP and similar structures clearly have great value as relational data stores for information retrieval problems, but their usefulness in interactive graphics is not clear.

III. GRAPHICS APPLICATIONS

This section deals with two problems that are really the same problem: 1) how to represent graphical information in a data structure; and 2) how to relieve the programmer of the job of display generation and management. A solution to the first problem also solves the second--since once we have a fixed data-structure representation, we can build a processor to generate the display from this structure. The real problem is to find a representation amenable to both display structure and problem structure. The several possibilities described below are followed by a brief general discussion of display generation.

RING STRUCTURE REPRESENTATIONS FOR LINE DRAWINGS

The Sketchpad representation of line drawings has been the basis for succeeding work in this field, and little that is new has been added. In Sketchpad, a drawing is a collection of *entities*, each of which is represented by a bead. This discussion concerns five types of entities (although there are many more in the actual implementation): points, lines, circle arcs, pictures, and instances. The first three should be obvious. These entities constitute a nearly minimal selection of building blocks; clearly other common entities (e.g., general conics) could be added to the list, but tiic display generator must be aware of them. A picture entity collects in its picture ring the lines, arcs, and instances that make up the picture. It also collects all instances of that picture in an instance ring. An instance references a picture by appearing in its instance ring, and the referenced picture is a subpicture of the picture in whose picture ring the instance appears. Thus, a display is a hierarchy with a picture at the top, containing instances of other pictures, which in turn may contain instances, etc.

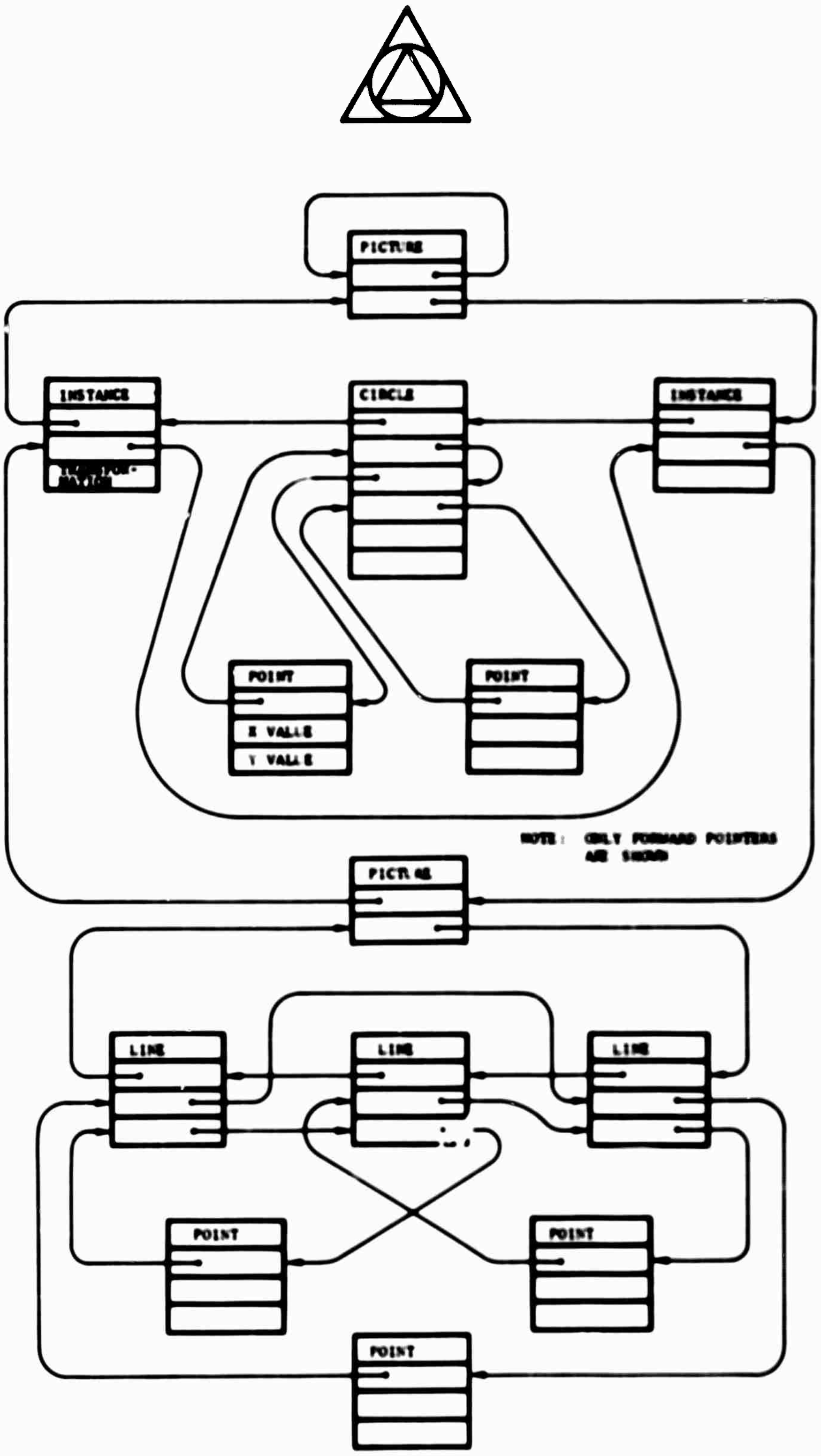
Figure 5 shows a brief description of the internal structure of the entities. Each bead contains the following in addition to a header component:

1) A line contains three chickens--one for the picture ring and two for the end points.
2) A circle arc contains four chickens--one for the picturє ring and three for end points and center. It also has two value components, angle of arc and radius.
3) A point contains one chicken for the picture ring--one hen to collect the lines and circles referencing this point (the ring goes through the chickens listed above), and two values giving the coordinates.
4) An instance contains two chickens--one each for the picture ring of the picture containing it and the instance ring of the picture it references. It also contains four values, giving a transformation (rotation, translation, and scale) to be applied to the picture it references.
5) A picture contains two hens--one for the picture ring and one for the instance ring.

The arrangement of hens and chickens in this structure introduces a set of dependencies that are consistent with the deletion mechanism provided for Sketchpad structures. A bead containing a chicken is said to depend upon the bead containing the hen for that chicken. When an entity is deleted, all dependent entities must be deleted as well (e.g., a line must have end points). Deletion mechanisms are discussed below (Sec. IV).

LINE DRAWING IN TRAMP

For completeness, an example of line-drawing representation in an associative-memory structure is included. Implemented in TRAMP [15] (structurally almost identical to LEAP),



NOTE: ONLY FORWARD POINTERS ARE SHOWN

Fig. 5--A Sketchpad or CORAL Line Drawing

a display is represented by a tree whose nodes represent positions, lines, points, and pictures. The tree structure is a set of triples that is distinguished by attribute CLAS. The value of the triple is a node in the tree, and the object of the triple is a daughter of that node.

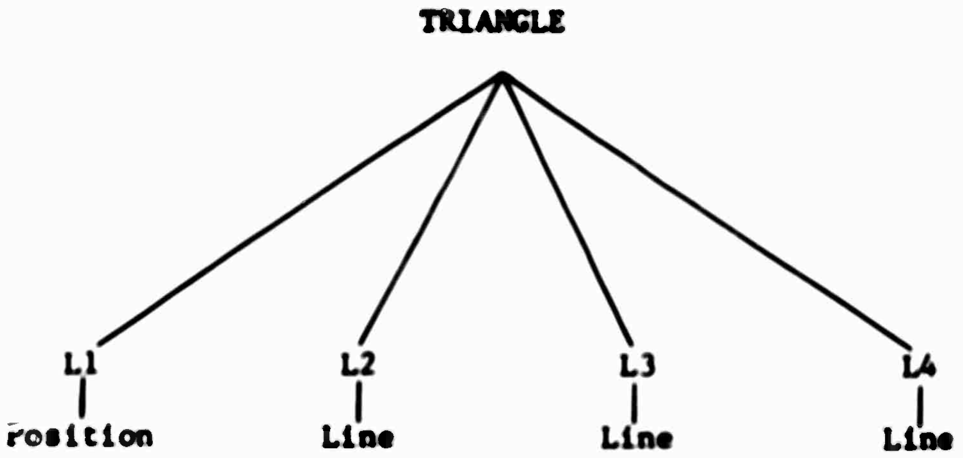
A terminal node determines the type of entity, viz., position, line, or point. These correspond to display orders: i.e., position means invisible line, line means visible line, and point means display a point. The branches must thus be ordered, from left to right, in the sequence in which they are to be displayed. Thus a line, in the Sketchpad sense, is represented by two successive nodes--either position-line or line-line.

The single immediate predecessor of each terminal node is the name of that entity, and is also the object of a pair of triples whose values are coordinates and whose attribute is COOR. All further predecessors are names of pictures: i.e., names of collections of positions, lines, points, and other pictures. Figure 6 shows a triangle using this scheme.

REPRESENTATION OF TOPOLOGICAL NETWORKS

Many application programs are not concerned with the geometry of a display, but rather with the topology: i.e., the connections between parts. This is particularly true of such applications as electrical or logical network design.

The Sketchpad representation does include topological information as part of its constraint satisfaction mechanism (but in a complex and unnatural way not described here). Another system, CSMP (Continuous Systems Modeling Program) [19] has been designed to represent only the topological structure. The representation consists of a set of entities (e.g., resistors and capacitors) that must be made known to the display generator in some undescribed way. Each entity



The Store of Triples

CLAS, L1, TRIANGLE	COOR, L1, X1
CLAS, L2, TRIANGLE	COOR, L1, Y1
CLAS, L3, TRIANGLE	COOR, L2, X2
CLAS, L4, TRIANGLE	COOR, L2, Y2
CLAS, POSITION, L1	COOR, L3, X3
CLAS, LINE, L2	COOR, L3, Y3
CLAS, LINE, L3	COOR, L4, X4
CLAS, LINE, L4	COOR, L4, Y4

Fig. 6--A TRAMP Line Drawing

has an arbitrary number of *attacher points*, which correspond to ring components of the bead representing the entity. Each node in the topological structure is represented by a ring through the attacher points at that node (see Fig. 7).

Although this system handles a large class of computer-aided design problems in a simple way, it is unable to deal with geometric considerations. However, it clearly could be merged with the Sketchpad representation. What is needed is the addition of ring components for attacher points to instances.

DISPLAY GENERATION

Consider the problems of translating a problem structure into a program for controlling a display. In general, this display program will be a sequence of display orders (codes that control beam movement, etc.) transmitted through a data channel, under control of a channel program. This channel program has two instructions: one sends a block of display orders, giving a starting location and word count; the other is a transfer in channel instruction, which allows a transfer of control within the channel program. These channel instructions may be viewed as pointers that give structure to the display program. The problem is how to generate the display structure from the problem structure.

Cotton and Greatorex [9] describe a system that builds a display structure from a problem structure very similar to Sketchpad. The display structure is very similar to the problem structure. Mainly, ring pointers are replaced by transfer in channel commands, and values by display orders, in such a way that one merely starts the channel program at a picture block; the display processor transfers from block to block finally ending, once again, at the picture block. The differences are that instances are expanded and point blocks are removed in favor of including coordinates in

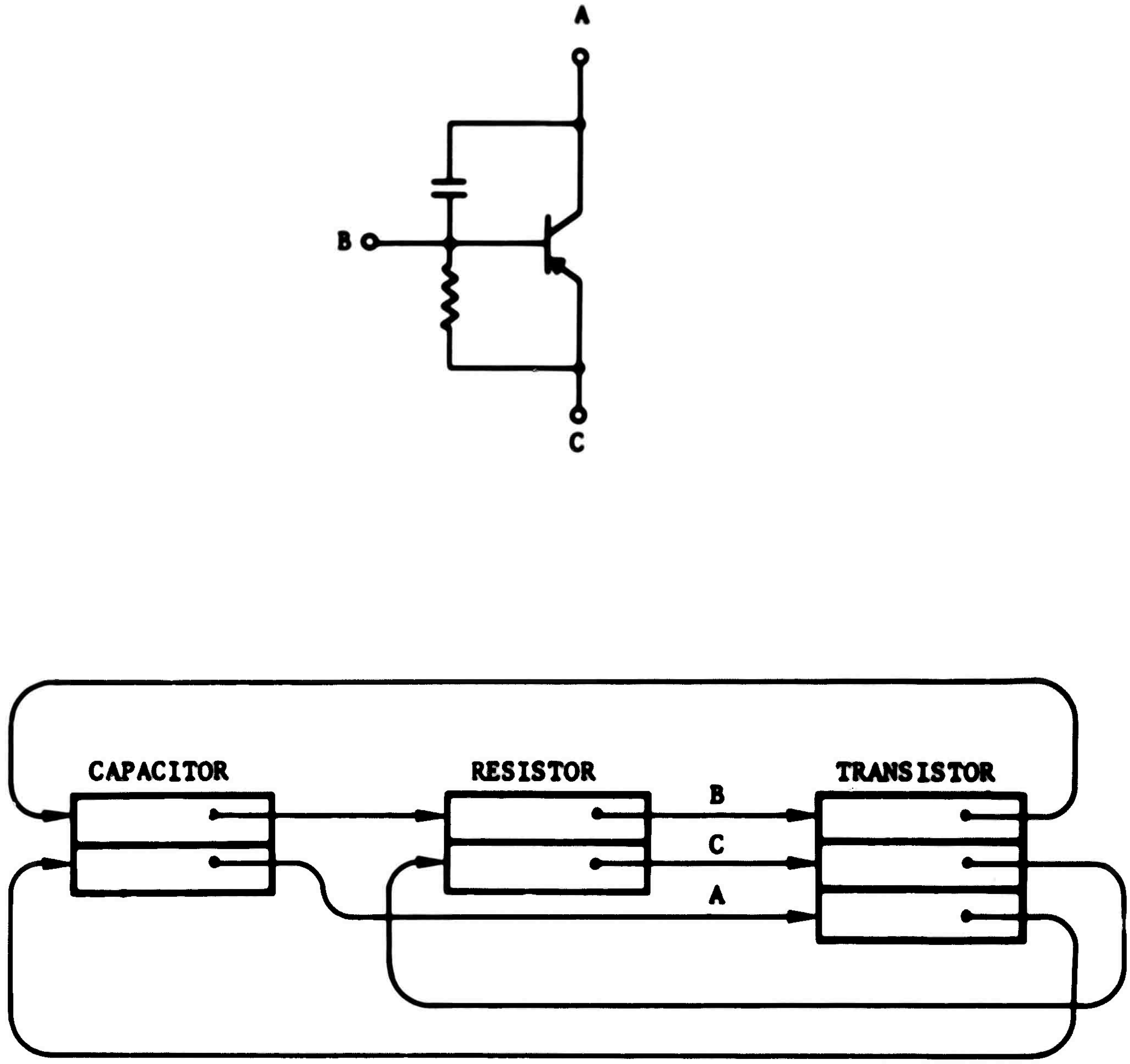


Fig. 7--The CSMP Representation

line and circle blocks. (Actually point blocks are not in the problem structure, in the Cotton-Greatorex system.) Figure 8 shows an example of a display structure for the Sketchpad structure of Fig. 5 (p. 17).

Is it reasonable to merge the problem structure and the display structure since they have many similaritics? The difficulty develops with the various transformations that must be applied in generating the display structure. One of these is the translation from problem coordinates to display coordinates, or from values to display orders. If the display space is smaller than the problem space, scissoring (the removal of objects that lie outside the display image) may be involved. In addition, the transformations (rotation, translation, and scaling) specified by instances require that a separate copy of a picture be made for each instance.

Display generation in TRAMP is simple. Terminal node types and coordinates are transmitted in order, from left to right, in the tree.

No other display generation methods have been described, because work has depended on the application program to generate the display through a sequence of calls to primitive-display functions. These functions often include subpicture-definition capabilities, providing the ability to display or delete previously generated subpictures. Therefore, these functions are really a set of data-structure primitives for manipulating the display structure, requiring the programmer to learn two sets of primitives.

INTERACTIVE ASPECTS

A question of primary interest is how to relate graphical input to a data structure. The choice is between relating input directly either to the problem structure or to the display structure (and then to the problem structure). The second alternative has invariably been chosen for two reasons:

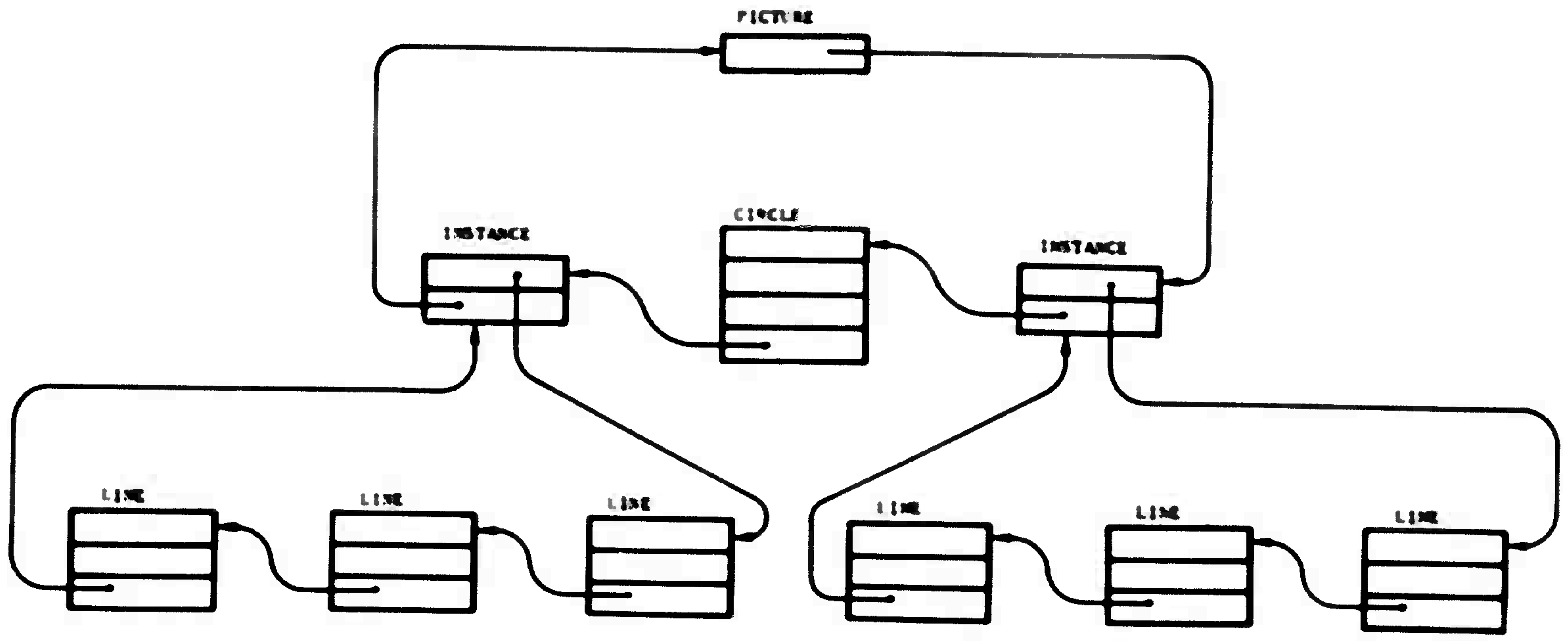


Fig. 8--A Display Structure for the Line Drawing of Fig. 5

1) most hardware provides input via light pen, in which case the hardware automatically relates input to display structure; and 2) transformations have been applied to the problem structure so that only the display structure knows what is being displayed.

Therefore, a means of relating display structure back to problem structure is required (assuming input can be related to the display structure). In a system with display primitives, the display processor is unaware of the problem structure. To circumvent this problem, display primitives generally return identifiers with which the displayed items will be referenced in the future. The application program is then forced to cross-reference these identifiers with its data structure, an awkward solution at best.

With the Cotton-Greatorex structure [9], however, the solution is simple. Each entity in the display structure is put into a ring headed by the entity from which it was generated in the problem structure. The TRAMP structure [15] has an equally simple solution. Each entity name is made the object of a triple whose attribute is HIST and whose value is the location in the display structure.

CSMP [19] has an interesting solution: the display is not generated by the application program, but by the user sitting at a console in conversation with the display processor, which is then responsible for generating the problem structure from the display structure, rather than vice versa. CSMP does this by dividing the display surface into 64 small squares and then searching the display structure for groups of attacher points lying in the same square. Many questions relating to the generality of the display processor are left unanswered.

IV. LANGUAGES

The following discussion of languages has two goals: 1) the description of primitive operations for data structure management; and 2) the presentation of the syntax used to specify these primitives. The primitives are discussed in four groups, along with the appropriate syntactic forms. The first is simple references, or retrieval of data items from generalized structures. The section on references deals with the specific structures defined in CORAL and LEAP, followed by a description of the updating of structures; i.e., the creation, insertion, and deletion of beads. The section concludes with the more complex searching and iteration operations, once again applied to the specific ring structures and associative memory structures.

SIMPLE REFERENCES

The problem with data-structure processing in a typical algebraic language (e.g., FORTRAN) is that one name is bound to two values--a location and its contents--but the program controls only the contents.

There are many ways around this problem. Describing them requires introducing some terminology, borrowed this time from the BCPL language [2]. An *expression* is a sequence of variables, manifest constants, parentheses, plus and minus, and unary operator *rv*. A *variable* is a name with two values (as in FORTRAN). A *manifest constant* is a name with only one value. This name may be a number (e.g., 5) or an arbitrary identifier that has become a manifest constant in some undefined way. The important point is that its value is known to the language processor (compiler). The *location*, or left-hand value of a variable is a manifest constant. Parentheses, plus, and minus are used in the usual way.

An expression is evaluated to yield a single value. The value of a variable in an expression is the right-hand value, or contents of the location determined by the left-hand value. An indirection operator, rv, applied to an expression, yields the contents of the location determined by the value of the expression.

We now can describe, in terms of the above, the referents provided by various languages which are essentially a small subset of BCPL. Consider first the referent

```
A(B) ≡ rv (A + B) .
```

With no restrictions on A or B, this is the meaning of A(B) in BCPL. But most languages restrict A to be a manifest constant. In FORTRAN and its relatives, A is a manifest constant whose value is assigned by the compiler. That is what is meant by the statement

```
DIMENSION A(...) .
```

In AED [1], the value of A is assigned by the programmer, using the non-executable statement

```
A $=$ value .
```

It must still be a manifest constant however, and is required to be declared so by

```
COMPONENT A .
```

In AED, B is viewed as a pointer to a bead, and A is the offset of a particular component. Thus, A(B) references a component of a bead.

Next consider A(B(C)). As one might expect, in most languages this is rv(A + rv(B + C)). In AED, however, it can also be rv(A + B + C), provided A is a COMPONENT, and B is another manifest constant called a SUBELEMENT. If B is also a COMPONENT, the other meaning is used. A(B + C) also means rv(A + B + C). In BCPL only, one also may have A(B)(C), meaning rv(rv(A + B) + C).

L^6 [3] is a low-level language in which variables, called *bugs*, are identified by single letters. Reference is accomplished by concatenation; i.e., AB means rv(A + B), where A is a bug name and B is a manifest constant declared by defining the internal structure of a bead. The declaration is made by using the instruction (i DB j k); i is the value of B, and j and k are bit numbers that determine shifting and masking to be performed after the storage access. ABC means rv(rv(A + B) + C), and such strings can be arbitrarily long, the same as BCPL, without parentheses.

In PL/1, a bead--a PL/1 structure in which the items at level two are components--is declared by:

```
DECLARE 1 ⟨structure name⟩ CONTROLLED ,
        2 ⟨component name⟩ ⟨attributes⟩ ,
        2 ⟨component name⟩ ⟨attributes⟩ ,
          .
          .
          .
```

If A is a component name, and B a variable with attribute POINTER, then B → A is equivalent to rv(A + B), where A is a manifest constant whose value is assigned by the compiler to be its relative position in the structure. A component may be a substructure, via

```
2 A , 3 C , 3 D ...
```

In this case, B → A.C means rv(A + B + C), where both A and C are manifest constants. A componen1 also may be an array in the FORTRAN sense, and B → A(C) also means rv(A + B + C). B → A → C means rv(rv(A + B) + C). If a structure containing component A is declared CONTROLLED (B), then A means B → A. Note, however, that pointer arithmetic is illegal in PL/1, but not in BCPL, AED, or L^6. BCPL is the simplest and most general of these languages; in addition, its data items are completely typeless, whereas PL/1 provides a complex array of types, and AED uses the common ALGOL types.

SET REFERENCES

The above languages allow one to reference single-data items in any generalized data structure. If a more specific structure is defined, however, one can devise ways of referencing such substructures as rings.

In CORAL and Sketchpad, ring-structure primitives are implemented by macros in an assembler language. APL (Associative Programming Language) [12], however, is an extension of PL/1 that manipulates a CORAL structure. As above, a CORAL block is declared as a PL/1 structure, but with attribute ENTITY, instead of CONTROLLED. Hens are declared with attribute SET, and chickens with attribute MEMBER. Value components can be any other PL/1 data item. Variables of type ENTITY contain pointers to entities.

Of interest here are references to three things: rings (called sets in APL), entities, and components of entities. An entity is referenced either by an entity variable or by designating a particular member of a set. The latter is done by number, since rings are ordered. The syntax is:

⟨entity ref⟩ ::= ⟨entity variable⟩ | ⟨set ref⟩(⟨integer⟩) .

A set is referenced by giving the name of a ring component of a particular entity. The syntax is:

⟨set ref⟩ ::= ⟨entity ref⟩.⟨component name⟩ .

Clearly, this also serves to reference value components. For example, A.B and A.B(5).C are set references; and A.X, A.B(3).X, and A.B(5).C(2).X are value references. In these examples, A is an entity variable, B and C are ring component names (declared as SET or MEMBER), and X is a value component name.

Actually, the same syntax might serve as well in a generalized structure by using the integers as duplication factors, and removing the dots that separate names. Furthermore, we have L^6 references. For example, A.B(5).C(2).X would be ABBBBBCCX, conceptually very similar to the APL reference. Clearly, the difference lies in the description.

The LEAP language [14-15] is very similar to ALGOL, with the addition of several statements and data types to handle the relational structure. The relational structure stores triples. The attributes, objects, and values of triples are declared as ITEMs. An ITEM is a manifest constant, whose value is the address of a typical ALGOL variable. The value of the variable is accessed by the operator γ, which is identical to rv. For example,

INTEGER ITEM A

makes A the address of an integer. Its value is γA. There are also variables, called ITEMVARS, whose right-hand values are ITEMs. A triple is specified as

$$A \cdot 0 \equiv V ,$$

where A, 0, and V are ITEMS or ITEMVARS.

The implementation of another data type in LEAP, called SET, is not described. A SET is a collection of ITEMS, and may be specified in a variety of ways:

1) A set variable;
2) A list of ITEMS (e.g., {A, B, C});

3) Unions, intersections and complements of sets;

4) A · O--the set of values of all triples with attribute A and object O;

5) A'V--the set of objects of all triples with attribute A and value V.

CREATING, INSERTING, AND DELETING

The creation of a bead requires a call to storage management to allocate a block of words. In BCPL, AED, and L^6 this requires an explicit function call specifying the size of the bead. The function returns an address used to update the variable that will reference the bead, namely B in A(B). In PL/1, creation is accomplished via the statement:

```
ALLOCATE ⟨structure name⟩ SET ⟨pointer variable⟩ ;
```

and in APL by the almost identical

```
CREATE ⟨entity name⟩ CALLED ⟨entity variable⟩ .
```

In LEAP, a triple is created by the statement:

```
MAKE ⟨attribute⟩ · ⟨object⟩ ≡ ⟨value⟩ ,
```

where attribute, object, and value are ITEMS or ITEMVARS. ITEMS may also be created dynamically by

```
NEWITEM → ⟨itemvar⟩
```

or

```
N(⟨expression⟩) .
```

In the second case, the value of the new ITEM is initialized to the value of the expression.

Generalized structures are built by explicitly updating components of beads with pointers, requiring only the reference mechanisms already described. Triples are added to a LEAP structure by the same make instruction that creates them, and structures are built only by adding triples.

But a CORAL structure is built by inserting blocks in rings, done in APL by the statement:

```
INSERT ⟨entity variable⟩ IN ⟨set ref⟩ .
```

This statement is ambiguous, however, because blocks in a ring are ordered, and specify the position in the ring where the block is to be inserted. In APL, this is done by specifying ordering as an attribute of a set component name. Two such orderings are FIFO, meaning insert entities at the end of the ring; and

```
ORDERED INCR ON ⟨component name⟩ ,
```

which means that entities in a ring are to be kept sorted on a particular component.

Deleting is a bigger problem since it involves un-creation as well as un-insertion. In a generalized structure, these must be done separately. Un-creation, or returning of a bead to free space, is done by a function call, just as in creation. In PL/1, the keyword is FREE. Un-insertion consists of updating pointers to the returned bead. Unfortunately, structures may be made inconsistent in complex ways that the programmer must be aware of.

In a CORAL structure, however, these steps can be more precisely defined. One may request that a block (entity) be removed from a ring (set). The APL statement is

```
REMOVE ⟨entity ref⟩ FROM ⟨set ref⟩ .
```

This requires that pointers in the adjacent blocks in that ring be updated to skip the block in question. One also may delete an entity or a set (DELETE ⟨entity or set ref⟩). Deletion of a set means deletion of all entities in the set. Deletion of an entity means removal from every set of which it is a member, deletion of every set for which it has a SET component (hen), and return of the bead to free storage. A single delete may activate four separate functions: two deletes (both recursive), a remove, and a free; first real advantage of CORAL over a generalized structure in terms of quantity of code necessary to perform the operation.

<u>ITERATION AND SEARCHING</u>

Operations that reference sets as a whole are found only in APL and LEAP. The first of these is "do something for each element of a set." The APL syntax is:

```
FOREACH ⟨entity variable⟩=⟨entity name⟩ IN ⟨set ref⟩
        WITH ⟨boolean exp.⟩ UNTIL ⟨boolean exp⟩ ;
⟨statement list⟩; END .
```

The statement list is executed once for each entity in the set satisfying the WITH and UNTIL clauses, which are optional. The entity variable points to the entity in question during each execution.

In LEAP, where sets are of primary importance, the statement has two different forms and allows multiple iteration variables, called locals. The first form is quite similar to the above:

```
FOREACH ⟨local⟩ IN ⟨set expression⟩ DO ⟨statement list⟩ .
```

The second is:

```
FOREACH A·O≡V DO ⟨statement list⟩
```

where any one or two of A, O, and V may be locals. If V is a local, for example, it is equivalent to FOREACH V IN A·O; but if both O and V are locals, the statement list is executed once for each pair (O, V) such that A·O≡V is in the store.

APL offers one further statement not found in LEAP, potentially the most valuable, that enables one to search rings in various ways. The syntax is:

```
FIND ⟨entity variable⟩ = ⟨entity specification⟩
     WITH ⟨boolean⟩, UNTIL ⟨boolean⟩,
          ELSE ⟨statement⟩ .
⟨entity specification⟩ ::=
          (⟨integer⟩)⟨entity name⟩ IN ⟨set ref⟩
          | ⟨set name⟩ CONTAINS ⟨entity ref⟩ .
```

This statement either finds an entity satisfying the WITH and UNTIL clauses (optional), and updates the entity variable, or it executes the else statement. The entity specification directs it to search for the *n*th entity in a set, with the given properties, or to search for the entity whose set component heads the ring containing a specific entity.

The literature on ASP [13] includes an interesting pictorial representation for ring structures (see Fig. 9). This pictorial representation could be useful not only as an aid to the design of display and problem representation but also as an input to some graphical language (as opposed to a character-string language).

When discussing languages, one notes an absence of any reference to graphics, a condition more or less reflected in the literature. Clearly, however, one could define the Sketchpad entities in APL and obtain a very graphical-looking program (illustrated in Fig. 10) for the simple entities described in Sec. III.

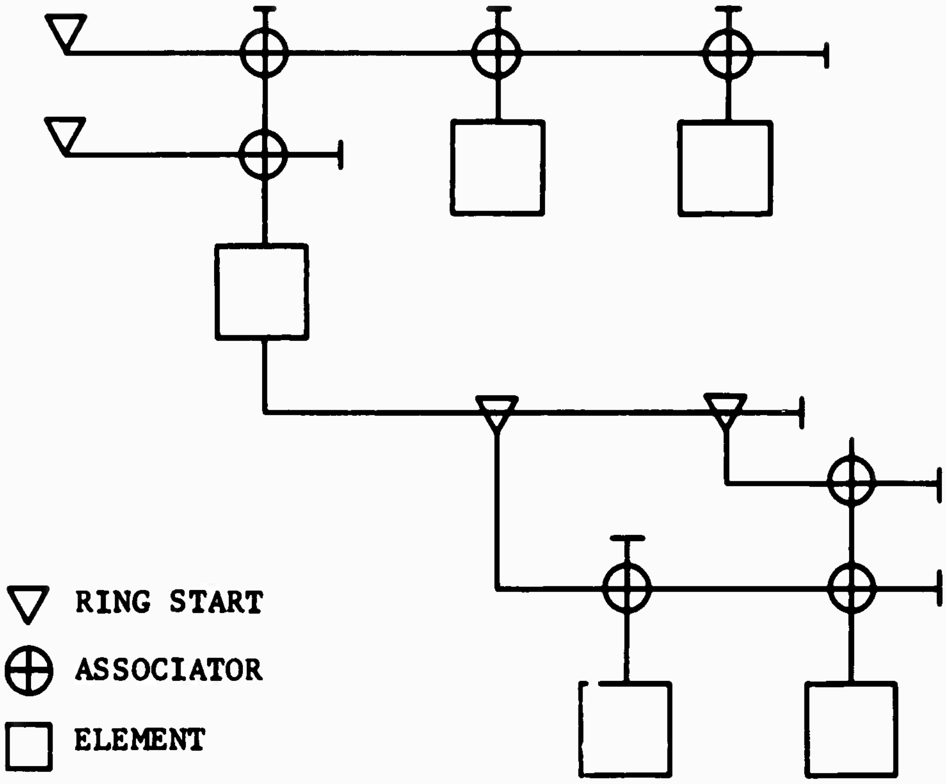


Fig. 9--The ASP Pictorial Structure Representation

```
DECLARE   1 picture ENTITY,
            2 pring SET FIFO,
            2 iring SET FIFO,
          1 instance ENTITY,
            2 pring MEMBER,
            2 iring MEMBER,
            2 trans,
              3 (xtrans, ytrans, sin, cos) FIXED,
          1 line ENTITY,
            2 pring MEMBER,
            2 epoints MEMBER,
          1 circle ENTITY,
            2 pring MEMBER,
            2 epoints MEMBER,
            2 (radius, angle) FIXED,
          1 point ENTITY,
            2 pring MEMBER,
            2 epoints SET,
            2 (xvalue, yvalue) FIXED,
```

Fig. 10--Defining the Sketchpad Entities in APL

V. STORAGE MANAGEMENT

In all of the structures discussed abcve, except associative memory (LEAP), specific calls to storage-management routines are required to get and return beads. From the logical point of view, the specifics of storage management are irrelevant; but in terms of efficiency, they are crucial. In addition, a programmer who is aware of storage-management strategy may be able to write a more efficient program. Several strategies for managing a structure space assumed to be in core are described below, followed by a discussion of secondary storage techniques. Many of the methods described here are implemented in the AED free-storage package [20].

STORAGE-ALLOCATION STRATEGIES

The first step in designing a storage-management package is to choose the available bead sizes. A requirement is that the largest bead used in the structure be available. One choice, which greatly simplifies the problem, is to supply only one block size. A large variance of bead size will result in a lot of wasted space. But if the variance in bead size is less than 25 percent of the mean, it could be a good choice--especially if time is much more important than storage efficiency. To allocate fixed-size beads, a list of available beads is maintained. When a new bead is requested, the first one on the list is supplied; when a bead is returned, it is appended to the end of the list.

It is possible to supply only beads of a few different sizes; but except in special cases, this is no better than supplying exactly the size requested, which is probably the best solution. There are several ways of doing this: one can maintain a free storage list in increasing order of size (see Fig. 11). To supply a bead, one searches the list

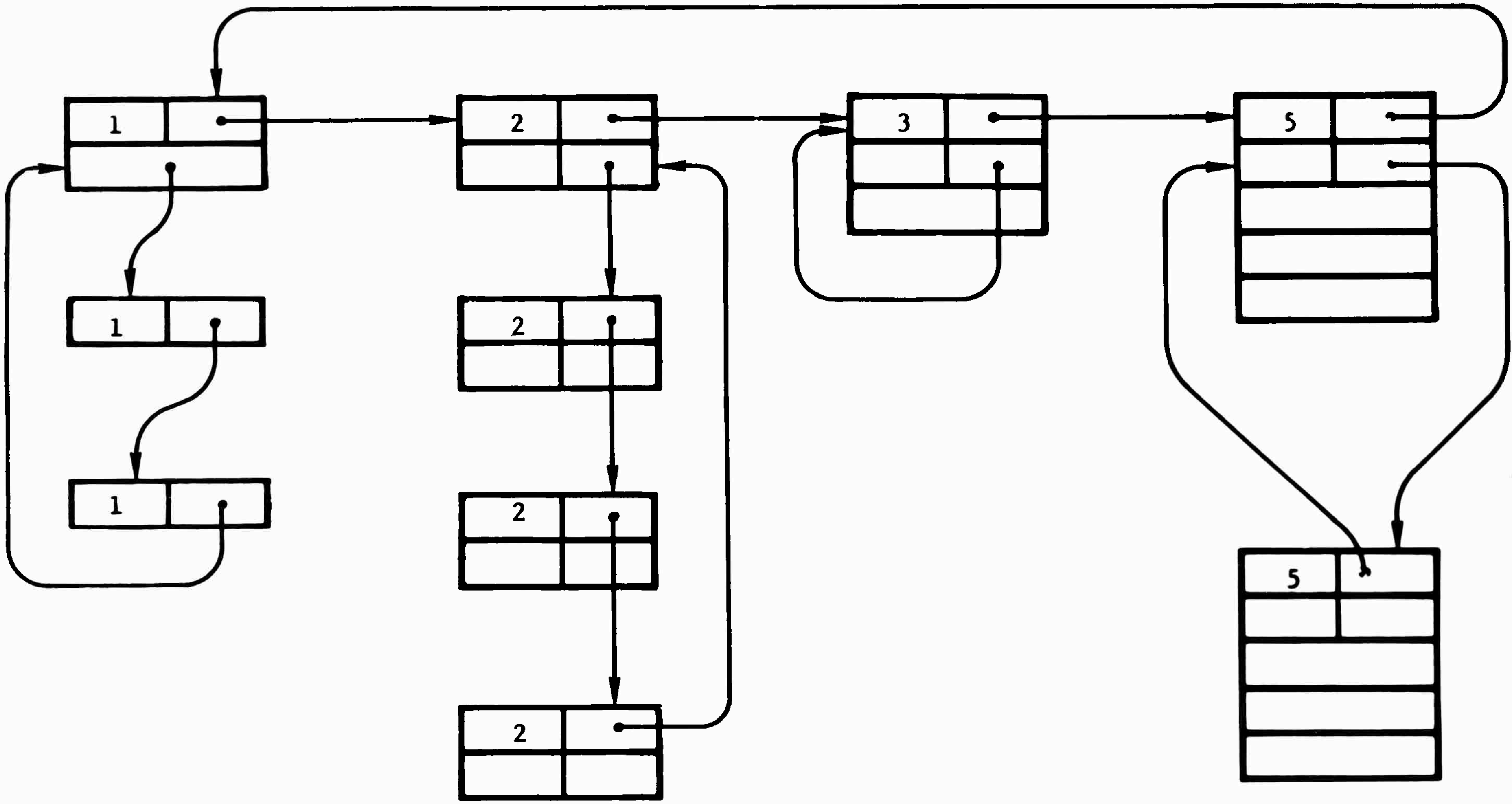


Fig. 11--A Free Storage Management Strategy

for a bead of the right size; if there are none, a larger one is split, and the remainder added to the list in the proper place. If no bead is large enough, there is no simple solution; although two smaller beads might be contiguous, such a case is difficult to find. Thus, there is a chance of considerable loss in space; but the method is fairly efficient.

A second strategy uses a free-storage list in increasing location order. Beads are allocated by searching for a large enough bead, and then splitting it if necessary. When a bead is returned, however, it is combined with any adjacent beads; so that free beads are always as large as possible, and the problem with the increasing size strategy is avoided.

But there is still storage lost because a large bead may not be available when a combination of several small non-contiguous beads would be large enough. The only way to recover this space is by moving used beads, a process called compaction. Compaction is extremely time-consuming because pointers to moved beads must be updated. This requirement also makes compaction impossible in a generalized structure since there is no way of knowing which data items are pointers. In a CORAL structure it is not too difficult, however, because of the back pointers. In any ring structure, one can find the pointers to update by following the ring all the way around; the time consumed is proportional to the average length of the rings. In any case, compaction should be considered only when the program would terminate without it.

Thus, there are four strategies in storage allocation:

1) Single fixed size;
2) Small number of fixed sizes;
3) Arbitrary sizes (no coalescing of returned beads);
4) Arbitrary sizes (coalescing of returned beads).

Each step requires more overhead but uses space more efficiently. *Garbage collection*, used rather freely to mean one or another of the free storage management techniques also refers to the LISP [21] process of scanning the structure and marking used beads, then collecting unused beads in a free storage list. Such a process is unnecessary in the structures discussed here, because beads are always returned explicitly (e.g., by the delete mechanism in APL). Because of the confusion about this term, it should probably not be used.

SECONDARY-STORAGE METHODS

Since data structures can become very large, some means of keeping them partially in secondary storage is necessary. One solution is hardware or software paging techniques (which are easier on the programmer but fail to take advantage of problem structure), in which the related information becomes scattered throughout structure space, requiring more secondary-storage references than necessary. One way of avoiding this problem is to divide the structure space into zones, as determined by the programmer, whc is then required to specify a particular zone when getting and returning beads. This zoning not only allows related information to be kept together, but also permits different storage-allocation strategies for different zones.

Hardware paging still restricts the size of virtual memory, and software paging can be done only when all references are processed through data-structure primitives. Another solution is to require the programmer to deal with secondary storage through specific file operations that store and retrieve parts of the structure. Once again, the zone technique may be useful because one could direct the storage and retrieval of zones, which takes maximum advantage of problem structures but places an added burden on the programmer. Another advantage of explicit file operations is that the space can be compacted as it is transferred, which not only saves space on secondary storage but also saves core space when reloaded.

STORAGE MANAGEMENT IN LEAP

Although the associative memory structure in LEAP requires a great deal of storage, it is efficient because of the implementation of the hash function, which is f (Attribute, Object). The high-order bits of an attribute are used as the track address of a block of storage containing all the triples with that attribute; important because related information is kept together.

Then, the attribute and object are exclusive "or-ed" to obtain an offset within the block of a hash bead. The low-order bits of the attribute are used as the offset of a use-ring header.

The storage within a block is divided into hash beads, conflict beads, value beads, and use headers in about equal quantities. Conflict beads and value beads must be allocated by one of the strategies described. When any bead type is exhausted, the remainder of space in the block is lost; so fragmentation is still a problem.

GLOSSARY

Attribute	In LEAP, one of the three data items in a triple.
Bead	A contiguous subset of the structure space, which is not a part of any other bead.
Block	In CORAL, a bead containing header ring components and value components. Distinguished from the CORAL nub.
Chicken	In Sketchpad, a component containing four pointers; used to make a bead part of a ring.
Component	A contiguous subset of a bead. The smallest object pointed to by a pointer.
Element	In AED, a bead is also an n-component element. In ASP, the bead containing value-data items.
Entity	A bead that represents some graphical object (e.g., a line, circle, etc.).
Hen	In Sketchpad, a component containing three pointers, one of which is part of every ring.
ITEM	In LEAP, any member of a triple. It also has a value.
MEMBER	A chicken in APL.
Manifest Constant	As part of a language, a name (character string) having only one value. The value is known to the language processor.
Nub	In CORAL, a bead containing two ring components, which, in effect, add ring elements to blocks.
Object	In LEAP, one of the three members of a triple. Otherwise used as an author defines it.
Pointer	A data item containing address information that is extracted via a pointer function.
Ring	Any circular sequence of pointers.
Ring component	In Sketchpad and CORAL, the components containing the ring-structure pointers.
Ring element	In CORAL, a chicken.

Ring start

In CORAL, a hen. Also, in ASP, a bead that expands hens, as does the nub in CORAL.

SET

In APL, a ring. In LEAP, a collection of ITEMS.

Structure space

The subset of computer storage used to hold the data structure. The range of the pointer function(s). May include all types of storage.

Triple

In LEAP, an ordered set of three data items. The principal object contained in a LEAP structure.

Value

In LEAP, one of the three data items in a triple. Otherwise, any non-pointer data item.

Value component

In Sketchpad and CORAL, all components except header components and ring components.

REFERENCES

1. *AED-0 Programming Manual*, Preliminary Release No. 2, MIT Electronic Systems Laboratory, November 1964.

2. Richards, Martin, *The BCPL Reference Manual*, MIT Project MAC, Memo M-352-2, January 1969.

3. Knowlton, K. C., "A Programmer's Description of L^6," *Communications of the ACM*, Vol. 9, No. 8, August 1968.

4. *PL/1 Language Specifications, SYSTEM 360 Reference Library*, C28-6571, IBM Corporation, Data Processing Division, White Plains, N.Y.

5. Sutherland, I. E., "Sketchpad: A Man-Machine Graphical Communication System," MIT Lincoln Laboratory Technical Report No. 296, January 1963.

6. Johnson, T. E., "Sketchpad III: A Computer Program for Drawing in Three Dimensions," *AFIPS Conference Proceedings*, Spring Joint Computer Conference, Vol. 23, 1963, pp. 347-353.

7. Roberts, L. G., "Graphical Communication and Control Language," *Second Congress on the Information System Sciences*, November 1964, pp. 211-221.

8. Ling, M.T.S., "Man-Machine Communication with Graphical Console," Engineering Summer Conference on Computer Graphics, University of Michigan, 1965.

9. Cotton, I. W., and F. S. Greatorex, Jr., "Data Structures and Techniques for Remote Computer Graphics," *AFIPS Conference Proceedings*, Fall Joint Computer Conference, Vol. 33, 1968, pp. 533-544.

10. Parker, D. B., "Solving Design Problems in Graphical Dialog," *On-Line Computer Systems*, McGraw-Hill, N.Y., 1967, pp. 176-219.

11. Chasen, S. H., "The Introduction of Man-Computer Graphics into the Aerospace Industry," *AFIPS Conference Proceedings*, Fall Joint Computer Conference, Vol. 27, 1965, pp. 883-892.

12. Dodd, G. G., "APL--A Language for Associative Data Handling in PL/1," *AFIPS Conference Proceedings*, Fall Joint Computer Conference, Vol. 29, 1966, pp. 677-684.

13. Lang, C. A., and J. C. Gray, "ASP--A Ring Implemented Associative Structure Package," *Communications of the ACM*, Vol. 11, No. 8, August 1968.

14. Rovner, P. D., and J. A. Feldman, "LEAP Language and Data Structure," to be published in *Information Processing 1968, Proceedings of IFIP Congress 1968.*

15. -----, "An ALGOL-Based Associative Language," *Communications of the ACM*, Vol. 12, No. 8, August 1969.

16. Ash, W., and E. H. Sibley, "TRAMP: An Interpretive Associative Processor with Deductive Capabilities," *Proceedings of the 23rd National Conference of the ACM*, 1968, pp. 143-156.

17. Sibley, E. H., R. W. Taylor, and D. C. Gordon, "Graphical Systems Communication: An Associative Memory Approach," *AFIPS Conference Proceedings*, Fall Joint Computer Conference, Vol. 33, 1968, pp. 545-555.

18. Symonds, A. J., "Auxiliary Storage Associative Data Structure for PL/1," *IBM Systems Journal*, Vol. 7, Nos. 3 and 4, 1968.

19. Baskin, H. B., and S. P. Morse, "A Multilevel Modeling Structure for Interactive Graphic Design," *IBM Systems Journal*, Vol. 7, Nos. 3 and 4, 1968.

20. Ross, D. T., "The AED Free Storage Package," *Communications of the ACM*, Vol. 10, No. 8, August 1967.

21. McCarthy, J., *et al.*, *LISP--1.5 Programmer's Guide*, MIT Press, Cambridge, Mass, 1962.

CUMENT CONTROL DATA

ORIGINATING ACTIVITY

The Rand Corporation

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

REPORT TITLE

A SURVEY OF DATA STRUCTURES FOR INTERACTIVE GRAPHICS

AUTHOR(S) (Last name, first name, initial)

Hamilton, J. A.

REPORT DATE April 1970	6a. TOTAL NO. OF PAGES 53	6b. NO. OF REFS. 21
CONTRACT OR GRANT NO. DAHC15-67-C-0141	8. ORIGINATOR'S REPORT NO. RM-6145-ARPA	
9a. AVAILABILITY/LIMITATION NOTICES DDC-1	9b. SPONSORING AGENCY Advanced Research Projects Agency	

10. ABSTRACT

Compares methods of organizing data within a computer to permit many interactive computer graphic applications. A data structure is a collection of blocks of machine words (beads) within a subset of the computer's memory. A program for processing such a structure must be able to create and destroy beads and to reference data items (bit strings). The first ability is provided by LEAP'S associative-memory storage allocation system. The referencing ability furnished by several procedural languages enables the programmer to design structures. However, several predesigned ring structures are useful in interactive graphics--e.g., Sketchpad or CORAL.

11. KEY WORDS

Computer graphics
Information processing
Computer programming languages