Data Structure and Storage Management by Dr. D.T. Ross

Lecture 1 M.I.T.

Introduction :

I do not believe it is worthwhile for me or anybody else to provide recipes for "data structures" with any claim for general utility, particularly at this stage of our knowledge. Instead, I want to stimulate your minds with some ideas which allow data structures to be considered in general, in a sensible fashion. I shall start this stimulation with the dogmatic assertion that there is no such thing as a data structure by itself, even though such things are discussed and written about in the literature. To me a data structure is only part of the more general idea of a model, and without this broader interpretation data structures as such are not meaningful. There is also no single data structure useful for many purposes, although the general ideas of data structures can be manipulated in many ways. The key point is that the data structure is a very important part of a model and the data structure form must be selected to suit that model. In this first lecture I wish to discuss the question "What are data structures about?" In the next lecture I want to talk about storage allocation which is at the central core of data structures, and finally, in my last lecture, I want to take a high-level view of a very general kind of data structure called "abstract strings", and see how they fit into the general picture.

Much of material in these lectures is based on an experimental course called "Software Engineering" which I have given once at M.I.T. Its general concept is to give software a firm basis and also to do things; so that students end up having the idea of building a model, and can also make use of the general and powerful concepts they learn elsewhere in the Computer Science courses. However the M.I.T. course is done in a different order from the lecture I will give here because there we start with lectures on AED-0 language (Algol Extended for Design). This language, which is an extension of Algol 60 (to include such ideas as pointers, variable length character strings, optional number of parameters in a procedure call, store a procedure name in a data structure) is used as a vehicle to express the ideas of the course. We have found most students (even those with considerable computer experience) have difficulty with the concepts of a pointer. Experience gained using AED-0 enables the students to grasp the more general concepts later. As yet I have not developed a rigorous mathematical treatment of all the ideas since they are still too flexible. Starting with the generalities (as I must do here) leads to problems of comprehension, since the experience is lacking.

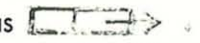
Modelling in General

This will enable us to see Data Structures in their proper setting. We define a PLEX (from PLEXUS meaning an interconnected network) as a complete model of something (either abstract or concrete) which must have

a) Data

b) Structure

c) Algorithms

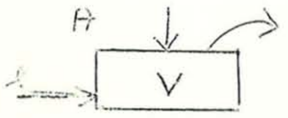
What we want to do is talk about modelling in general, and for this we need the full set a), b), and c) above, and not just talk about the structures in isolation as is so often done using figures such as  We will see that we can change the data structure part of a model without changing the model itself. We will now define terms used in giving a representation of the thing being modelled.

<u>Ideal</u> (Idealised Model or ideal plex)

<u>Mechanize</u> an Ideal by choosing a mechanical form or partitioning of the parts of the model.

<u>Implementation</u> of the plex. Finer and finer partitioning of different mechanizations leads to a detailed level we refer to as an implementation.

The programming language fits in as a way of expressing the various models. To be able to choose different mechanisations we want to separate data and structure from the algorithm part of the model. A data structure is composed of <u>elements</u> with <u>components</u> which may be thought of as <u>records</u> with <u>fields</u>. An Ideal <u>component</u> is represented by 'read' and 'store' procedures.



e.g. (i) value:= read (component name)

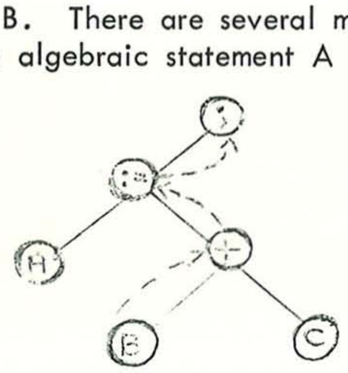
(ii) store (componentname, new value).

At this level the structure of the data is unimportant; we may think of the procedures representing somehow a "box" containing a "value". An Ideal <u>element</u> is a collection of ideal components, and we have "create" and "destroy" procedures for these ideal elements. To describe an ideal element we need to know what storage boxes (components) make it up, also we need to know the mode or type of value for each component. An element type is a cartesian product of components. An <u>Ideal Plex</u> describes models which can be built from ideal elements, and we have procedures "growth" and "mouse", which are a critical part of the total model for they generate and manipulate specific data structures. The "growth" algorithm is the set of rules for building structure with the proper relationships amongst its parts. Thus it corresponds to the parsing algorithm, the grammar for a given plex type, if you will.

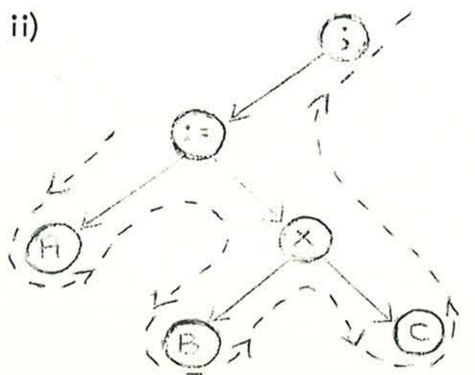
The "mouse" algorithm knows how to step through a data structure in an orderly fashion, i.e. it linearises the data structure. The key thing about a mouse is that it allows you to describe the rules for an operation in atomic fashion in such a way that the operation is induced to a higher level automatically, regardless of the complexity of detail.

N.B. There are several mice possible for a complex data structure, e.g. for the algebraic statement A : = B+C; two possible mice are :

i)



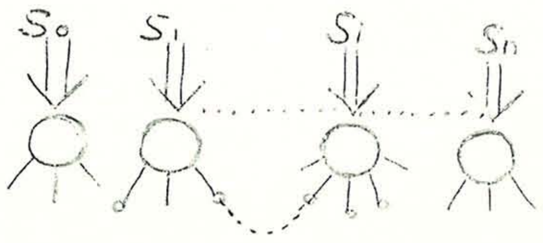
this mouse gives the semantic parse showing evaluation sequenc ii)



this is the syntactic mouse e.g. to print in algebraic notation.

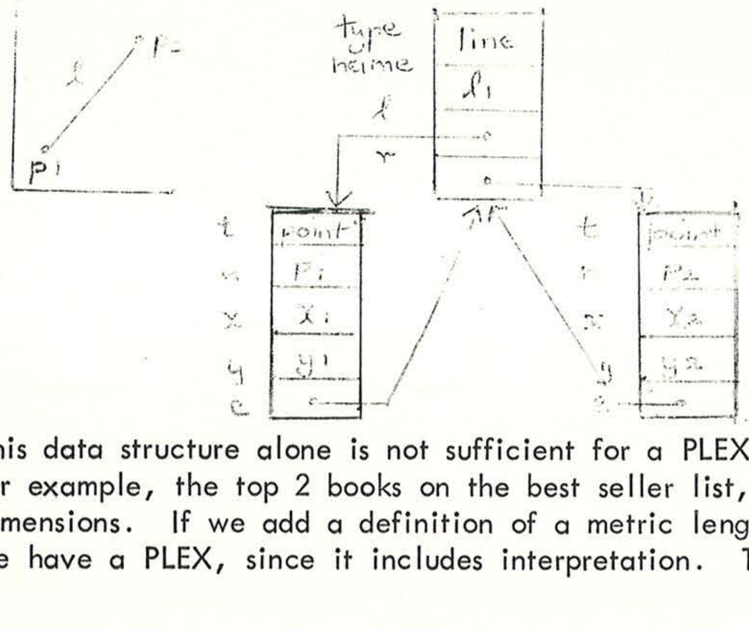
Once defined the sets of procedures (read-store, create-destroy, and growth-mouse) constitute a generic definition of a type of PLEX of which there may be many specific instances. (Notice that the mouse with the destroy procedure as operation rule is the inverse of the growth algorithm).

As an example of the way a growth algorithm grows a data structure, consider several state variables So, S_1,-- S_i-- S_n

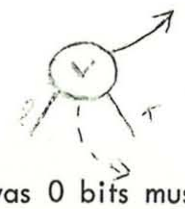


All the connections are considered to be points (even if they are not they can be made to be). Each S_i "holds" an element in a given state of completion. Whenever a new connection is made (depending upon the contents of the total sets of S_i's), the modified element must be moved to a new state variable, thereby changing the total state (all S_i's), so that another connection can be made. Whenever activity stops, a new element is created for S_o, so growth continues.

To show that data, structure, and algorithm all are important, but may take different forms, consider these examples :



This data structure alone is not sufficient for a PLEX. It could be anything - for example, the top 2 books on the best seller list, as well as a line in two dimensions. If we add a definition of a metric length $(l_1\ ) = \sqrt{x^2+y^2}$, now we have a PLEX, since it includes interpretation. The data structure form also may take radically different forms. Returning to the example of a tree given before, a typical node is shown. "Precedence" information (the solid and dashed arrows) takes 30 bits (on 7094) in this form, but the octal stream mechanization takes no bits at all, because you can store information in the algorithm rather than in the data structure. However, while the 30 bits transform to 0 bits, a new "context code" which was 0 bits must now occupy 3 bits in the octal stream form. Thus we see aspects of the model changing between data structure and algorithm domains as we change mechanization. The precedence form is a good mechanization for changing as all the pointers are there, whilst the octal stream form is linearised and hence is easy to put out on disks and is relocatable etc. etc. These different characteristics of radically different mechanical forms of the same ideal model are the primary reason why it is not wise to speak of data structure alone, but only as part of a total plex model. It is also important that the programming language used to express plexes allows mechanizations to be changed easily to suit various circumstances.



Lecture 2

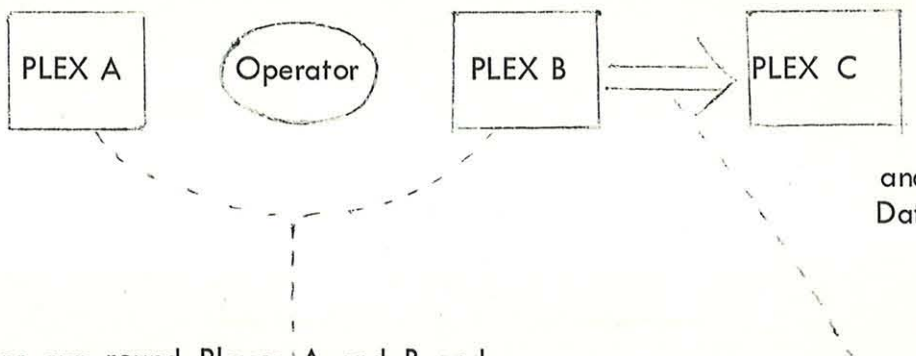
Summary of previous lecture

I should like to start by going over the main points and definitions in yesterday's lecture. We defined the complete model as a PLEX which had to have a) Data b) Structure c) Algorithm. Plex definition specifies a generic type of model of some high-level entities (e.g. Symbol tables, I/O systems). The algorithm part describes the interpretation of the data structure, and we customarily leave this interpretation fixed. Another part of the algorithm complements the data structure part of definition (e.g. algorithms "growth" and "mouse"). When we leave the algorithm part fixed we work with specific instances of grammatical data structures. The Ideal definition is a starting point in which we map all of the data and structural aspects of the definition into the algorithm domain. We also leave the interpretation parts of the algorithm aspect unchanged.

A Component is the finest level of structure (a "box") holding a single datum (unspecified except as to type) and is characterized by read/store procedures.

Out of these we build up to elements, which are cartesian products of the components, and are characterized by procedures create/destroy. Thence we reach Plex structure, characterized by procedures growth/mouse, which provide the basis for control of complexity in specific models. The growth algorithm grows the data structure according to the proper rules, and the mouse algorithm allows one to extract information from the data structure in a desirable order, no matter how complex the detail of the specific data structure may be. These algorithms allow us to treat the entire data structure as a whole but work with only a limited amount of local information at any one time. Because the algorithms growth and mouse are integral with generic definition they can induce local operations to the proper global operations on the entire complex data structure.

Given these definitions our picture of information processing can look like the figure below :



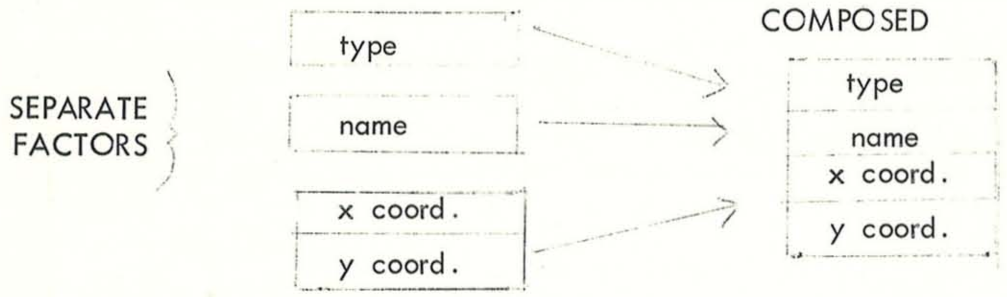
where Plex A and B are actual Data Structures

Two mice run round Plexes A and B and pick out (using read procedures) the right element to be operated on by the operator. The operator combines these results and feeds the growth algorithm.

The growth algorithm grows Plex C (using create and store procedures).

Plex Factor

An important idea is to factor and compose generic plex definitions. In general, the treatment of encoding of type information, for example, can be treated as a separate 'plex factor' that can be put together with many other things. Another instance is the storing and manipulation of names of things. We can compose these with other factors, such as co-ordinate data, in different combinations. Taking the individual "create" procedures for each separate factor we define new elements which are the cartesian product of all 4 components. When defining the new procedures, we also carry along the type constraints for each of these factors.



This is the major reason why we want to keep the specific implementation or mechanisation separate, but still have the meaning carried across.

Re-defining of implementation details is often necessary when putting these "plex factors" together, e.g. in one structure type 1 may be a line, type 2 a point and in another structure type 1 may be a real number, type 2 a boolean expression. If we put these structures together we wish to resolve the type clash automatically. We do this by just re-defining the read/store procedure and create/destroy procedure bodies without re-defining all the algorithms that depend on them.

An individual plex factor is represented by a set of programs; a programming language is the notational vehicle for expressing these ideas and making things work. Thus, an :

Integrated Package of routines is the physical representation of the plex factor. These procedures have a common set of declarations of such things as global variables and global data structures. The individual procedures are not only for building and manipulating data structure, but they also express the intent of the package. There are several kinds of routines in the package.

Atomic routines. The choice of these is most important because if they are too complex the system becomes rigid. It is necessary for these atomic routines to combine easily with others. The full set "spans the space" (in the sense of basis vectors of a vector space) of the plex factor.

Molecular procedures - these are procedures which could be written as combinations of the atomic procedures, but we include them for convenience as an integral part of the package. (The basis need not be minimal, and is more useful if it is not).

Sub-atomic procedures - these allow you to go down into the actual implementation machinery, e.g. in our example to change the encoding of type. Every integrated package is therefore a representation of some plex factor, and it is important to have automatic means for modifying and combining the programs of packages to compose factors.

Free storage system - we will take this as an example of an integrated package. The objective is to represent any piece of data structure by a piece of storage. Assume storage is word or byte oriented.

A Bead is a number of contiguous words of storage and is a good implementation of an element since we can use the addressing mechanism of the machine to carry out read/store actions on components. When you are making a data structure you will have beads of different sizes e.g. 2 words, 3 words etc. It is better to keep these different size beads separated. If you do not, you have what has been called a "heap", and this necessitates a sophisticated garbage collection in order to use store which has been released after use. As we build our data structures we often need something equivalent to the scaffolding used in erecting a building. We must handle our storage such that we can throw away this scaffolding wholesale when we are finished with it, without leaving physical storage peppered full of odd-sized holes.

Storage Zones. In storage we have the following zones :

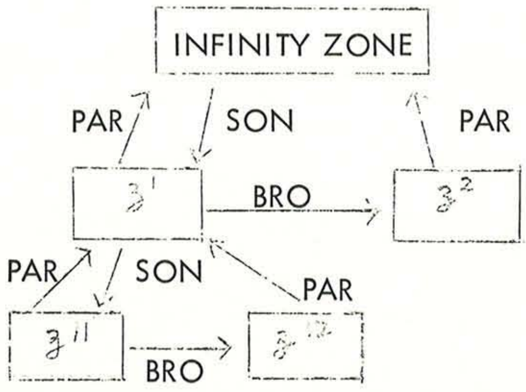
Infinity zone (from which all user storage is derived)

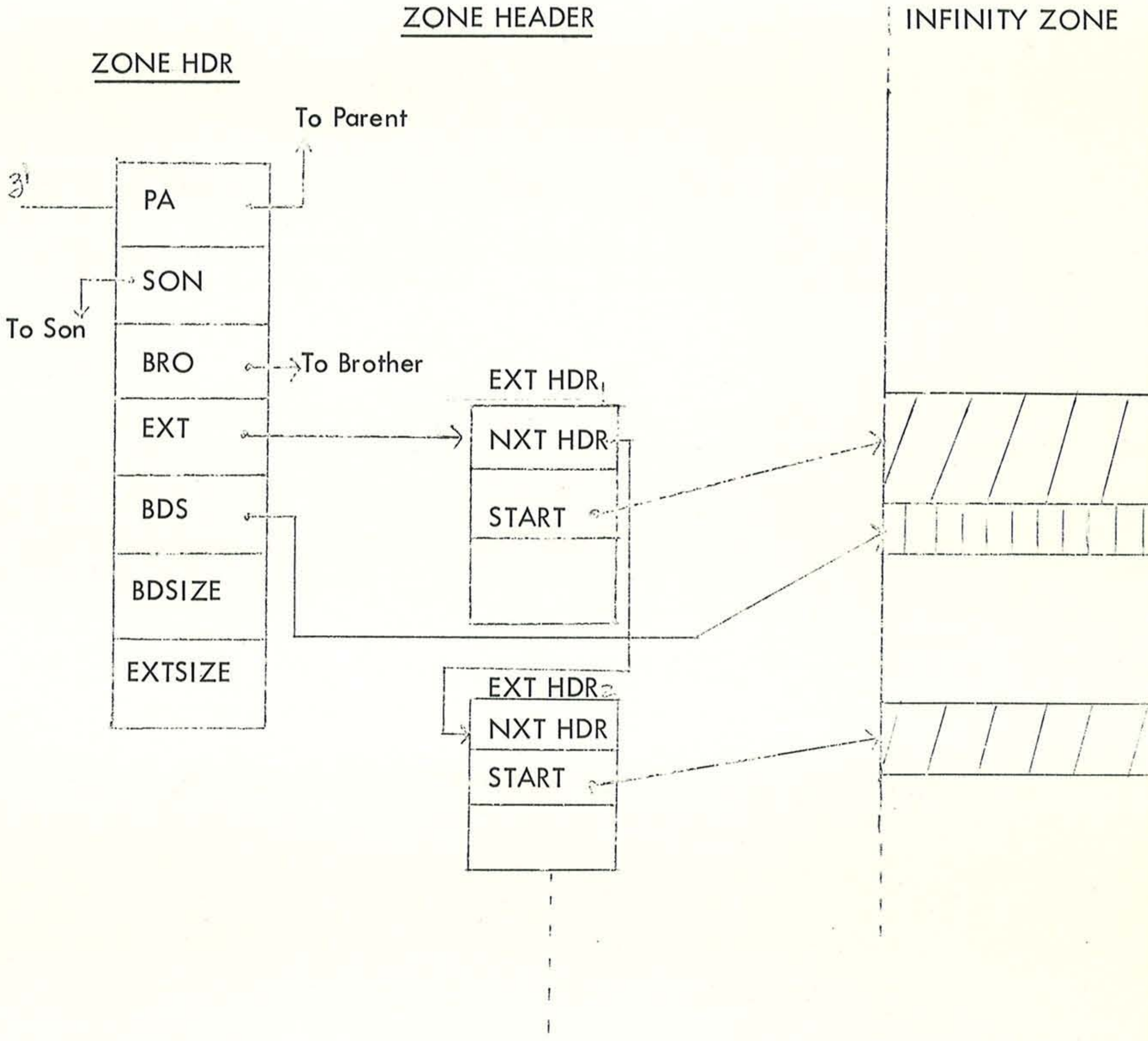
Header zone

Extension zone

These three make up the Divinity Zone, which is given storage by the operating system.

We require the ability to chop up available store for different purposes and then further sub-divide for sub-purposes. For example, the Infinity Zone may be divided up so that it is the "parent" of two zones, z1 and z2. Initially only part of the Infinity Zone is allocated to z1 and z2, so that the remainder can be parcelled out gradually depending upon which one needs more store and when. z1 and z2 can be further sub-divided as shown by the diagram overleaf. This gives a nesting facility.





In the zone header the pointers PA, SON and BRO are self-explanatory. EXT points to the "extension header". BDS starts a chain of beads of store currently available in z1. In the extension header, EXT HDR , there are three pointers:
(i) NXT HDR points to the next extension header, EXT HDR$_2$.
(ii) START points to the beginning of the first address in the infinity zone allocated to that extension.
(iii) NWDS specifies the number of words in that extension.

Another factor of the zone header specifies the beadsize (BDSIZE) of beads in z1 (assumed to be all the same size) and the size of extensions (EXTSIZE) which should be requested, when current storage is exhausted.

The zone and extension headers are the machinery of the zone and we will now give some simple examples of the procedures used :

Example 1

Procedure to define a new zone

```
z := DEFZONE (INITSZ, EXTSZ, BDSZ, FHELP, ZONE)
```

where z is the zone pointer, which references this kind of data structure created by the call on DEFZONE.

INITSZ is the initial size of store to be made available - it may be zero

EXTSZ is the amount to be made available for extension each time

BDSZ is the bead size

FHELP is a user supplied procedure to help the basic free storage procedure overcome any insoluble problems

ZONE is the parent zone

This is the create procedure of the free-storage plex.

Example 2

Procedures to take and return bead B of size S from ZONE, Z.

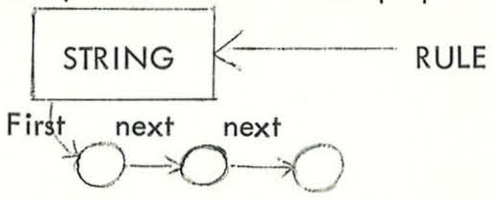
B =: FREE (S, Z) FRET (S, B, Z)	In the body of FREE there is a piece of program which looks for storage in its own zone but if none is available it asks for storage from its parent and so on.

N.B. The parent of the INFINITY zone is the operating system and thus at only one point does storage pass down from the operating system.

FRET returns B to the BDS chain for satisfying a later FREE request.

Lecture 3

In this lecture I would like to draw together the ideas of the previous lectures to show that different types of data structure are often very closely related. In order to do this I want to use the example of an abstract string. The concept of an abstract string we wish to use is not the Algol definition of character string, but a string of arbitrary elements. They can be interconnected elements when a single element may be simultaneously on any number of strings and indeed may initiate any number of further strings. Things grouped together in a string are "similar in kind" in some way. A string is an ordered set with behavioural properties as well. Lists, stacks, rings, queues, hash-coded tables, etc. all are instances of strings. Each string has a rule for describing the set, e.g. all A's, either A or B, etc. and the other properties of the string.



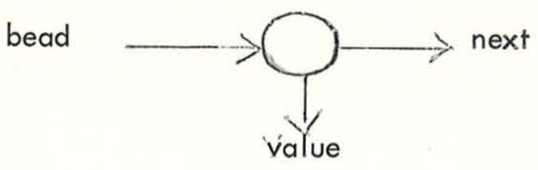
A string is modelled by elements of the following kinds :

1. bead Next, Value

2. string Type, First

3. String type several "basic functions" giving the rules for that string type.

Properties of elements.



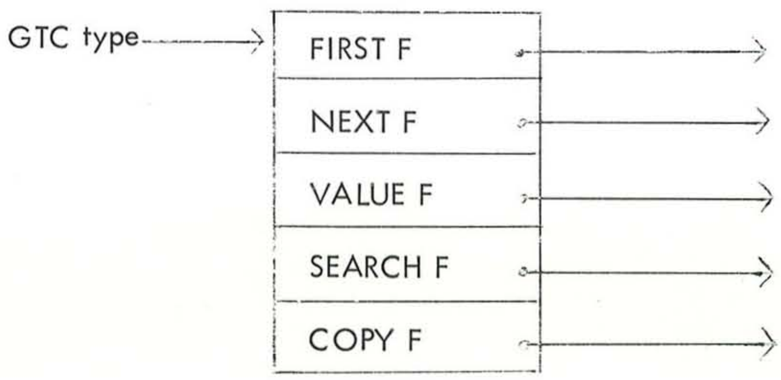
We need basic functions to tell us how to obtain first, next, value, how to search, and how to copy. These functions are :

```
FIRST F    )
           }  read-store pair
NEXT F     }
VALUE F    )
SEARCH F
COPY F     )  create-destroy pair
```

These are the algorithms which give the behaviour. High-level string manipulating functions call on the basic functions. For example, using the AED-0 language, the string package procedure FIND, which finds a value on a string, and if there is no such value calls the bead-valued procedure beadpro, is as follows :-

```
DEFINE POINTER PROCEDURE FIND(VAL, STRING, BEADPRO)

        WHERE INTEGER VAL ; POINTER STR ;

        POINTER PROCEDURE BEADPRO ;

TO BE BEGIN GTCPT = RGTC (FIND.C, STR) ;

                  FIND = DOIT (SRCHF (GTCPT), FIND.C, VAL,

                           PREV, STR, BEADPRO, GTCPT, RETURN);

        END ;
```

DOIT is an extension of ALGOL 60 which takes the first parameter (in this case the search function SRCHF) as a function designator with the remaining parameters as the parameters of this function. DOIT (F, X, Y, Z); is the same as F(X,Y,Z); RGTC stands for Read Generalised Type Component of the string STR to give access to the required set of basic functions. The string type is represented by a bead of the form :



Where each component contains a pointer to one of the basic functions. Such a bead is created by another function (not given here) to define a new string type using a particular set of basic functions.

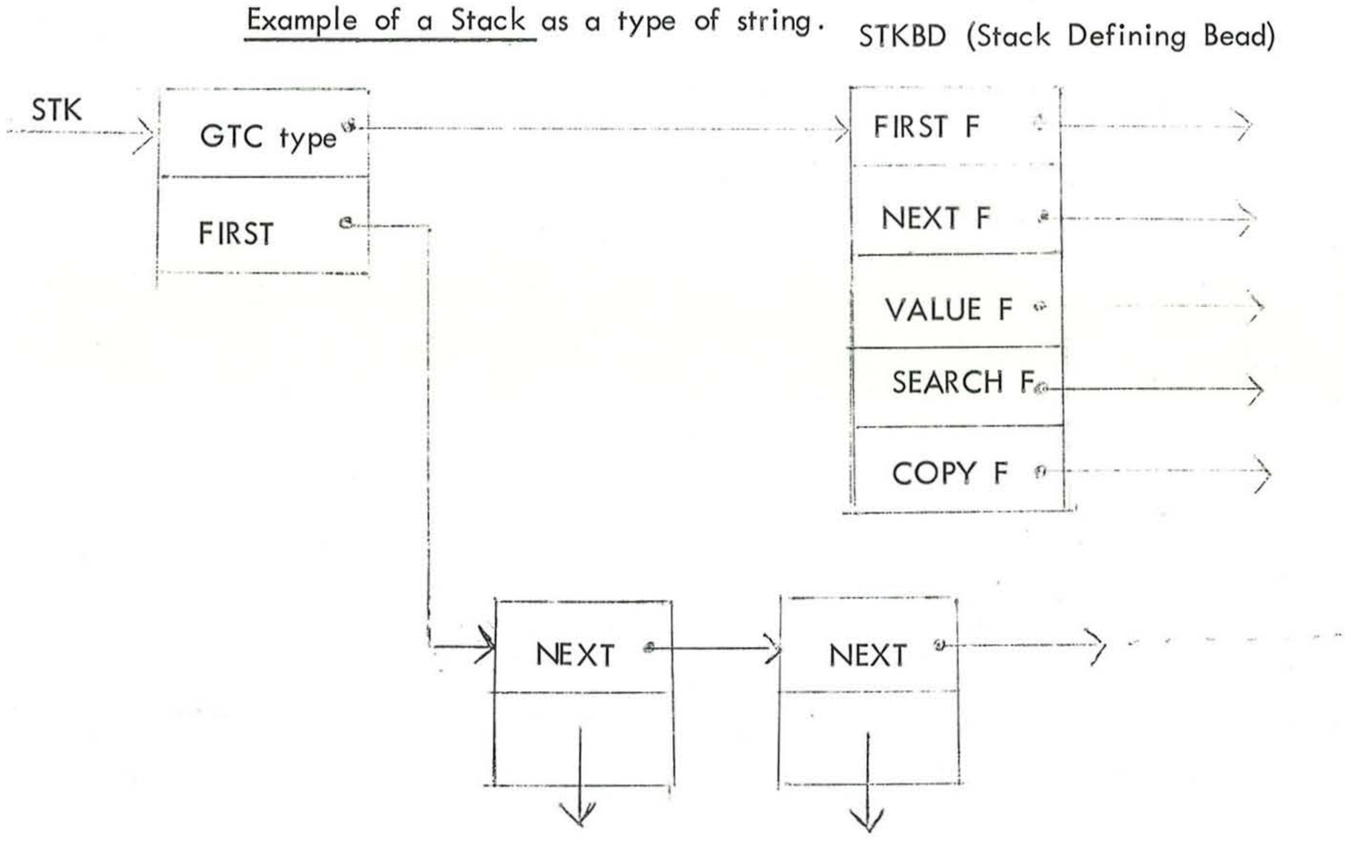
So the action of the above procedure, which is to find a value on a string, is to call the RGTC procedure (which is supplied by the define and which knows about the encoding of the STR types) and from this we get a GTC pointer (GTCPT). We look up the search procedure (SRCHF) with respect to this GTCPT and this search function gives the required aspect of the string behaviour. The search function is a sub-atomic type of routine as are all basic functions. The arguments of DOIT and RGTC are there for the following reasons : FIND.C is a parameter which informs the procedures RGTC and SRCHF that they are being called by the FIND procedure. It is therefore the context of the call. It can be interpreted to point back through several layers of procedure calls, which is of course useful in error tracing situations. VAL, STR, BEADPRO are from the specification part.

PREV is a global pointer variable which couples the search function to other basic functions and other high-level string functions. SRCHF will set it to point to the bead previous to the found bead (i.e. such that NEXT(PREV)= found bead).

GTCPT and RETURN are very interesting arguments.

They allow one to get more features in. The first one allows the use of extra parameters with the type definition which may be added as a "tail" to the GTC bead and used by the basic functions. The second one (RETURN) allows one to put in an arbitrary number of additional arguments with a local call (in this case the call on FIND), so that still further special parameters can be made available to the basic functions, such as SRCHF. Here all of these elaborations are expressed in AED-0 as run-time features, but other parts of AED-0 can be used to automatically eliminate features that are not needed in a particular application. Thus the very abstract string concept with any number of basic functions is reduced to what you require and gives an efficient program in a very highly automated fashion.

<u>Example of a Stack</u> as a type of string.



The NEXT F for example could be :

```
B1 := NEXT (B, S, L);
```

which says get next bead after bead B with respect to string S and if none exists go to label L. The test for stack empty will be in the body of NEXT F and therefore can follow any desired convention. Similarly for the other functions.

At this stage questions were asked on what form are the elements of a string if the element belongs to several strings, and also on the efficiency of implementation of the general concepts. While answering the questions the lecturer explained that the crucial point of the whole approach was to have high-level techniques for handling entire programs as though they themselves were objects. You have to modularise and factor your thoughts, express those thoughts in terms of these techniques, and then map from one mechanization to the other using facilities such as AED provides.

An example considered in the "Software Engineering" course at MIT was the ordered list. The examples given in APPENDIX A and APPENDIX B convey the ideas of alternate mechanizations of a single ordered list concept, and also illustrate the "massaging" of program expressions to achieve concise formulations.

Finally, in reply to a question, the difficulties of preparing and grading examination for the "Software Engineering" course, which deals with such general concepts, were discussed. A book based upon the course is now in preparation and should appear in mid-1969.

APPENDIX A

SUBJECT: Example of Discovering the Essence of a Problem

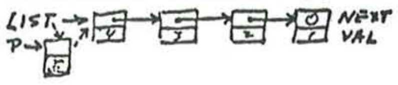
The enclosed notes summarize my lecture on writing a program to make an ordered list, accepting integer values between 0 and 10,000. The strategy selected was to insert each new value at the appropriate place in a growing list of beads with NEXT and VAL components. Other strategies are, of course, possible, and depending upon other unstated conditions, some different strategy might be preferable. But whatever strategy is selected, the same process of honing the expressions of the problem to achieve the most elegant formulation should be followed. Only when the redundancies and irrelevancies have been removed from each of two strategies can those strategies properly be compared.

Each of your homework and computer assignments should be subjected to the same process, insofar as you are able. Developing skill in these techniques is an essential foundation upon which further true software engineering can be based. The process exemplified here is not necessarily a time-waster and an expensive luxury. With practice, the disciplined evolution of clear thinking expressed in clean and elegant programs can shorten the debugging process far more than sloppy work performed in a mad and ill-directed rush.

Later in the course, we will be working with very powerful concepts and approaches. As with any other instance of increased leverage, these powerful techniques can turn into ugly monstrosities which do more harm than good, if they are not properly applied. At that time, the importance of the elegance and precision which can only come from skill in these basic manipulations will be quite apparent.

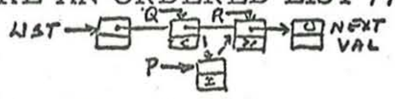
Note in particular the final versions of this ordered list example - one for the machine and one for other people. In order to allow me to understand your programs well enough to evaluate them and offer helpful suggestions, would you please emulate this format in work you hand in, as well. I recommend highly that you make these neat and complete versions before running on the computer, for the discipline of adding detailed remarks to explain the program frequently will show you weak points or errors in your current formulations. Excessive wordiness is not requried. Add pictures of your bead structures and use crisp English as well as AED language constructs. Assignments will turn out short if you do them well.

We started with this simpler problem already solved-



```
         BEGIN ... PROGRAM TO COLLECT VALUES //
         .INSERT MAIN
         POINTER LIST, P $,
         INTEGER I $,
         POINTER COMPONENT NEXT $, NEXT $=$ 0 $,
         INTEGER COMPONENT VAL $, VAL $=$ 1 $,
LOOP$ ISI(GIN(I), STOP) ... READ NEW VALUE INTO I, STOP IF NOT
                                   INTEGER $,
         VAL(P=FREZ(2))=I ... PUT IN NEW BEAD "HELD" BY P $,
         NEXT(P)=LIST ... TACK OLD LIST ONTO NEW BEAD $,
         LIST=P ... UPDATE LIST TO INCLUDE NEW BEAD $,
         GOTO LOOP ... REPEAT $,
STOP$ FINISH( ) END FINI
```

Then starting with the major idea of scanning and splicing-



```
         BEGIN ... FIRST TRY TO MAKE AN ORDERED LIST //
         .INSERT MAIN $,
         POINTER LIST, P, Q, R $,
         INTEGER I $,
         POINTER COMPONENT NEXT $, NEXT $=$ 0 $,
         INTEGER COMPONENT VAL $, VAL $=$ 1 $,
LOOP$ ISI(GIN(I), STOP) ... SAME IDEA AS BEFORE $,
         VAL(P=FREZ(2))=I $,
         R=LIST ... INITIALIZE SCANNING VARIABLE $,
SCAN$ IF VAL(R) LES I THEN BEGIN R=NEXT(Q=R) $, GOTO SCAN END
            ... Q MARCHES ALONG BEHIND R UNTIL VAL(R) GEQ I $,
         NEXT(Q)=P ... VAL(Q) < I ≤ VAL(R) SO START SPLICE $,
         NEXT(P)=R ... FINISH SPLICING NEW BEAD IN $,
         GOTO LOOP ... REPEAT $,
STOP$ FINISH( ) END FINI
```

Now we start asking questions. The above is fine for the general step n-to-step n+1, but what about special cases?

1. What happens if I GEQ VAL(R) for all R, (i. e. scan falls of the end)?

   <u>Ans</u> R=NEXT(last R)=0 $, GOTO SCAN tests VAL(0) !

   So we need to fix it up.

   Better way is-

```
     SCAN$ IF VAL(R) LES I ~~[illegible]~~ AND
                  ~~[illegible]~~(R=NEXT(Q=R)) NEQ 0 THEN GOTO SCAN $,
```

2. What happens now if VAL(LIST) GEQ I, (i. e. goes at beginning)?

   <u>Ans</u>. NEXT(Q)=new P (ugh!) and LIST is unchanged!

   Better way is-

```
SCAN$ IF VAL(R) LES I AND
              (R=NEXT(Q=R)) NEQ 0 THEN GOTO SCAN $,
         IF R EQL LIST THEN LIST=P ... PUT AT BEGINNING //
              ELSE NEXT(Q)=P $,
```

3. What happens if LIST=0, (i. e. the very first time)?

   Ans IF VAL(R=LIST=0) bombs out again!

   Better way is-

```
IF (R=LIST) EQL 0 THEN GOTO FIRST $,

SCAN$ ~~~~~~~~ (same as before)
         IF R EQL LIST THEN FIRST $ LIST=P
              ELSE NEXT(Q)=P $,
```

   Here, since we already had the desired LIST=P we used it via GOTO FIRST instead of putting in another.

This exhausts all conditions, so we now have our first complete expression of the problem. (Note that the completest statement of a problem often is a method of solution.)

But this expression of the problem has built into it the history of our discoveries. There is probability zero that we discovered things in just the right sequence and resolved each new thing in exactly the right way.

So we now proceed to seek improvements without "loosing" or distorting the ideas of our complete understanding of the problem.

Let's first collect our thoughts-

```
       BEGIN ... FIRST COMPLETE PROGRAM COLLECTED //
       ~~~(Same Declarations)~~~
LOOP$ ISI(GIN(I), STOP) $,
      VAL(P=FREZ(2))=I $,
      IF(R=LIST) EQL 0 THEN GOTO FIRST $,
SCAN$ IF VAL(R) LES I AND
        (R=NEXT(Q=R)) NEQ 0 THEN GOTO SCAN $,
      IF R EQL LIST THEN FIRST $ LIST=P
           ELSE NEXT(Q)=P $,
      NEXT(P)=R $,
      GOTO LOOP $,
STOP$ FINISH( ) END FINI
```

We notice that NEXT(Q)=P is the proper disposal of the new bead for both the normal and the at-end cases, and LIST=P is the proper thing for the special empty list and at-beginning cases.

If we describe these cases precisely, we get one IF expression that tells which way to dispose of P.

The "at the beginning" arises when 1.) LIST is empty or 2.)the new value is LES the first listed value, so instead of GOTO FIRST we move the LIST=P up to the THEN slot of the first IF.

```
        IF (R=LIST) EQL 0 OR VAL(LIST) GEQ I
            THEN LIST=P
            ELSE SCAN$ IF VAL(R) LES I AND
                (R=NEXT(Q=R)) NEQ 0 THEN GOTO SCAN ~~$,~~
                    ELSE NEXT(Q)=P $,
```

Now we notice that initially we have already tested VAL(R) GEQ I so we undo the AND and GOTO SEEK.

```
        IF (R=LIST) EQL 0 OR VAL(R) GEQ I
            THEN LIST=P ... NEW BEAD GOES AT BEGINNING //
            ELSE BEGIN GOTO SEEK ... VAL(LIST) ALREADY TESTED $,
                  SCAN$ IF VAL(R) LES I THEN          (see below)
                        SEEK$ IF(R=NEXT(Q=R)) NEQ 0
                                   THEN GOTO SCAN $,
                         NEXT(Q)=P ... VAL GEQ OR END REACHED //
                  END $,
        NEXT(P)=R ...SAME AS BEFORE $,
        GOTO LOOP $,
```

That's better, but looks messy. Why can't we flip things around to get rid of that strange GOTO SEEK?

We can! By using the execution sequence of AND which says that in X AND Y, if X is FALSE, Y is never executed or tested because the AND expression is already known to be FALSE.

So we reinstate the AND version, but with the order reversed.

```
        IF (R=LIST) EQL 0 OR VAL(R) GEQ I         =LIST, but R version
            THEN LIST=P                           would probably be in a
            ELSE ~~BEGIN~~                        machine register.
                  SCAN$ IF(R=NEXT(Q=R)) NEQ 0 AND VAL(R) LES I
                            THEN GOTO SCAN ~~$,~~ ELSE
                         NEXT(Q)=P
                  ~~END~~ $,
        NEXT(P)=R $,
        GOTO LOOP $,
STOP$ FINISH( ) END FINI
```

This seems to be the best we can do, since it protects against VAL(0) and does the right thing as fast as possible in all cases.

But-----

A general idea which works in a surprisingly large number of cases (if you can see how) is to completely eliminate the special initial cases by making the n-to-n+1 normal case work in all circumstances.

This is done by building in part of our ideas in the form of an initial data structure so that the special cases obey the same rules as the general case.

In this example, we can eliminate the LIST=P disposal of the new bead (and in fact the P variable itself!) if we

(1) Ensure that LIST is never empty
(2) Ensure that VAL(LIST) is LES any possible I value.

If these conditions are true, LIST=P will never be used!

So we create an initial bead with (in effect) $-\infty$ in it.

```
        VAL(LIST=FREZ(2))= -1 ... GIVEN THAT 0 ≤ I ≤ 10,000 $,
LOOP$ ISI(GIN(I), STOP) ... WE'LL DEPEND ON I HOLDING THE NEW VALUE
                                 WHILE WE FIND OUT WHERE TO PUT IT $,
        R=LIST $,
SCAN$ IF(R=NEXT(Q=R)) NEQ 0 AND VAL(R) LES I
              THEN GOTO SCAN $,
        NEXT(Q=NEXT(Q)=FREZ(2))=R $,
        VAL(Q)=I ... FINALLY DISPOSE OF NEW VALUE $,
        GOTO LOOP $,
STOP$ FINISH( ) END FINI
```

Here we wait until we've found the right Q before getting the new bead and put it in the right spot directly. Then Q can play the role of P which may be dropped from the declarations.

This all came from dropping out the LIST=P case.

Now we ask can we drop the NEQ 0 test (which really functions only at the beginning and thus also is a special case)?

Sure we can, if we can make the VAL(R) LES I test play the same role. I. e. make sure VAL (last R) is always GEQ I, i. e. $+\infty$!

So we add to the initialization another bead with (essentially) $+\infty$ in it.

```
        VAL (LIST=FREZ(2))= -1 $,
        VAL (NEXT(LIST)=FREZ(2))= 1C35 ... 1 * 2^35 $,
LOOP$ ISI(GIN(I), STOP) $,
        R=LIST $,
SCAN$ IF VAL(R=NEXT(Q=R)) LES I THEN GOTO SCAN $,
        NEXT(Q=NEXT(Q)=FREZ(2))=R $,
        VAL(Q)=I $,
        GOTO LOOP $,
STOP$ FINISH( ) END FINI
```

This seemed to me to be the shortest and fastest possible, BUT ---
After class it was pointed out to me that in focusing on the + ∞ argument, I had overlooked a fact that crept in when the LIST=P case was dropped out.

Namely, if you examine the above you see that we no longer __ever__ test VAL(LIST) but only VAL(__NEXT__(LIST)). Therefore, the - ∞ is not needed at all!

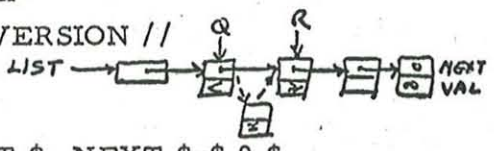
So ---

```
        LIST=FREE(1) $,
        VAL(NEXT(LIST)=FREZ(2))=1C35 $,
        etc.
```

Which can in turn be nested to give-

The most elegant and compact version-

```
        BEGIN ... FULLY NESTED VERSION //
        . INSERT MAIN $,
        POINTER LIST, Q, R $,
        INTEGER I $,
        POINTER COMPONENT NEXT $, NEXT $=$ 0 $,
        INTEGER COMPONENT VAL $, VAL $=$ 1 $,
        VAL(NEXT(LIST=FREZ(1))=FREZ(2))=1C35 $,
LOOP$   ISI(GIN(I), STOP) $,
        R=LIST $,
SCAN$   IF VAL(R=NEXT(Q=R)) LES I THEN GOTO SCAN $,
        NEXT(Q=NEXT(Q)=FREZ(2))=R $,
        VAL(Q)=I $,
        GOTO LOOP $,
STOP$   FINISH( ) END FINI
```



Which can mechanically be expanded for people to understand-

```
        BEGIN ... FULLY EXPANDED VERSION //
        (Same Declarations)
        LIST=FREZ(1) ... GET A "LEFT BOUNDARY" BEAD $,
        NEXT(LIST)=FREZ(2) ... ATTACH A "RIGHT BOUNDARY" BEAD $,
        VAL(NEXT(LIST))=1C35 ... MAKE VALUE OF RIGHT BOUNDARY
                                      INFINITE $,
LOOP$   ISI(GIN(I), STOP) ... READ AND TEST A VALUE.  GOTO STOP IF
                                                    NOT INTEGER $,
        R=LIST ... INITIALIZE SCANNING VARIABLE TO START OF LIST $,
SCAN$   Q=R ... INSIDE SCAN LOOP, Q REMEMBERS PREVIOUS BEAD FOR
                                      SPLICING $,
        R=NEXT(Q) ... MOVE SCANNING VARIABLE TO CONSIDER NEW BEAD $,
        IF VAL(R) LES I THEN GOTO SCAN... IF VALUE IN LIST SMALLER,
                                               KEEP GOING $,
        NEXT(Q)=FREZ(2)... WHEN GET HERE VAL(Q)<I≤ VAL(R) SO PUT NEW
                                                     BEAD AFTER Q $,
```

(continued on next page)

```
        Q=NEXT(Q) ... NO LONGER NEED OLD Q, SO POINT TO NEW BEAD $,
        NEXT(Q)=R ... TIE R BEAD AFTER NEW BEAD, COMPLETING THE
                                                     SPLICE $,
        VAL(Q)=I ... FINALLY PUT IN THE VALUE THAT CAUSED THE
                     COMMOTION $,
        GOTO LOOP ... NOW READY TO READ A NEW VALUE, SO GO BACK $
STOP$ FINISH( ) END FINI
```

For non-trivial problems (the most common variety!) this kind of churning can greatly deepen a person's understanding of the problem.

There is no other way to dispose of the "woods-for-the-trees" syndrome. It is not just playing games and should not be abused into just another type of coding-bum behavior.

Instead it should be recognized as an essential component of truly competent software engineering.

No matter what the problem, nor what your general scheme for attacking that problem may be, if you think sloppy and work sloppy and have a sloppy expression of the problem it is impossible to achieve an elegant and powerful solution.

You never will be able to see every facet and aspect at the start. Practice and train yourself to recognize the good, solid features of each attempt as well as the weak points. Then maintain the good points [R=NEXT(Q=R) was discovered on the very first step and still is present in the final version!] as you successively resolve the weak points.

In this way you will develop a professional skill which will allow you to evolve an elegant solution to the most complex problems.

Even as simple a task as making an ordered list has, as we have seen, many rich layers to be worked through.

Study this example again and again -

Then go and do ye likewise!!

APPENDIX B

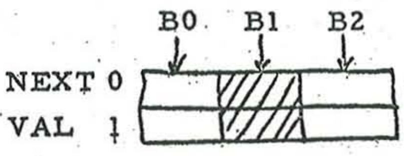
SUBJECT: Example of Mechanical Transformation from Beads to Arrays.

These notes summarize my lecture on the essentially-mechanical transformation of the ordered-list example ~~(Doc. 6,687-16, March 5, 1968~~) from its original bead-oriented form to a form using arrays.

(see APPENDIX A) ^

The comparisons made between beads and arrays may be summarized as follows:

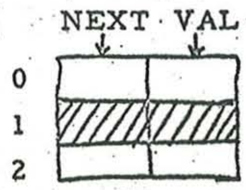
BEADS	ARRAYS
Object-oriented scheme	Property-oriented scheme
Component name selects property	Array name selects property
Pointer selects object	Index selects object



NEXT(Bi)=NEXT(i)

VAL(Bi)=VAL(i)

▨ = Object 1



Compilation Values

NEXT \$=\$ 0 (integer)	NEXT is LOC (pointer)
VAL \$=\$ 1 (integer)	VAL is LOC (pointer)

Run-Time Values

Bi is LOC (pointer)	i is index (integer)

Compile Time

No restriction on number of objects	Fixed number of objects allocated

Run Time

Dynamic storage allocation cost.	No allocation cost, but must check range of index dynamically for reliability.

Conserves

$$\begin{tikzcd}
\text{Flexibility} & & \text{Run time} \arrow[ddll, <->] \\
\text{Costs} & & \\
\text{Run time} & & \text{Flexibility} \arrow[uull, <->]
\end{tikzcd}$$

But have common referent structure: A(B) means "Property A of Object B"

We started with the following program for making an ordered list using the bead method.

```
        BEGIN ... FULLY EXPANDED VERSION //
        .INSERT MAIN $,
        POINTER LIST,Q,R $,
        INTEGER I $,
        POINTER COMPONENT NEXT $, NEXT $=$ 0 $,
        INTEGER COMPONENT VAL $, VAL $=$ 1 $,
        LIST=FREZ(1) ... GET A "LEFT BOUNDARY" BEAD $,
        NEXT(LIST)=FREZ(2) ... ATTACH A "RIGHT BOUNDARY" BEAD $,
        VAL(NEXT(LIST))=1C35 ... MAKE VALUE OF RIGHT BOUNDARY
                                  INFINITE $,
LOOP$ISI(GIN(I),STOP) ... READ AND TEST A VALUE.  GOTO STOP IF
                                                NOT INTEGER $,
        R=LIST ... INITIALIZE SCANNING VARIABLE TO START OF LIST $,
SCAN$ Q=R ... INSIDE SCAN LOOP, Q REMEMBERS PREVIOUS BEAD FOR
                                  SPLICING $,
        R=NEXT(R) ... MOVE SCANNING VARIABLE TO CONSIDER NEW BEAD $,
        IF VAL(R) LES I THEN GOTO SCAN ... IF VALUE IN LIST SMALLER,
                                          KEEP GOING $,
        NEXT(Q)=FREZ(2) ... WHEN GET HERE VAL(Q)<I≤ VAL(R) SO PUT NEW
                                                BEAD AFTER Q $,
        Q=NEXT(Q) ... NO LONGER NEED OLD Q, SO POINT TO NEW BEAD $,
        NEXT(Q)=R ... TIE R BEAD AFTER NEW BEAD, COMPLETING THE
                                                SPLICE $,
        VAL(Q)=I ... FINALLY PUT IN THE VALUE THAT CAUSED THE
                       COMMOTION $,
        GOTO LOOP ... NOW READY TO READ A NEW VALUE, SO GO BACK $,
STOP$ FINISH( ) END FINI
```

We first perform an essentially mechanical transformation in which the NEXT and VAL components become arrays, but the logic of the program is essentially unchanged. The main points of concern are:

1.) Changes in declarations
2.) Initialization
3.) Free storage usage.

Since obtaining a new N-word bead from free storage is equivalent to increasing by one the maximum used index on N arrays, we could define an array-oriented free storage procedure

```
DEFINE INTEGER PROCEDURE FREZ(N) WHERE INTEGER N
  TOBE FREZ=MAXR=MAXR+1 ... NOTE: N IS NOT RELEVANT $,
```

where MAXR is assumed to be suitably initialized at first. FREZ merely returns the next available index value.

But since this procedure body is so short, we will choose to edit it into programs (with a maximum size check corresponding to the declared array size) instead of leaving FREZ calls.

For the above program, the <u>declaration</u> changes are as follows:

1.) The pointer variables become integer index variables, along with I.
INTEGER I, Q, R, MAXR $,
LIST disappears by the convention that LIST as an index will always have the value 0, i.e., the list begins at the 0 position of the arrays.
2.) The component declarations become array declarations.
INTEGER ARRAY NEXT(200), VAL $,
Note that NEXT also is index-valued now. (The choice of 200 as the size of the NEXT and VAL arrays is arbitrary.)

For the above program, the <u>initialization</u> changes are as follows:

1.) LIST=FREZ(1) disappears by the above convention that LIST$\equiv$0.
2.) NEXT(LIST)=FREZ(2) $, becomes
NEXT(0)=MAXR=1 $,
because LIST=0 and the object with VAL = $+\infty$ has index 1.
3.) VAL(NEXT(LIST))=1C35 $, becomes
VAL(1)=1C35 $,
because NEXT(0)=1 by the previous statement.

The remainder of the program is unchanged, except for replacing LIST with 0 and the above expansion and test for FREZ. So the first, mechanically produced array version is:

```
               BEGIN ... FIRST ARRAY VERSION //
               .INSERT MAIN $,
               INTEGER I, Q, R, MAXR $,
               INTEGER ARRAY NEXT(200), VAL $,
               NEXT(0)=MAXR=1 $,
               VAL(1)=1C35 $,
LOOP$          ISI(GIN(I), STOP) $,
               R=0 $,
SCAN$          Q=R $,
               R=NEXT(R) $,
               IF VAL(R) LES I THEN GOTO SCAN $,
               IF(MAXR=MAXR+1) GRT 200 THEN GOTO TROUBLE $,
               NEXT(Q)=MAXR $,
               Q=NEXT(Q) $,
               NEXT(Q)=R $,
               VAL(Q)=I $,
               GOTO LOOP $,
TROUBLE$       GOUT(.C. /OUT OF STORAGE/) $,
STOP$          FINISH( ) END FINI
```

This program will execute faster than the bead version, even though its strategy is the same, because of the much simpler FREZ mechanism. It is rigid, in that it will accept no more than 200 values, and occupies the same storage for one value as it does for 200, but supposedly that was an acceptable price to pay for shorter running time. But we have done no honing, as yet, to see if that running time can be improved. If we look at the program, however, we see that since the strategy has been left unchanged by this first mechanical transformation, no new changes suggest themselves except to move the free storage IF expression up to follow LOOP, so that if we do not STOP, but do run out of storage, the program halts as soon as possible. (Note that this has the effect of replacing FREZ(2) by MAXR in the original bead version.)

A technique which is always possible when transforming from a bead-oriented program to an array-oriented program is to take some pointer component and treat it as the basis for the object-naming scheme, in which case the array representing that component may be eliminated entirely. In other words, if we choose to number objects not in the sequence of creation (as the above MAXR version of FREZ does), but in the "natural" sequence given by a component such as NEXT, then for the entire NEXT array we would have

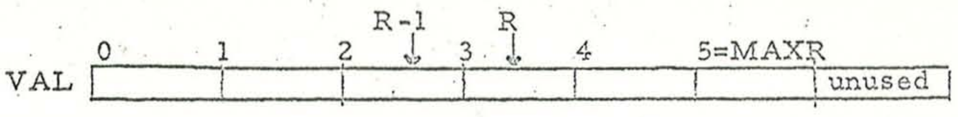
$$\text{NEXT}(i) = i+1$$

so we don't really need the NEXT array.

If the order inherent in such a NEXT component is immaterial, this "natural" sequence can just as well be the FREZ creation sequence, and this process of eliminating NEXT saves storage at no corresponding cost of time.

If, however, the order inherent in NEXT is a part of the problem model itself (as in the present case of an ordered list), the saving in storage must be paid for by an increase in running time.

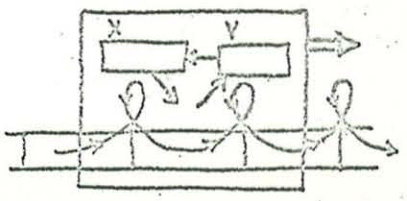
The fact that order is important says that FREZ must be affected. In particular, consider a (horizontal) VAL array before a new object is created:



Suppose the new value must be inserted between VAL(R-1) and VAL(R). Then all values from VAL(R) on must be moved up one place to "make room" for the new value.

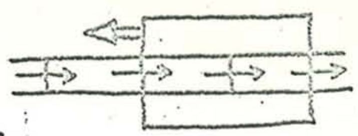
This may be done neatly in increasing order by introducing two temporary variables to hold values:

```
        INTEGER P, R, X, Y, MAXR $,
        P=R $,
        X=VAL(P) $,
   L$   P=P+1 $,
        Y=VAL(P) $,
        VAL(P)=X $,
        IF P EQL MAXR THEN GOTO DONE $,
        X=Y $,
        GOTO LOOP $, DONE $
```



But the same job is done even more neatly with no temporary storage, if it is merely done backwards:

```
        INTEGER P, R, MAXR $,
        P=MAXR $,
MOVE$ VAL(P)=VAL(P-1) $,
        IF (P=P-1) GRT R THEN GOTO MOVE $,
```



This may be used to write a general-purpose pair of procedures to handle free storage for arrays when index order is important.

```
INTEGER PROCEDURE AFREZ ... IN OUTERMOST BLOCK $,
DEFINE PROCEDURE SET. AFREZ (ALLOCATED. SIZE, VAL) WHERE
      INTEGER ALLOCATED. SIZE $, INTEGER ARRAY VAL
      TOBE BEGIN INTEGER MAXR $, MAXR=-1$,
          DEFINE INTEGER PROCEDURE AFREZ(P) WHERE INTEGER P
            TOBE BEGIN INTEGER Q, R $,
                   IF(R=MAXR=(Q=MAXR)+1) GRT ALLOCATED. SIZE
                      THEN BEGIN GOUT (.C. /OUT OF STORAGE/) $,
                                  FINISH( ) END $,
             MOVE$    VAL(R)=VAL(Q) $,
                      IF(Q=(R=Q)-1) GRT P THEN GOTO MOVE $,
                      VAL(P)=0 $,
                      AFREZ=P
                  END
           END $,
```

By declaring AFREZ globally, it may be called from any block. A call to SET. AFREZ

SET. AFREZ(200, ANY. ARRAY) $,

will cause future AFREZ calls

AFREZ(13) $, AFREZ(147) $, etc.

to make the specified positions zero after moving the necessary values already present in ANY. ARRAY out of the way. Note that AFREZ assumes its argument specifies an already-occupied index. Can you fix it so that this is either checked or causes no problem? (If you choose the "causes no problem" challenge, you must somehow augment the concept of this kind of array-handling. I. e., it is not clear that feeding random integers between 0 and 200 to AFREZ, causing it to "make room", can perform anything useful without a major change of strategy and concept. <u>Is</u> there a "more general" AFREZ, or must we just patch it to check the validity of P?)

We may make a "no-NEXT" version of the ordered list problem in an almost-mechanical fashion as follows. Again we must only worry about declarations, initialization, and free storage, with the new concern about use of NEXT. The first change we encounter is to omit the declaration of NEXT:

```
INTEGER ARRAY VAL(200) $,
```

Then the initialization changes: There now is no need for a left-boundary object so

```
        NEXT(0)=MAXR=1 $, becomes
          MAXR=0 $,

  and  VAL(1)=1C35 $, becomes
          VAL(0)=1C35 $,
```

After reading in a new value, the free storage index and check of MAXR, which was moved up in the program, is ok. Then R=0 $, will properly refer to the initialization object, so we next consider Q=R $,.

Now, Q was used only to perform the splice in the bead strategy (and also in the mechanically produced first array version which did not change the strategy), so it can be omitted from the declarations and from the program. (This could be mechanically deduced from the fact that Q only plays an essential role in NEXT manipulations.)

Therefore Q=R $, becomes a mental note to use R in place of Q in what follows.

The next statement, R=NEXT(R) $, has three effects:

1.) In no-NEXT terms it becomes R=R+1 $,
2.) It causes us to alter the mental note to replace Q by R-1
3.) It forces us to reconsider the R=0 $, that we skipped over. It must now become R=-1 $, since the $-\infty$ object was left out.

The SCAN loop is satisfactory, but now we encounter the insert operation with R set to the lowest value that must be moved. Instead of using AFREZ(R) (which would index MAXR again) we write in the working statements themselves (expanded for clarity) after adding a new integer P to the declarations:

```
        P = MAXR ... START BACKWARD SCAN $,
MOVE$   VAL(P)=VAL(P-1) ... MOVE EACH VALUE $,
        P=P-1 ... COUNT DOWN $,
        IFP GRT R THEN GOTO MOVE $,
```

Now R is available for storing the new value in I. Note that old R is now R+1.

But let's see how the mechanical process confirms that I should be stored in VAL(R).

The previous program said

```
        NEXT(Q)=MAXR $,
```

referring to the new free index as MAXR. The above MOVE loop has made R the new free index. We have a mental note that Q is really R-1 so NEXT(Q)=(R-1)+1=R is R=MAXR, checks ok.

Next the previous program said

```
        Q=NEXT(Q) $,
```

so we again change the mental note to say that now Q is really R, as above. Then the previous program said

```
NEXT(Q)=R $,
```

i.e., NEXT(R)=old R=R+1, which checks.

So when the previous program said

```
VAL(Q)=I $,
```

our latest mental note that Q is really R says

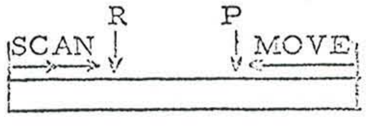
```
VAL(R)=I $,
```

as we expected.

So the "no-NEXT" version of the ordered-list program which again has been produced by essentially mechanical means is:

```
              BEGIN ... NO-NEXT VERSION //
              .INSERT MAIN $,
              INTEGER I, P, R,MAXR $,
              INTEGER ARRAY VAL(200) $,
              MAXR=0 $,
              VAL(0)=1C35 $,
        LOOP$ ISI(GIN(I),STOP) $,
              IF(MAXR=MAXR+1) GRT 200 THEN GOTO TROUBLE $,
              R=-1 $,
        SCAN$ R=R+1 $,
              IF VAL(R) LES I THEN GOTO SCAN $,
              P=MAXR $,
        MOVE$ VAL(P)=VAL(P-1) $,
              IF(P=P-1) GRT R THEN GOTO MOVE $,
              VAL(R)=I $,
              GOTO LOOP $,
    TROUBLE$ GOUT(.C. /OUT OF STORAGE/) $,
       STOP$ FINISH( ) END FINI
```

Once again we must examine the program to see if further honing is possible. In this case, since dropping out NEXT has changed the strategy somewhat, we can make further progress to take advantage of the properties of arrays. The relevant question to be asked is: Why count up for the scan and then count down for the move?



Sure enough, the SCAN and MOVE loops are complements of each other as far as locating R is concerned, so either one can do the work of the other. (Just as VAL(MAXR) LES 1C35 does the same job as the old NEXT(R) EQL 0).

The SCAN loop has only to be aware of the order, but the MOVE loop must be done as written in order to move the required values to "make room".

Therefore, the backward MOVE loop is the one to take as a basis.

So the new strategy is to move backwards through the VAL array until a value to be moved is LEQ the new value in I. That value will not be moved, the loop will stop, and I will be put in the vacant slot.

```
            BEGIN
            .INSERT MAIN $,
            INTEGER I,Q,R,MAXR $,
            INTEGER ARRAY VAL(200) $,
            MAXR=0 $,
            VAL(0)=1C35 $,
     LOOP$ ISI(GIN(I),STOP) $,
            IF(R=MAXR=(Q=MAXR)+1) GRT 200 THEN GOTO TROUBLE $,
     SCAN$ VAL(R)=VAL(Q) $,
            IF VAL(Q=(R=Q)-1) GRT I THEN GOTO SCAN $,
            VAL(R)=I $,
            GOTO LOOP $,
 TROUBLE$ GOUT(.C. /OUT OF STORAGE/) $,
     STOP$ FINISH( ) END FINI
```

This seems to be the best program for the revised strategy. Note that this "no-NEXT" version has the same rigidity as the previous array version, takes half the data space, but requires more running time due to the need to "make room" for new values. It takes less time than the previous mechanically derived "no-NEXT" version, but the scan loop of both of these versions has more operations than the scan based upon the bead strategy. Except for small numbers of values (in which case the FREZ overhead is significant) the bead strategy is faster, the first array version being the fastest so far developed here.

Notice also that this version is good for the ordered list problem as stated here, but that it does not work well if duplicate input values are to be discarded (as in symbol tables). The previous versions, which only exercise free storage when an insertion is to be made are all applicable, however.

If the frequency of duplicates is very high (as it is in the case of symbol tables) then it is important to improve the scan speed. The previous "no-NEXT" version will allow a logarithmic search to be applied between 0 and MAXR for this purpose.

The resulting scheme may even be preferable to a hash-coding scheme, since the list is always maintained in order.

The point of all this has not been to solve the ordered list example in even more ways, but to illustrate how the bead approach can be transformed into the array approach by an essentially mechanical process, including the elimination of a NEXT component.

The bead method, with its explicit calls in FREZ and FRET, shows precisely when objects come into existence and when they are no longer needed. The corresponding information is diffused in the array scheme.

The bead approach, being object-oriented is the closest match to the abstract mechanization-free concept of the problem. It yields the most complete overt expression of the problem, and thereby makes a good base for these essentially mechanical transformations.

Finally, the bead approach gives the most flexible debugging and running environment, and the transformation to array form even for a production system may not be warranted.

APPENDIX C

Suggested further reading:-

1. Douglas T. Ross. "The AED approach to generalized computer-aided design." Proceedings A.C.M. National Meeting, 1967. pp. 367-385.

2. Douglas T. Ross. "The AED Free Storage Package" CACM 10, 8 (August '67) pp. 481-492