A ring processing package for use with FORTRAN or a similar high-level language

Linden F. Blake, Rosemarie E. Lawson and I. M. Yuille*

* *Admiralty Research Laboratory, Teddington, Middlesex*

Many applications of a computer, particularly in engineering design, utilise relationships between blocks of data and require a facility for building and manipulating data structures which permit the expression of these associative relationships. This paper describes a software package that enables associative data structures to be represented in a computer store by means of rings of address pointers connecting blocks of data in an orderly manner. The package has been implemented on a KDF9 computer with a disc store operated by the Egdon 3 system. The software is written in machine code but by means of a set of small auxiliary routines operations may be carried out by calling FORTRAN subroutines. By this means manipulations of associative data occupying up to two million words may be included in FORTRAN programs. By writing different auxiliary routines the package could be used in ALGOL or another high level language. Implementation on another computer would not be difficult.

(Received April 1969)

Many modern applications of a computer, particularly in the fields of computer aided engineering design and computer controlled drawing or display, are concerned with collections of objects, each of which has properties represented by some data, arranged in such a way that the association of one object with another is explicit. For example, certain objects may possess some common property or they may possess some hierarchical structure such as a family tree. Data representing such a collection of objects is known as an associative data structure. It is required to represent an associative data structure in a computer store in such a way that it may undergo rearrangement and modification when required and so that the associative relationships may be determined easily when the data structure is entered at any point.

An association between two blocks of data in a computer store may be represented by adding to one block of data a word which contains a pointer to the address of a similar word added to the other block of data. By building on this concept multiple associations may be achieved by adding several words to each data block and by using many address pointers to link the blocks together in some orderly fashion. The simplest arrangement that will satisfy the requirements of the previous paragraph is one in which the pointers form rings that may be traversed from one block of data to another, eventually returning to the first, and to designate one position in each ring to be its starting-point. It is convenient, however, to have pointers that enable a ring to be traversed in either direction and also to make it possible to go directly to the start of a ring.

The various published implementations of this concept differ in their arrangement and handling of the pointers, in their methods of execution, and in the command languages by which the systems are used. A review of several schemes was published by Gray (1967). All required compiler type programs to translate commands, written in the chosen terminology, into machine code or, in the case of APL (Dodd, 1966) for example, into PL/1 statements and subroutine calls. There is as yet no standard terminology for describing associative data structures, or operations on them, and some of the published command languages are far from easy to use.

The object of the present work was to provide a ring processing software package that permitted the representation of large complex associative data structures which could be operated upon in a manner which was easy to program. There is a tendency, at present, for engineering applications programs to be written in FORTRAN (in the hope that they will run on different computers without having to be re-written) and it seemed to be desirable to provide a set of operations within the framework of a proven high-level language rather than a separate facility. For these reasons our ring processing routines, written in a low-level language, are accessible to a programmer by calls to FORTRAN subroutines. (The package uses small auxiliary routines to connect it to the FORTRAN and it would be easy to enable it to be used with another high-level language, for example, as a set of ALGOL procedures.) This obviates the need for a special compiler to include ring processing operations and permits the programmer to use the jump facilities of FORTRAN to embed the basic ring processing operations in sophisticated program loops if he so desires. The subroutines permit all required operations on the structured data but the detailed organisation of the rings is carried out automatically. The names given to the subroutines are concerned with ring processing operations and the interpretation in terms of the data structure represented depends entirely on the application.

The ring processor has been implemented on a KDF9 computer and is designed to fit into the Egdon 3 operating system (Poole, 1968). The system makes use of a four million word disc store and it was desired to use this to store large data structures which could be operated upon by running various programs in the computer. It was decided to hold each element of a ring in one 48-bit word. Although the use of forward, backward and ring start pointers in each element would have been desirable this would have restricted each address pointer to 14 bits and confined the total size of the structured data to less than 16,384 words. It was therefore decided to use 21-bit pointers which allow the data to occupy up to two million words if required. (Each element of a ring contains a forward pointer and either a backward or a ring start pointer as will be described in more detail later.) The data is held on pages in files on the disc store and pages are brought into the core store when required. By this means the ring processing package effectively permits the extension of the addressable memory within a FORTRAN program to about two million words.

Representation of data structure in computer store

The basic item of the representation of an associative data structure comprises a number of contiguous computer words and will be called an entity. Many different types of entity may exist in the same ring structure. A typical entity is shown in **Fig. 1.** The first word is the entity header and into this is packed information concerning its type, region, number and size. The next few words are elements of rings formed by address pointers as described in the next paragraph. (There is a maximum of seven of these, only six of which are available for general use.) The remaining words of the entity contain data specific to the object represented by that entity; these data may be alpha-numerical, e.g. the name of the object, and/or purely numerical data concerning one or more properties of the object, depending upon the application.

The arrangement of a ring is based on that of the CORAL language which was developed at the M.I.T. Lincoln Laboratory (Sutherland, 1966). This allows rapid tracing through the ring in a forward direction and permits tracing backwards or to the start of a ring with little extra effort. In our implementation each element of a ring is formed in one 48-bit word of KDF9 store and consists of four parts. Three bits identify the type of element, three bits form a number specifying its position in the entity and the remainder of the word contains two parts of 21 bits each. One of these parts holds the page and relative address of the next element in the ring. One element is the start of the ring and is called a ring start element; all other elements are subordinate to it and are called associative elements. In the ring start element the second 21 bits records the number of elements in the ring. In the associative elements the second 21 bits hold a page and relative address which points either to the ring start element or is used as a backward pointer. A backward pointer always points to an element with a back pointer and rings are formed with start pointers and back pointers alternating as shown in **Fig. 2.** The less useful pointers are thus stored in half the space but this involves only a small sacrifice of operation time.

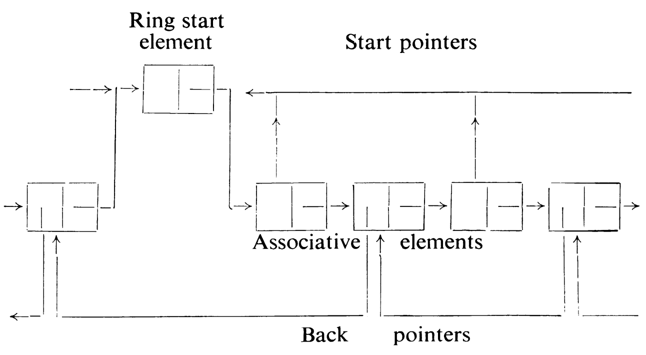

Fig. 2. Arrangement of ring

Entity header
Associative element in type ring
Associative element 1
Associative element 2
Ring start element 1
Ring start element 2
Data word 1
Data word 2
Data word 3

Fig. 1. Typical entity

The association between two entities is achieved by choosing the associative element, at a given position in one of the entities, and putting it into the ring starting at a given position in the other entity. A third entity may be associated with these two by putting one of its associative elements into the same ring and further entities may be added in the same way.

When it is desired to put one associative element of an entity into the rings starting in two or more other entities a problem arises because an associative element has only sufficient room for the pointers of one ring. This problem is handled automatically by the program by letting it join rings together by means of a device called a knot. (These were called nubs in CORAL.) A knot is a two-word block of store each word of which is an element of a ring. When an associative element of an entity is to be put into two rings it becomes the start element of an auxiliary ring of two associative elements, one in each of two knots. The other two elements of the two knots become associative elements in the two rings of which the entity is to become a member. If it is desired to place the entity in a further ring the number of knots is increased by one, and so on. Thus the effect of the auxiliary ring is to provide an extension of one associative element of an entity when (and only when) required. It is best thought of in this way so that the original associative element in the entity may be regarded as one associative element tied to more than one ring. A ring without knots will be referred to as a simple ring and one with knots will be called a complex ring. The creation, organisation and deletion of knots is handled entirely by the processing package and need not concern the applications programmer.

A small associative data structure is shown in **Fig. 3** and illustrates the use of knots to join rings. Entities *A*, *B*, *C* and *D*, are in a simple ring having its starting element in entity *I*. Entity 1 is in one ring starting in entity *A*. Entity 2 is in two rings starting in entities *A* and *B*. Entity 3 is in three rings starting in entities *B*, *C* and *D*. The associations represented in Fig. 3 are completely general and depend upon the meaning given to the data structure in a particular application. For example, in part of an information system entity *I* might represent a particular subject, entities *A*, *B*, *C* and *D* papers on that subject (with titles and sources held as data) and entities 1, 2 and 3 the authors of the papers (with their names held as data).

The limitation on the number of ring elements in an entity may appear to imply a limitation on the number of rings that can have their ring starts in a given entity, but this is not so in practice. For most applications this limitation will not be a restriction. If it is, however, the difficulty may be overcome by defining an entity with up to five ring start elements and by putting one or more of these entities into a ring starting in the original entity thus effectively extending indefinitely the number of ring starts there. If the latter is to have a large number of associations (as is implied) it may be convenient to include in the extra entities some words of data for identification or they may be identified solely by their positions round the ring starting in the original entity.

Organisation of storage

The system is primarily intended to run on a computer with disc as well as core store. (There is, however, a version of the software that uses only the core store.) For the KDF9 computer a software paging supervisor enables the whole of the available store to be addressed in one way. A suitable amount of core store is set aside for manipulations of the data structure and into this pages may be read from the disc. The core store set aside must be able to accommodate at least six pages but the system will run more efficiently if more pages can be accommodated. Each 21-bit address pointer, to a ring element or an entity, specifies a page number and the address relative to the beginning of the page. When a pointer is encountered the paging supervisor examines its list of pages in core. If the required page is present the page and relative address are converted into the appropriate core address. If the page is not present it is read into a free part of the array allocated to pages in core store. Then the page in core which has remained unused for the longest time is read back to the disc (except in certain circumstances to be mentioned later) leaving a free space in the array for the next page required from the disc. While the unwanted page is being written onto the disc the pages in core may be used with little interruption to the program.

After many manipulations of a data structure it may

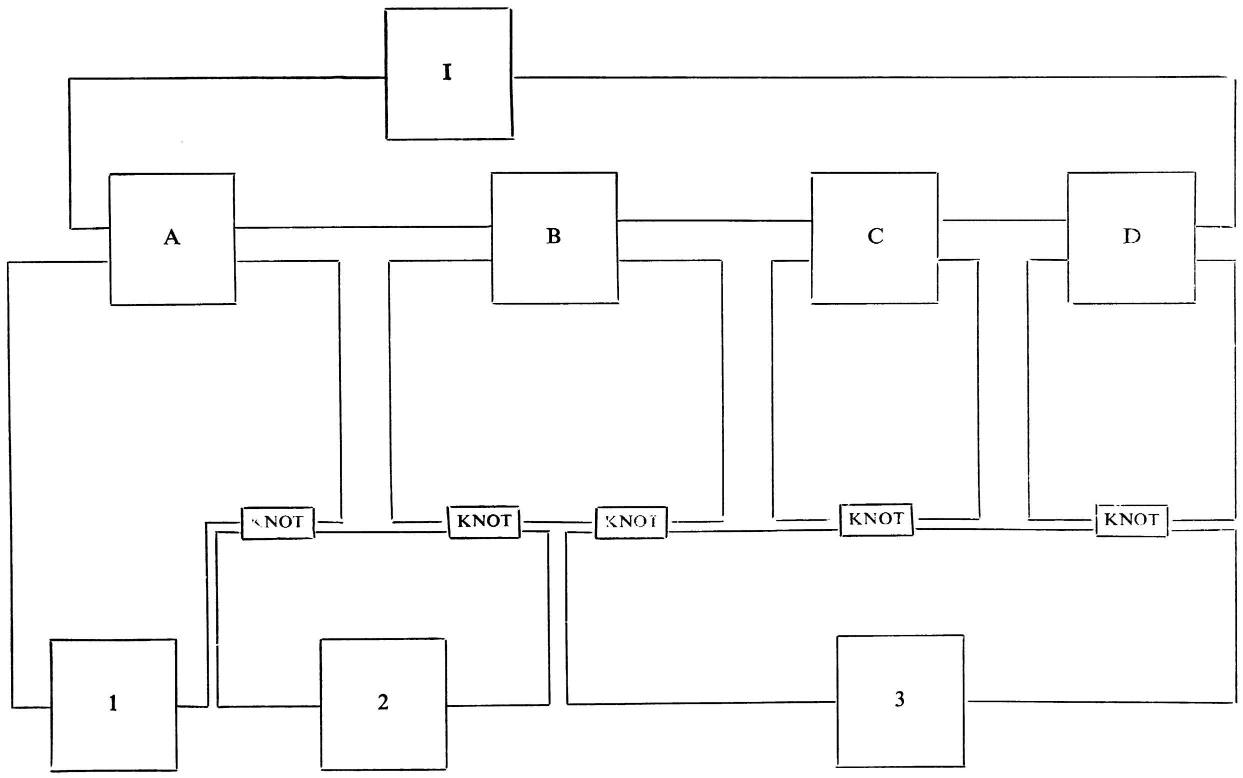

Fig. 3. Typical ring structure

be anticipated that linked entities may be scattered over many pages. To increase efficiency, entities may be made in any of 63 regions and the pages for one region only hold entities in that region. Calls for entities in one region will bring into core pages holding other entities in that region. Thus, by setting up the representation of a data structure so that the most frequently used associations occur within particular regions the programmer can decrease the probability that further pages will have to be read from the disc before operations on the ring structure may continue. Also, in some manipulations of a data structure it is convenient to set up an auxiliary structure which is later discarded. If this is set up in a separate file of pages its deletion, when no longer required, does not involve the creation of gaps in the main ring structure which have to be removed by the data compacting routine.

On the KDF9 disc store the pages of each region are held in a separate file in the Egdon 3 system. In addition to the obvious advantage, that the user has all the facilities of the Egdon archiving system, this helps the operation of the Egdon system during which, as a safeguard against disc failure, a periodic dumping to magnetic tape is carried out of all files that have been altered since the last dumping. With a ring structure held in several files only those regions that have been changed need to be dumped.

Facilities available

The software comprises a User Code package plus some small auxiliary routines that enable operations on the ring structure to be carried out by calling FORTRAN subroutines in the normal way. Thus programmers are able to include operations on associative data in programs written in a high-level language. There are facilities for:

(*a*) Creating and deleting entities.
(*b*) Establishing and altering the associative links between entities.
(*c*) Tracing round rings of entities.
(*d*) Interrogating entities.
(*e*) Obtaining access to data held in the data parts of entities.
(*f*) Compacting the data structure so that it occupies the smallest possible space.

The subroutines that provide these facilities are summarised in the following sections. The subroutines require the programmer to specify a number of integer variables, or their values, in the usual way. The definitions of these are given below together with the names allocated to them in this description.

JA =	Position of associative element in entity.
JD =	Position of data word in entity.
JS =	Position of ring start element in entity.
KEA =	Entity page and relative address. There must be at least three variables KEA1, KEA2, KEA3, . . . which the system can use to store address pointers to entities. These are the normal means of referring to an entity between subroutine calls.
LEN =	Entity number.
LT =	Type of entity.
LR =	Region number.
MD =	Position of a word in the array holding the associative data in core store relative to beginning of array.
NA =	Number of associative elements defined to be in entity of given type.
ND =	Number of data words defined to be in entity of given type.
NS =	Number of ring start elements defined to be in entity of given type.
N =	Number of steps to be taken.
NOT =	Number of steps not taken.
NEN =	Number of entities.
NRG =	Number of rings.

Creation and deletion of entities

DEFINE (LT, NA, NS, ND)
Define the number and type of elements in entities of type LT.

CHAD (LT, ND)
Redefine the number of data words in entities of type LT.

MAKE (LT, LR, LEN, KEA)
Create an entity of type LT in region LR. The number of the entity will be stored in LEN and its page/relative address will be stored in KEA.

FREE (KEA)
Return to the system the space which was occupied by the entity stored at the address in KEA.

Associative links

PUT (KEA1, JS1, KEA2, JA2)
The element JA2 of the entity at KEA2 is inserted at the beginning of the ring starting at element JS1 of the entity at KEA1.

PUTEND (KEA1, JS1, KEA2, JA2)
As PUT but element JA2 of the entity at KEA2 is inserted at the end of the ring starting at element JS1 of the entity at KEA1.

SWOP (KEA1, JS1, KEA2, JA2, KEA3, JA3)
Interchange the element JA2 of the entity at KEA2 with the element JA3 of the entity at KEA3 in the ring starting at element JS1 of the entity at KEA1.

TAKE (KEA1, JS1, KEA2, JA2)
The element JA2 of the entity at KEA2 is removed from the ring starting at element JS1 of the entity at KEA1.

Movement in rings

TYPSTP (LT, LR, KEA, N, NOT)
Take N steps round the entities of type LT in region LR from the entity at KEA and store in KEA the address of the entity so reached. If KEA = 0 when the CALL is executed stepping will be from the ring start. (If the ring start is reached before the stepping is completed NOT is set to the number of steps not taken and KEA set to zero.)

ENTSTP (KEA1, JS1, KEA2, JA2, N, NOT)
Take N steps from the entity at KEA2 element JA2 round the ring starting in element JS1 of the entity at KEA1. KEA2 becomes the address of the entity so reached and JA2 the number of the required associative element in that entity. If KEA2 = 0 at the CALL stepping is from the ringstart KEA1, JS1. (NOT and KEA2 as for TYPSTP.)

RNGSTP (KEA1, JA1, KEA2, JS2, N, NOT)
Take N steps round the entities which start rings containing the associative element JA1 of the entity at KEA1 beginning from the element JS2 of the entity at KEA2. The address of the entity reached will be stored in KEA2 and JS2 will become the number of the required ring start element within that entity. If KEA2 is set to zero before the CALL stepping will be from the associative element itself. (NOT is set to the number of steps not taken if all the rings are found before stepping is completed and KEA2 is set to zero.)

GET (LT, LR, LEN, KEA)
In KEA record the address of entity number LEN of type LT in region LR. (If there is no entity of this number and type in the region KEA will be set to −1.)

Interrogation of entities

INFO (KEA, LT, LR, LEN, ND)
Record the type, region, number and number of data words of the entity stored at the address in KEA in LT, LR, LEN and ND respectively.

RINGS (KEA, NA, NS)
Record the number of associative and ring start elements in the entity stored at KEA in NA and NS respectively.

ENDATA (KEA, JD, MD)
Record in MD the position in the page array of the JDth word of data of the entity stored at KEA.

ENTT (LT, LR, NEN)
Record in NEN the number of entities of type LT in region LR.

ENTR (KEA, JS, NEN)
Record in NEN the number of entities in the ring starting at element JS of the entity at KEA.

ENTA (KEA, JA, NRG)
Record in NRG the number of rings to which the element JA of the entity at KEA belongs.

Discussion of subroutines

Each of the subroutines is used by treating it as a FORTRAN subroutine and calling it in the usual way. Up to 127 different configurations of entity may be defined for a particular application and any number of any type may be made as required in any of the 63 regions. For over a year the ring processing package has been in use in connection with a project for computer aided design of ships and it has been found that the use of fixed size entities has helped rather than hindered this work. Occasionally, however, it is necessary to vary the number of words of data held in entities of a given type and this is done by calling CHAD before MAKE each time an entity of that type is made.

The address pointers KEA are used in nearly all the routires. These consist of a page and relative address which permit the recording of the address of any entity on the disc. These pointers are held as FORTRAN variables so there is no restriction on their numbers and they may be recorded or deleted according to the needs of the program.

The routine GET does not alter the pages through which it searches for the required entity and special provision has been made to speed up its operation. Successive pages read from the disc overwrite each other in core until the appropriate page is found and no pages are written back to the disc. The possibility of finding an entity even though it is not in any ring is not usually provided in ring processors but has been found to be quite useful in the computer aided ship design project mentioned earlier.

Two routines enable entities to be placed at the beginning or end of a ring. The use of SWOP with appropriate FORTRAN programming permits an entity to be placed at any specified position in a ring without upsetting the alternation of the backward and ring start pointers. For example, the following puts the entity whose address is held in KEA2 into the ring starting at the entity whose address is held in KEA1 at a position in the ring *after* the entity whose address is held in KEAB (known to be in the ring).

```
  CALL PUT (KEA1, JS1, KEA2, JA2)
1 KEA3 = KEA2
  JA3   = JA2
  CALL ENTSTP (KEA1, JS1, KEA3, JA3, 1, NOT)
  CALL SWOP (KEA1, JS1, KEA3, JA3, KEA2, JA2)
  IF (KEA3.  NE.  KEAB)1
```

There is a notable absence of what are usually called 'GO ROUND' facilities. In fact these are unnecessary because any such facility may be built up by writing appropriate FORTRAN programs around the facilities available in the package. (Provision of some simple 'GO ROUND' facilities was contemplated but examination of those provided in other work of this nature showed them to be of restricted application and it was preferred to keep the package as small as possible so that transfer to another computer would be simplified.) To illustrate the ease with which FORTRAN programs may be written to carry out complex operations on data structures the subroutine given in **Table 1** carries out one of two different operations upon each entity of any ring structure starting at element JS in the entity whose address is held in KEA; which operation is executed depends upon whether or not the entity under consideration is in a ring starting elsewhere. The two different operations are named FUNCA and FUNCB respectively.

With the array dimension 20 as shown this routine operates on ring structures to a depth equal to 20, but of course this figure could easily be changed. The subroutine could have been written to operate recursively, but this was not done because FORTRAN IV does not include recursion. No other published ring structure package known to the authors permits the writing of such complex ring structure operations with such ease.

If the two functions are as follows:

```
SUBROUTINE FUNCA (KEA, JS, KE, JA)
CALL TAKE (KEA, JS, KE, JA)
CALL FREE (KE)
RETURN
END

SUBROUTINE FUNCB (KEA, JS, KE, JA)
CALL TAKE (KEA, JS, KE, JA)
RETURN
END
```

each entity of the ring structure starting in the element JS of the entity whose address is KEA will be deleted unless it is also a member of another ring structure. For example if, in **Fig. 4,** KEA was the address of entity A and JS = 1 all the entities marked X would be deleted.

The subroutine ENDATA (KEA, JD, MD) gives the user access to the data part of the entity whose address is held in KEA by assigning to MD the position, in the array holding the associative data in the core store, of the JDth word of data in the entity. Data may be taken from the array by any legitimate FORTRAN statement and similarly may be put into the array (i.e. into the data part of the entity). It is the programmer's responsibility to ensure that such operations are carried out only within the data area of the entity, when it is in core at a known address, (using INFO if necessary).

The software also contains facilities for initiating the system, for error tracing, and for deleting or archiving complete data files. It is possible for one program to set up a data structure and for other programs to be run subsequently to alter the data in various ways. These facilities are not described here.

Table 1

The subroutine PEDOR

```
         SUBROUTINE PEDOR (KEA,JS,FUNCA,FUNCB)
         EXTERNAL FUNCA,FUNCB
         DIMENSION KE(20),KRSE(20),KAE(20),NE(20),NRSE(20),NAE(20)
         M=1
         KE(1)=KEA
         KRSE(1)=JS
         NRSE(1)=1
         KAE(1)=0

C        FIND NUMBER OF ENTITIES IN RING
       1 CALL ENTR (KE(M),KRSE(M),NE(M))
C        TEST IF RING EMPTY
         IF (NE(M).EQ.0) 2
         M=M+1
         KE(M)=0
C        FIND NEXT ENTITY IN LOWER LEVEL
         CALL ENTSTP (KE(M-1),KRSE(M-1),KE(M),KAE(M),1,NR)
       7 CALL RINGS (KE(M),NAE(M),NRSE(M))
         N=NAE(M)
C        TEST IF ENTITY IS A MEMBER OF OTHER RINGS
         DO 5 I=1,N
         CALL ENTA (KE(M),I,MSR)
         IF (MSR.EQ.0)5
         IF ((I.EQ.KAE(M)).AND.(MSR.EQ.1)) 5
C        ENTITY IS MEMBER OF ANOTHER RING
         ASSIGN 11 TO L
         GOTO 8
       5 CONTINUE
C        TEST IF ALL RING STARTS HAVE BEEN CONSIDERED
         IF (NRSE(M).EQ.0) 3

         KRSE(M)=1
C        WORK ROUND NEXT RING
         GOTO 1

       2 NRSE(M)=NRSE(M)-1
C        JUMP IF ALL RING STARTS CONSIDERED
         IF (NRSE(M).EQ.0)4
         KRSE(M)=KRSE(M)+1
         GOTO 1

C        TEST IF ORIGINAL ENTITY REACHED
       4 IF (KE(M).NE.KEA) 3
         RETURN
C        ENTITY ONLY ATTACHED TO RING STARTING IN GIVEN ENTITY
       3 ASSIGN 10 TO L
       8 KTS=KE(M)
         KTA=KAE(M)
C        FIND NEXT ENTITY AT SAME LEVEL IF ANY
         NE(M-1)=NE(M-1)-1
         IF (NE(M-1).EQ.0) 9
         CALL ENTSTP(KE(M-1),KRSE(M-1),KE(M),KAE(M),1,NR)
       9 GOTO L,(10,11)
      10 CALL FUNCA(KE(M-1),KRSE(M-1),KTS,KTA)
         GOTO 12
      11 CALL FUNCB(KE(M-1),KRSE(M-1),KTS,KTA)
C        TEST IF OTHER ENTITIES AT SAME LEVEL
      12 IF (NE(M-1).NE.0) 7
         M=M-1
         GOTO 2
         END
```

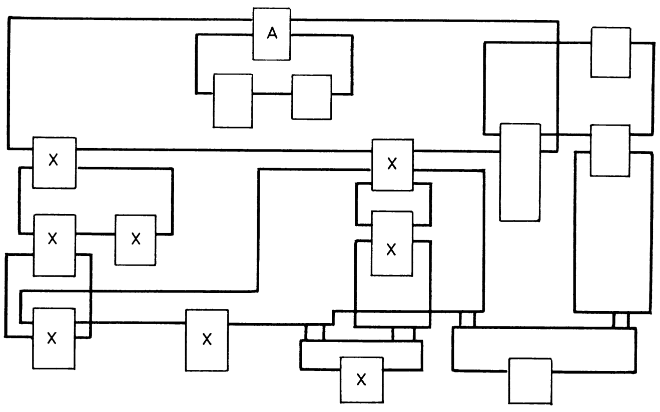

Fig. 4. To illustrate operation on a complex ring structure

Organisation of free space and compacting of data

After operations on a ring structure involving the use of subroutine FREE there will, in general, be unoccupied spaces between entities on the pages. Although these spaces are used first, when entities are made, after a time the free spaces will gradually accumulate and the data will occupy more space than it needs. It is therefore necessary to have a routine for compacting the data into a smaller space. This is a time-consuming operation because the entities must be moved and each time this happens all the pointers to the entity must be found and changed. It is therefore desirable to run the compacting routine only when it appears to be necessary rather than, say, at the end of each run of a program which changes the data. In order to discuss the operation of the compacting routine it is first necessary to describe in more detail the organisation of free space within the pages.

The first word on each page is an associative element in a ring of pages in a given region. The second word on each page holds the region (6 bits) and the number (11 bits) of words in the largest block of free spaces on that page and starts two rings through the first words of all the blocks of free space on the page. These rings consist of pointers comprising 11-bit addresses relative to the beginning of the page. One of the rings connects the blocks in order of increasing size and has only forward pointers. The other ring connects the blocks in the order in which they are found on the page and has forward and backward pointers. A further 11 bits in the first word of each free space block are used to indicate the number of words in the block.

When subroutine MAKE is called a search is made through active pages in the specified region for a page having a block of free words large enough to hold the new entity. The headers of pages of the required region that happen to be in the core store are examined first and only then is the search continued, if necessary, by reading pages from the disc. No alteration is made to a page if it has not enough free space so, to save time, successive pages read from the disc overwrite each other in the core store and none is written back to the disc. (The number of pages searched before claiming a new page may be specified by any particular applications program.) If none of these pages has enough free space for the new entity a new page is taken from the ring of free pages in the region and put into the ring of active pages. When a suitable page is found the ring of free space in it is searched in order of increasing size. The smallest suitable block of free words is made into an entity and the rings of free space on the page are amended accordingly. When subroutine PUT or PUTEND is called and it is necessary to create a knot there is a similar procedure, in the region of which the entity being put is a member, starting with the page holding that entity. Subroutine FREE simply replaces the entity header by the first word of a block of free space, sets the size of the block in this word, and inserts it into appropriate places in the rings of free space on the page.

The compacting routine is activated by calling subroutine COMPAC. It operates in two stages. In the first stage a number NC of pages to be used is specified. In general, considering each entity on the current page in turn, the compacting routine steps back NC pages through the ring of active pages in the region ignoring pages that are full. The entity or knot being considered is moved to this page if there is room for it and all the pointers in other entities or knots in rings of which it is a member are changed accordingly. If there is no room on a page the compacting routine steps forward to the next page (ignoring full pages) and continues to do so until the current page is reached. It then deals with the next entity or knot on the current page and so on until it has attempted to move all the entities from the current page to a vacant space on one of the NC pages before it. If the current page is completely free as a result of these operations it is thereafter excluded from the compacting operation by taking it from the ring of active pages in the region and putting it into the ring of free pages in the region. If not, the entities left on the page are moved to the top of the page and the remaining free space combined into one large block. The next page then becomes the current page for the compacting routine. If NC is set to zero the routine only compacts the entities within the current page itself and combines any separate free spaces in the page.

Each file on the disc contains blocks of pages. During the second stage of the compacting procedure a test is made to determine if it is possible to accommodate all the active pages in the region in less blocks than are currently allocated to it. If it is possible the active pages in the last *n* blocks, where *n* is the number of blocks no longer required, are moved onto the free pages in the earlier blocks. The file is then made smaller by *n* blocks. If the size of the file cannot be decreased the second stage of the compacting process is not carried out.

Comparison with other similar work

Ours is a ring processing package in which the construction of rings is automatically carried out and it cannot therefore be compared with low-level packages such as L^6 (Gray, 1967) with which the user creates his own rings from primitive building blocks, nor can it be compared with packages such as AED or those which use hash coding techniques. Our debt to the CORAL ring structure has already been mentioned. The only systems similar to ours and known to be in general use are APL (Dodd, 1966) and ASP (Lang and Gray, 1968) so attention will be confined to these two.

Both ASP and APL use only one type of entity to store data and the number of words of data in each entity (element in ASP) must be declared each time one is created. In most applications, however, many identical entities exist in a data structure and there are usually only a few distinct types each type having a different data length. This led us to arrange for the definition of up to 127 distinct entity types each having a fixed number of data words but to allow the number of words of data to be changed (by calling the CHAD subroutine) before an entity is made if this is required. This has been found to be a convenient arrangement.

If we represented an ASP structure by means of our package each entity (ASP element) would have only one associative element and only one ring start element and an arbitrary amount of data. We would also need to define one other type of entity, with one data word for name and type, to represent the 'ring starts' of ASP (let us call this a ring start entity). Our knots would correspond to the ASP associators except that in ASP there would be a knot even if the entity was only in one ring. Each time an ASP element (or entity in our terminology) was put in a ring to associate it with another element, a ring start and an associator (in our terminology a ring start entity and a knot) would have to be created and the address pointers between all four items would have to be set up. **Fig. 5** shows the ASP structure corresponding to the one shown in Fig. 3. In Fig. 5 a triangle represents an ASP ring start and a circle represents an ASP associator. It is clear that ASP requires many more address pointers than our package, particularly for simple data structures, and the ASP ring starts and associators might be set up on different pages from those on which the elements they connect are placed. In practice, we have found that most data structures set up for computer aided design require only three or four rings (often only one) to start in each entity. This led us to restrict the total number of ring start and associative elements in a given entity to six so that, in the absence of ring start entities and with the provision of knots only when necessary, the data structures set up would occupy less space, and be processed more quickly, than with a package corresponding to ASP. (Should one require more ring starts than an entity can hold it is possible to

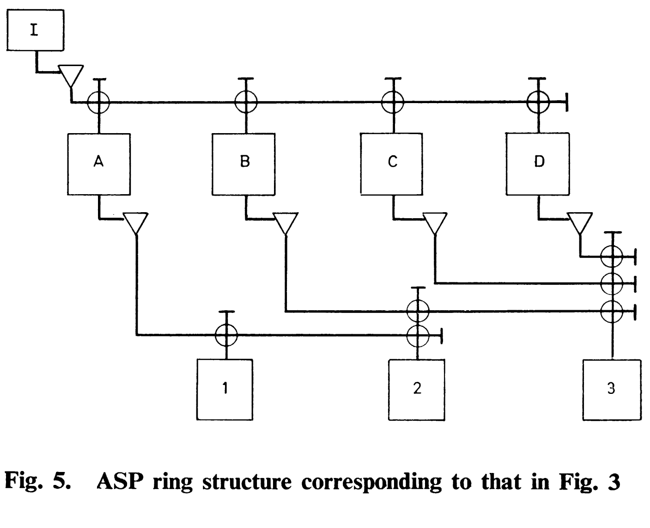

Fig. 5. ASP ring structure corresponding to that in Fig. 3

use the equivalent of ring start entities to extend the number, as described above under the heading **Representation of data structure in computer store.**) In this respect, therefore, our package resembles more closely the arrangement used by APL.

With our package an entity can be found (by GET or TYPSTP) even though it is not in any ring. This facility is not available in APL and in ASP an entity is automatically deleted if all its associations are removed.

APL statements are written into PL/1 programs and the programs must then be run as data for a preprocessor which outputs PL/1 statements and subroutine calls. This output can then be processed by a PL/1 compiler. With our package the FORTRAN (or other high-level language) statements and subroutine calls are written directly, thus removing the need for a preprocessor (but certain other facilities provided by the APL preprocessor —not concerned with ring processing—are not present). This arrangement has been found to suit programmers already experienced with FORTRAN and some quite complex ring processing operations have been written. ASP runs with a compiler and has been implemented on Atlas II using the Mixed Language System available on that computer so that programs written in different languages may be compiled into a common format. Thus on that computer ASP commands could be embedded in programs written in a high-level language, as in our system, but transfer to another computer would be more difficult. It appears that ASP has not been implemented with any backing store facilities.

Concluding remarks

The package includes all the necessary basic subroutines to enable FORTRAN programs to include complex ring processing operations by use of the ordinary FORTRAN facilities. The paging software allows up to two million words on the KDF9 disc store to be directly addressed by FORTRAN programs. The package includes comprehensive facilities for compacting a data structure from which data has been deleted. It has been thoroughly tested and, as the routines became available in 1968, they have been used in a number of programs being developed for on-line computer aided design of ships and the package is regarded as fully proved.

In different applications pages may differ in size from 40 words to 2,048 words (but must, of course, be of constant size for a given date structure). A large page will be read from the disc in only slightly longer time than a small page but may bring with it more redundant information. The proper use of the regions to hold different parts of a data structure should, however, make the use of large pages comparatively efficient. It is intended to carry out studies of the effect of page size and the use of regions on the performance of the package in various circumstances. These studies will take some time and the results will be reported later.

The connection of the package to FORTRAN is via a number of short subroutines. These could very easily be rewritten to connect the package to another high-level language such as ALGOL. The package, which took less than 2 man years to develop, has been kept as small as possible. (It occupies about 2,000 words of KDF9 User Code and is not a compiler type program.) It would not be difficult to have it rewritten to run on another computer and this is all that would be necessary to permit FORTRAN programs which include ring processing operation, to be run on a computer other than the KDF9.

Acknowledgement

The authors are grateful to the staff of the Computer Technology Department of the General Motors Corporation, Warren, near Detroit, for valuable discussions during a visit by one of the authors (I. M. Y.) to their Research Laboratories on 13 April 1967.

References

Dodd, J. C. (1966). APL—A Language for Associative Data Handling in PL/1. Proceedings Fall A.C.M. Joint Computer Conference, Nov. 1966.

Gray, J. C. (1967). Compound Data Structure for Computer Aided Design; A Survey. Proceedings A.C.M., National Meeting, 1967.

Lang, C. A., and Gray, J. C. (1968). ASP—A Ring Implemented Associative Structure Package, CACM, Vol. 11, No. 8, p. 550.

Poole, P. C. (1968). Some aspects of the Egdon 3 Operating System for the KDF9. IFIP Congress, Software 2, p. C43.

Sutherland, W. R. (1966). The CORAL Language and Data Structure. M.I.T. Lincoln Laboratory, Technical Report 405.