Management/ Data Base Systems
H. Morgan Editor

Structured Data Structures

Ben Shneiderman
Indiana University
and
Peter Scheuermann
State University of New York at Stony Brook

Programming systems which permit arbitrary linked list structures enable the user to create complicated structures without sufficient protection. Deletions can result in unreachable data elements, and there is no guarantee that additions will be performed properly. To remedy this situation, this paper proposes a Data Structure Description and Manipulation Language which provides for the creation of a restricted class of data structures but ensures the correctness of the program. This is accomplished by an explicit structure declaration facility, a restriction on the permissible operations, and execution-time checks.

Key Words and Phrases: structured programming, data structures, data base management system

CR Categories: 3.50, 3.51, 3.72, 3.73, 3.79, 4.20, 4.22

Authors' addresses: Ben Shneiderman, Department of Computer Science, Indiana University, 101 Lindley Hall, Bloomington IN 47401; Peter Scheuermann, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11790.

1. Introduction

Much attention has been focused lately on the notions of structured programming, a crucial factor when dealing with the design of large programming systems. Rather than viewing programming as an art and the programmer as the perpetual artist, in structured programming we are provided with a more systematic (and in a way more restricted) approach which facilitates debugging and proving assertions about programs.

One of the ideas developed by advocates of structured programming is the top-down elaboration of program control structures by a recursive process of successive refinements [7, 15]. No such process has been developed for dealing with data structures. The main reason for this is the improper intermixing of the semantic and the implementation concepts [3] of a data structure. The failure to distinguish between these concepts is a result of the conflicting factors involved in choosing a data structure: simplicity of element access, minimization of search time, dynamics of growth or elimination of data, simplicity of restructuring and extension, efficiency of storage utilization, and others.

With these in mind we propose a "structured" Data Structure facility, which we call a Data Structure Description and Manipulation Language (DSDML) to maintain a similar terminology to other groups (see CODASYL Report [1]). The DSDML provides data structure definitions in addition to the data definitions available in the host language (e.g. PL/I or COBOL). It will include explicit declarations of commonly used data structures and information about their access and manipulation characteristics. These characteristics include such features as reset pointers or end pointers and search rules.

The main advantages of such a facility can be summarized as follows.

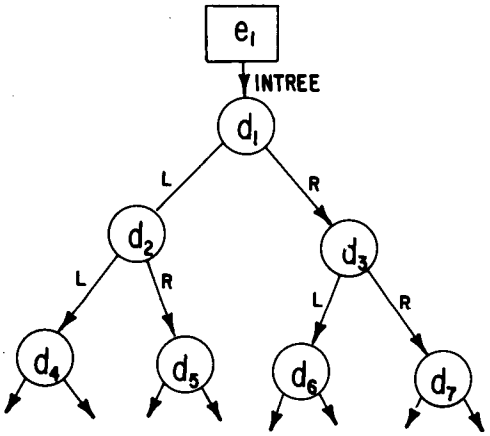
1. It provides (some) control mechanism over the behavior of data structures. Just as the "go to" is considered harmful to modular programs [2], in dealing with data structures, we want to eliminate unrestricted branches or edges. The permissible operations in the DSDML are more restricted than those in the CODASYL Report, but allow the creation of a wide variety of commonly used structures. The DSDML will be a useful tool in verifying that our structures are indeed "well formed" (i.e. enabling us to prove assertions about data structures). Declarations of variables in a programming language provide the compiler with the information necessary to prevent incorrect mixed mode operations from occurring. In FORTRAN, for example, one can declare variables to be REAL, INTEGER, LOGICAL, DOUBLE PRECISION, or COMPLEX. In a similar way, the DSDML (see Appendix) will prevent the programmer from mistakenly converting a binary tree into a three-way tree or from inserting a queue where a ring is expected or from obtaining undesired cycles, and so on.

The semantics will include provisions to prevent invalid operations. For example, if a one-way list with a bottom pointer and with reset pointers (pointing back to the first node from each node) is defined, it is not possible to make insertions along the bottom pointer or reset pointers.

2. Top-down programming can be achieved in terms of data structures, too. This follows from the fact that the DSDML allows for definitions of multilevel data structures in which the nodes of a given level structure serve as headers for the structures at the next level. Take, for example, the following application program involving a telephone system. A number of telephones are connected to a switchboard by lines. The switchboard has a number of links which can connect any two lines, but only one connection at a time can be made to each line. Assume that initially we disregard blocked calls; i.e. calls are lost if no links are available (obviously, the case is that there are more lines than links). The programmer might decide he needs a one-way list to keep track of all the free lines and run the program for this case. Later on, he might want to take into account the blocked calls and associate with every free line a waiting list of calls which could not be serviced because no link was previously available. In this case the desirable structure would be a one-way list of one-way lists. As mentioned above, the DSDML allows this kind of refinement by providing a multilevel structure, in which a higher level node represents the entire lower structure emanating from it. If the higher level routines have been written with sufficient generality and have been debugged, they will work for the refined structure. We need now only pay attention to the lower level routines.

3. It might be possible to obtain a more optimal storage allocation. By providing a data structure definition, the compiler (or run-time package) may have the ability to allocate storage in a more efficient way than the usual "space available stack" technique employed for most dynamic storage allocation schemes. For example, by declaring a binary tree with reset pointers, some information is provided about the amount of storage necessary for the nodes of the structure.

Fig. 1.



2. Mathematical Preliminaries

Before proceeding further with the DSDML, we present our view of data structures which follows the graph-theoretic model developed in [14]. For the reader unfamiliar with graph theory see Harary [4].

We formally associate with every data structure a quadruple G:

$G = (E,D,L,\Gamma)$ where

(1) E is a set of entry nodes, which are instantly accessible.

(2) D is a set of data nodes containing the information

(3) L is a set of labels, indicating the different accessing branches.

(4) Γ is a mapping which describes the relationship between the nodes and is defined as: $\Gamma : (E \cup D) \times L \rightarrow D$.

Note that Γ is a partial function that is one-one and onto (i.e. a partial bijective function).

As an example of this notation, consider the fully balanced binary tree in Figure 1. Here,

$$E = \{e_1\}$$
$$D = \{d_1, d_2, \ldots, d_n\}, \quad n = 2^k - 1$$

for some integer k, which represents the number of levels in the tree

$$L = \{INTREE,L,R\}$$

The mapping Γ is given by:

$$\Gamma\langle e_1, INTREE\rangle = d_1$$
$$\Gamma\langle d_i, L\rangle = d_{2i}, \quad \text{if} \quad i < 2^{k-1} - 1,$$
$$= \phi, \quad \text{if} \quad i \geq 2^{k-1},$$
$$\Gamma\langle d_i, R\rangle = d_{2i+1}, \quad \text{if} \quad i < 2^{k-1} - 1,$$
$$= \phi, \quad \text{if} \quad i \geq 2^{k-1},$$

ϕ being the null element.

Furthermore, the restriction is made that we deal only with well-formed structures (see [11]). $G = (E, D,L,\Gamma)$ is a well-formed structure if and only if:

(1) $E \cup D$ is a connected set of nodes, that is, if we consider the undirected graph which corresponds to G, there is a path between every pair of nodes; and

(2) E is a nonempty set with each node in E having in-degree zero (searching has to start at some entry point).

Formally,

$$E \neq \phi \;\&\; \forall e \,\forall l (e \in E \;\&\; l \in L \supset \Gamma^{-1}\langle e, l\rangle = \phi).$$

Note that the set D can be empty. This allows us to take into account cases like that of a queue with no elements in it at a given moment.

(3) Any node in D is reachable from at least one node in E. The transitive closure of Γ restricted to $(E \times L)$ yields D:

$$\bigcup_{i=1}^{n^*} \Gamma^i \,|_{(E \times L)} = D$$

where n^* is the smallest integer for which the equality holds.[1]

This condition is meant to exclude structures such as Figure 2 which contain unreachable nodes.

When dealing with "structured" data structures of the form $G = (E,D,L,\Gamma)$, the limitation $|E| = 1$ is imposed. All such structures thus have a single entry point which is always available for access by invoking the name of the structure. This restriction is necessary when defining multilevel structures, described later in more detail.

We now define the *uniform insertion* of two structures. This forms the basis for the idea behind the formation of multistructures.

Given two data structures $G_1 = (E_1, D_1, L_1, \Gamma_1)$ and $G_2 = (E_2, D_2, L_2, \Gamma_2)$, the uniform insertion of G_2 in G_1 denoted by: $G_1 \lhd G_2$ is:

$$G = G_1 \lhd G_2 = (E_1, \{D_1 \cup D_2\}, \{L_1 \cup L_2\}, \Gamma)$$

where $|E_1| = |E_2| = 1$ and Γ is defined as:

(i) on the set $(E_1 \cup D_1) \times L_1$:

$\Gamma\langle e_1, l_j\rangle = \Gamma_1\langle e_1, l_j\rangle$, if $e_1 \in E_1$ and $l_j \in L_1$,

(ii) on the set $(D_2 \times L_2)$:

$\Gamma\langle d_j', l_k'\rangle = \Gamma_2\langle d_j', l_k'\rangle$, if $d_j' \in D_2$, $l_k' \in L_2$,

(iii)

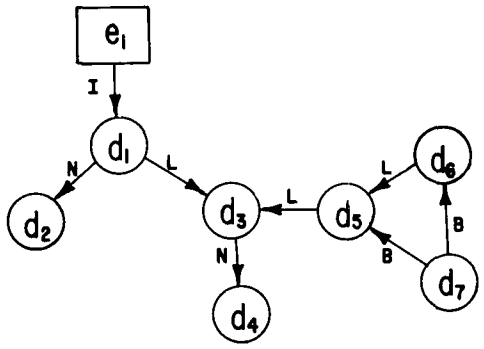
$\Gamma\langle d_i, l_j\rangle = \Gamma_2\langle e_2, l_j\rangle$, if $d_i \in D_1$, $l_j \in L_2$,

On the "boundary" between G_1 and G_2, Γ maps the elements of D_1 into the elements of D_2. The entry nodes of G_2 get "absorbed" by each of the data nodes of G_1. The branches emanating from the data nodes of G_1 bear the same labels as the branch emanating from the entry node of G_2.

3. Overview of the Data Structure Description and Manipulation Language

The facility described below is meant as an addition to the host language in much the same way that the CODASYL group [1] proposes to extend currently existing programming languages. The DSDML includes declarations for linear structures (one-way lists, two-way lists, rings, stacks, etc.) and tree, (binary, 3-ary, etc.) together with their access and manipulation characteristics (bottom pointer, back pointer, searchable option, etc.) and declarations for the operations on data structures (insert, delete, etc.). Combinations of these structures are also possible, such as one-way lists of trees, a ring of stacks, etc., and this is done in a PL/I-like mode by declaring different levels of nodes. These combinations are called *multistructures.*

Fig. 2.



A fundamental concept in the system is that of an interval, which is closely associated with the definition of the delete operation. An interval of a node d_i consists only of that node for linear structures. For trees, the interval is given by specifying the node d_i and a branch label l_i. The subtree rooted at the successor of d_i along l_i, i.e. $\Gamma\langle d_i, l_i\rangle$, constitutes the interval to be deleted. This is in accordance with the common way in which additions and deletions are made. Additions or deletions from lists are made one node at a time, while additions and deletions from trees are made by processing subtrees. The assumption is that by deleting an interior node in a tree, we want to delete all its successor nodes, too. There might be a case when after deleting an interior node in a tree the user might want to restructure the tree, but we chose for the sake of simplicity (and control) not to implement this overly powerful operation.

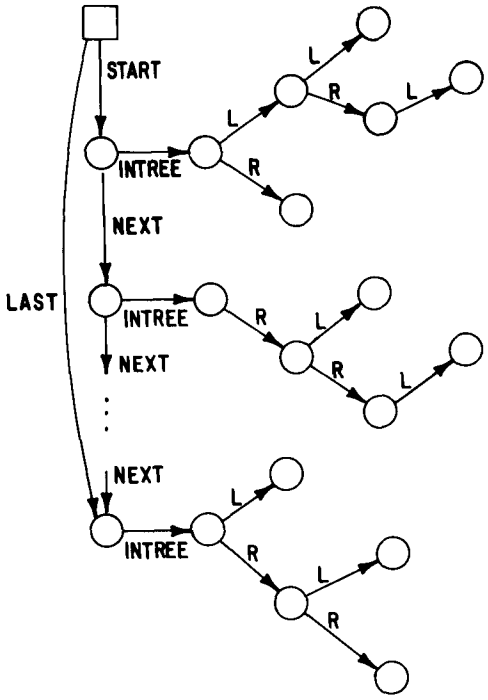
The second fundamental concept in this system is that in multistructures a higher level node is the owner of the entire lower level structure emanating from it. Consider the example of a one-way list with an end pointer (level 1) each of whose nodes is a binary tree (level 2) illustrated in Figure 3. Leaving out the definition of each node's data, we can describe the structure as:

```
DECLARE EXAMPLE
    ⋮  (description of data in entry node)
LEVEL(1) LIST(START) ONEWAY(NEXT) END(LAST)
    ⋮  (description of data in each node of the list)
LEVEL(2) TREE(INTREE) 2-BRANCH(L,R)
    ⋮  (description of data in each node of the tree)
```

The deletion of a node in the tree implies the dele-

[1] In general, $\Gamma^n\langle d_i, l_j\rangle = \Gamma^{n-1}\langle\Gamma\langle d_i, l_j\rangle, l_j\rangle$ and $\Gamma\langle\{d_1, d_2, \ldots d_k\}, l_j\rangle = \bigcup_{i=1}^{k}\Gamma\langle d_i, l_j\rangle$.

Fig. 3.



tion of the entire subtree associated with that node. The deletion of a node in the one-way list implies a deletion of the entire list emanating from it and a chaining through operation to remove the node from the one-way list. Notice that for simplicity we no longer name the nodes. For all the manipulations performed on the data structure, a "working" pointer is maintained pointing to the current node. By specifying the address of the current node and the branch label to travel along, the successor will be unique if it exists (since Γ is a partial bijective function).

The DSDML comprises four components: the Definition Statements, the Search Statements, the Manipulation Statements, and the Extended Manipulation Statements.

3.1 Definition Statements

Every structure definition begins with a DECLARE statement (production 1 in Appendix) followed by the name of the structure and a data description of the information to be kept at the entry node. For example, in the case of a file structure this information might include the date of creation, date of last alteration, a description of the information in the structure, or other descriptive information. Following the DECLARE statement, the system creates the structure by allocating space for its entry node. Recall that the entry node information is made available by merely invoking the name of the structure.

All structures must have at least one level, but could have several. In discussing a one-way list of trees of stacks, the one-way list is level one, the trees are at level two, and the stacks are at level three. It is possible to have more than one structure at each level. Consider a one-way list in which each node has a tree and a deque emanating from it. The one-way list is at level one, but both the trees and the deques are at level two. The levels must be described in ascending numerical order beginning with LEVEL (1) without skipping levels (productions 2 and 3). The nodes in a LEVEL (k) structure are the header nodes for structures in LEVEL ($k + 1$). When there is more than one structure at a level the sequence of level numbers mimics COBOL or PL/I structure definitions, for example, 1, 2, 3, 2, 3. Recall the definition of uniform insertion of two structures to understand why for a one-level structure the entry point is the only header node, while for a multi-structure, in addition to this, the nodes of a given level k serve as headers for the structures emanating from them.

It might be useful to distinguish here between two classes of objects which appear in the process of using data structures. These resemble very much the data and program objects which are part of any programming language [6]. Data objects can have a value, which may be changed in the course of the program (e.g. an integer variable, a pointer variable, a fixed character string). Program objects do not have values, but may be invoked to compute values for data objects (e.g. a procedure, a statement, an entire program). The dynamics of data structures presents us with two similar classes of objects: static and dynamic [3]. Static objects are in existence as long as the data structure is, while dynamic objects are created at run-time and have to be referenced through the static objects. Static nodes are headers of data structures (or entry points, in our terminology), and the branches emanating from them are also static objects. Dynamic nodes in a data structure are created by the different primitives operating on it (insert, delete, etc.) and thus may have a limited existence. Dynamic nodes can be referenced only through branches. To conclude, the branches emanating from dynamic nodes are also dynamic and represent the access path in the structure.

In particular, for the data structure definitions in the DSDML: the entry node (static) is made available by invoking the name of the structure; the pointer from the entry node to the first data node has a different label than the labels of the inner branches since it corresponds to a static branch. The data nodes and the pointers emanating from them are obviously dynamic objects.

The structure descriptions (production 4–9) provide information about the type of organization and branching structure.

3.1.1 Lists

Lists (production 4) have a specified pointer from the entry node to the first data node. The branch labels for the data nodes in one-way lists or two-way lists are

Fig. 4.

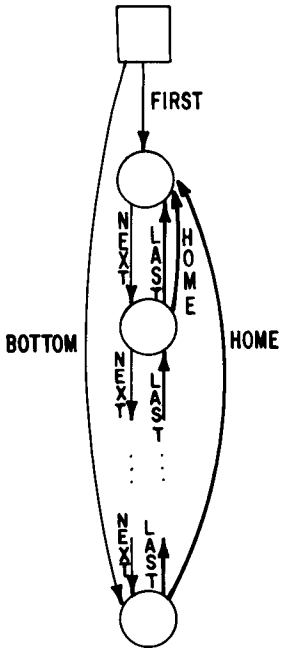
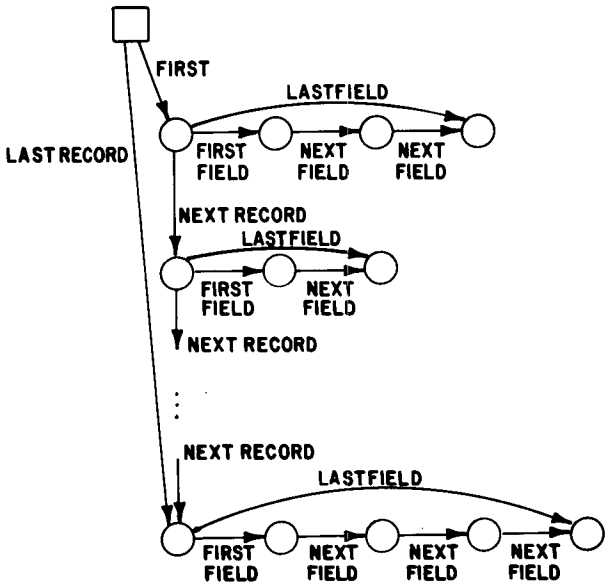


Fig. 5.



specified in productions 11 and 12, respectively. Reset branches which point to the first node are given labels in production 13. Finally, the header node for a list may contain a pointer to the last data node, the END pointer, as specified in production 15. The branch label names for searching, for reset, and for the end positions are chosen by the programmer; but the system checks for compatability.

As an example, consider the declaration of a two-way list with an end pointer and with reset pointers (see Figure 4):

```
DECLARE DUBLCHAIN
    ⋮ (data description of information at the entry node)
LEVEL(1) LIST(FIRST) TWOWAY(NEXT,LAST)
         RESET(HOME) END(BOTTOM)
    ⋮ (data description of information at each node)
```

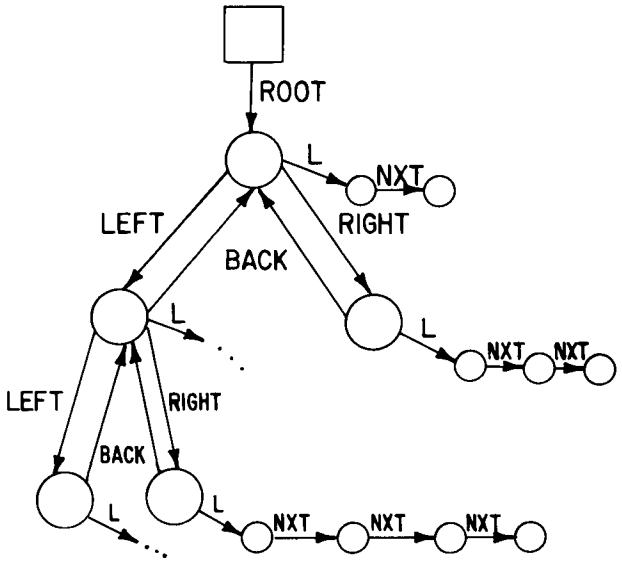
A file structure consisting of a variable number of records, each having a variable number of fields can be described as a one-way list of one-way lists with end pointers at both levels (see Figure 5):

```
DECLARE FILE
    ⋮
LEVEL(1) LIST(FIRST) ONEWAY(NEXTRECORD)
         END(LASTRECORD)
    ⋮
LEVEL(2) LIST(FIRSTFIELD) ONEWAY(NEXTFIELD)
         END(LASTFIELD)
    ⋮
```

3.1.2 Trees

The tree description has to start with a branch label for the pointer emanating from the entry node, followed by the branching information for each node of the tree. This includes specifying the number k of maximum possible branches from every node and their cor-

Fig. 6.



responding unique names. The maximum number of branches is implementation dependent. The difference between a 1-BRANCH tree and a one-way list is in the size of the interval for a node. In a one-way list the interval for a node contains only that node. In 1-BRANCH tree the interval for a node consists of all the nodes which are its successors. The back option permits the inclusion of labeled branches which point to the immediate predecessor or parent of a node. Finally, the reset pointer permits the inclusion of a pointer from each node to the first data node.

The definition of a binary tree (with back option pointers) of one-way lists would be (see Figure 6):

```
DECLARE BINTREE
   ⋮
LEVEL(1) TREE(ROOT) 2-BRANCH(LEFT,RIGHT)
         BACK(BACK)
   ⋮
LEVEL(2) LIST(L) ONEWAY(NXT)
   ⋮
```

3.1.3 Other Linear Structures

Rings are similar to lists, differing only on one detail: the last node in the list points back to the first node, thus (willingly) creating a cycle. In this sense the ring is not purely a linear structure.

Queues are also similar to lists, the difference being in the permissible operations. The queue can be entered by either of two pointers available from the entry node. Additions and deletions to queues may be made only at the tail or head of the queue, respectively. The tail and head nodes of a queue and any lower level structure emanating from them are available for searching, but interior nodes are not automatically accessible. If the ability to search interior nodes is required, this must be explicitly indicated by using the searchable option, in which one or two directional searching may be described. A one-way search begins at the head and leads to the tail; for two-way searching the second branch label indicates the tail-to-head direction.

Stacks are lists in which additions and deletions may be made only at the top. A distinction is usually made between stacks and pushdown lists: in the former all nodes are searchable; in the latter only the topmost nodes are accessible. To permit implementation of this distinction, the searchable option is included. Although the reset and end options are permitted, there is normally little use for them.

Deques are queue-like structures in which additions and deletions may be made at either end.

3.2 Search Statements

Searching these structures can be described in simple terms. Moving from node to node is accomplished by setting a pointer to point at the current node looked upon. Any number of pointers may be declared by the user program to keep track of several nodes at once. (It is sometimes desirable to keep a pointer to the previously searched node, too.) This process might be visualized as adding temporary entry nodes to the structure. Pointers may be saved, copied or compared exactly as is now permitted in PL/I. The reserved word NULL is used as the value of the pointer variables where there is nothing to point to, such as at the end of a one-way list.

To begin searching a structure, the pointer assignment statement given in production 23 has to be used. Since searching begins at the entry node, we ENTER the given structure name and set the current-node pointer to the address of the entry node. To move from the entry node or from any other node, the GET statement (production 22) is used providing the node location and the branch label to indicate the direction in which we want to travel. To search through the file structure in Figure 5, which was discussed earlier, the structure must be entered, searched down the records, and then searched across the fields. The following program, written in "extended" PL/I seeks out a particular field in the FILE structure. MATCHRECORD and MATCHFIELD stand for blocks of statements, or procedures, which check if the node currently retrieved is in the right record or field respectively.

```
DECLARE GRIPHOS
   ⋮
LEVEL(1) LIST(FIRST) ONEWAY(NEXT RECORD)
         END(LAST RECORD)
   ⋮
LEVEL(2) LIST(FIRSTFIELD) ONEWAY(NEXTFIELD)
         END(LASTFIELD)
   ⋮
CURNODE = ENTER(GRIPHOS);
   ⋮
CURNODE = GET(CURNODE,FIRST);
DO WHILE(CURNODE¬ = NULL);
IF MATCHRECORD THEN
  DO;
  CURNODE = GET(CURNODE,FIRSTFIELD);
    DO WHILE (CURNODE¬ = NULL);
      ⋮
    IF MATCHFIELD THEN CALL FOUND;
    ELSE
    CURNODE = GET(CURNODE,NEXTFIELD);
    END;
  END;
CURNODE = GET(CURNODE,NEXTRECORD);
END;
```

3.3 Manipulation Statements

The restriction to linear structures and tree-like structures permits the use of a simple set of primitives. A standard interpretation can be given to these primitives which depends only on these two classes of data structures. Standard interpretation refers to the property that the primitives should always produce the same result, independent of the sequence of operations which have preceded it and independent of the current status of the structure. The notions of interval and multistructure enable the realization of these primitives in the DSDML. Recall that additions and deletions to lists are made one node at a time, while additions and deletions to trees are made by subtree manipulation. This is in accordance with their respective interval definitions.

With regard to multistructures, the addition or deletion of a node at level k will affect the whole structure from level k down.

3.3.1 Insert Operation

The insert operation is defined in production 24. The structure specified, is inserted along the given branch label, emanating from the node specified. Note that the insertion has to be uniform, meaning that the structure to be inserted must match the structure definition of the receiving structure; trees may be added to a one-way list of trees; queues may be added to a one-way list of queues; nodes may be added to lists. For trees the insert operation can be interpreted as an "append" operation. It is defined only on incomplete nodes. An incomplete node is a node for which the branches defined in the declaration statement are not all in use. Other implementations would allow for more than one result.

The run-time package would prevent the insertion of a stack into a tree of queues or the insertion of a tree into a one-way list. The second function of the run-time package is to prevent the insertion of nodes along end pointers, reset pointers, etc. These two basic checks help prevent the creation of invalid structures. Only the structures that have been defined can be produced.

3.3.2 Delete Operation

When dealing with the delete operation two critical situations have to be avoided. First, care must be taken to ensure that a portion of the structure does not become unreachable (the so-called "hanging" structure). This danger is avoided in DSDML by the interval organization strategy. When a node is deleted, the entire interval is deleted, and a rechaining operation may take place. In one-way lists the deleted node is "chained through," and the pointers are reset to preserve the list properly. In trees an entire subtree is deleted and the branch entering the deleted subtree is set to NULL.

A second danger is that the deletion of a node will leave certain pointers with incorrect values. By explicitly defining the labels of every branch in the data structure, it is possible to properly reassign the pointers associated with the structure after such a manipulation has been performed. For example, the deletion of the top node in a stack must be accompanied by a resetting of the top pointer; and the deletion of the last node of a one-way list must be followed by the setting of a NULL pointer in the next to last node.

Two formal delete operations are defined in production 25 and 26. The delete operation as defined in production 25 is used for linear structures. Only the node to be deleted need be specified, and the meaning of the operation is precisely defined. The chain is preserved, the end pointer is reset, the reset pointer is deleted, etc. In two-way lists, the proper branches are altered.

The delete operation as defined in production 26 is used for trees. The node and branch label specified point to the root node of the subtree to be deleted. After the deletion, the branch label specified is set to NULL.

The space freed in performing a deletion may be returned to the free space list by a garbage collection technique. (Note that a more complicated situation appears when dealing with rings, which gives some of the flavor of the problem encountered with cyclic structures. The chain pointers are preserved the same way as in lists, but upon deleting the first data node (i.e. the node pointed by the entry node) additional arguments have to be taken into consideration for resetting the entry pointer.)

3.3.3 Detach Operation

The detach operation defined in production 27 and 28 is similar to the detete operation except that the nodes removed are not destroyed. They become an independent structure whose name is that specified in the statement. A detached structure may be inserted back into another structure at a later time in the processing.

3.4 Extended Operations

3.4.1 Copy Operation. A structure or substructure may be copied and assigned a new structure name by the use of the statement defined in production 29. The copy statement is similar to the detach statement except that the original structure remains unchanged.

3.4.2 Interchange Operation. This operation (production 30 for linear structures, production 31 for trees) permits two nodes which have the same definition and are at the same level, to be interchanged. It is equivalent to detaching both nodes and reinserting them in the alternate positions. Thus, two nodes in a list or a ring could be interchanged by the issuance of one statement, instead of four statements. Interchanging a tree and a one-way list would not be permitted, but interchanging two subtrees is permitted.

The statement

```
INTERCHANGE(node1, left, node2, left)
```

is equivalent to the following sequence of statements:

```
DETACH(node1, left, name1)
DETACH(node2, left, name2)
INSERT(node1, left, name2)
INSERT(node2, left, name1)
```

to perform the interchange of two subtrees. Notice that node1 and node2 are not moved, only their left subtrees are interchanged.

3.4.3 Replace Operation. The replace operation (production 32 for linear structures, production 33 for trees) permits a node or a subtree to replace another node or subtree if they have the same definition and are at the same level. It is equivalent to deleting one node, detaching a second node, and inserting the second node in the position occupied by the first node. Replacing a tree by a list would not be permitted, but replacing a subtree by another subtree would be permitted. The statement

```
REPLACE(node1, left, node2, right)
```

is equivalent to the following sequence of statements:

```
DELETE(node1, left)
DETACH(node2, right, name)
INSERT(node1, left, name)
```

3.4.4 Insertion of a List. It is often necessary to insert an entire list in another list. This can be accomplished by detaching and inserting one node at a time, but this can be tedious and unpleasant for programmers. To simplify the operation for the user, the INSERTLIST operation (production 34) is defined. This operation permits the insertion of an entire list into another list if their definitions are compatible.

Note that all these extended operations do not give additional computational power, but are simplifying operations which have higher conceptual meaning to programmers.

4. Estimation of the DSDML and Comparison with Other Systems

Most commercial systems which provide some kind of data structure description and manipulation facility limit themselves to one or two structure types. For example, the Integrated Data Store [5] system mainly restricts the programmer to the use of rings; and although networks can be obtained by interconnecting different rings, storage is not efficiently used and the underlying structure is hidden behind the superimposed structure supplied by the system. It appears that the problem is created by the fact that their "world-view" is essentially based on records with fixed format, fixed fields (a specified number of fields have to be in a record). This consideration limits the access mechanisms to operate on specialized structures.

The major design effort by the CODASYL Data Base Task Group [1] to develop a Data Description Language and a Data Manipulation Language for a generalized Data Base Management System was very much concerned with selecting a useful set of data manipulation primitives. They have solved some of the problems posed by earlier systems, but the complexity of the system makes it (sometimes) difficult to follow the effect of the operations. The situation is that the same operations might produce different effects depending on a detailed set of declarations made in a complicated Data Definition Language. For example, the STORE operation creates a node and possibly connects it to one or more structures, depending on how the structure has been defined. The complexity and danger in deleting nodes have led the designers to define no less than four DELETE operations, each of whose functions vary depending on which node is referenced and on how the structure is defined. As a result, the user who implements a complicated structure in this system would have to be extremely careful when invoking the operations.

A different approach to describe the various structures and operations may be found in attempts to define a generalized graph programming language [8, 9]. Each of these systems gives the necessary primitives to manipulate a graph in an arbitrary manner. Complex structures such as acyclic graphs and networks can be created, but no guarantee is given to prevent the user from obtaining an illegal logical structure. Note that the structures are built in terms of primitives which resemble the level of assembly language operations.

The attitude taken for designing the DSDML was to choose and implement higher level primitives with just the right amount of power.[2] A host programming language like PL/I provides list processing facilities, allowing the addition or deletion of branches and the creation or destruction of nodes by the use of pointer variables and the ALLOCATE or FREE operations. However, these primitives are too powerful, since they permit programmers to create structures which are not well formed. Unrestricted use of the built-in ADDR primitive can be made by assigning the returned value of ADDR to a pointer variable. Since the argument of the ADDR function can represent any physical address, a pointer can be defined to point to any object in the program.

In choosing the primitives for the DSDML, we did not want restrictive operations which are meaningful in only a very limited environment, nor did we desire a set of primitives which are so powerful that the user can unwittingly destroy a structure. The development of the DSDML depended very much on the restriction to basically two types of structures: linear and tree-like, and combinations of these. The explicit recognition of these distinct classes of data structures enabled us to assign a unique meaning to the operations, which depends only on the class of the structure being operated on. The results of the operations are always the same, independent of the sequence of operations which might have proceded it and independent of the current status of the structure.

Some questions were raised in the course of this work which we plan to investigate further. First of all a rigid formalism utilizing graph theory and formal semantics should be developed to indeed prove that any sequence of allowable operations in DSDML yields the well-formed structure that is defined in the declaration statement. Secondly, we plan to enlarge the category of structures dealt with by allowing multiple entry nodes and structures associated with indexes to include, for example, indexed sequential files.

Acknowledgment. We would like to express our sincere appreciation for the many useful and productive discussions with Professor Jack Heller, State University of New York at Stony Brook, and Dr. Arnold Rosenberg, IBM Thomas J. Watson Research Laboratories.

[2] The DSDML could be implemented in an extensible language, like Algol 68.

Appendix. Productions for a Restricted Data Structure Definition and Manipulation Facility

```
 1) <definition statement>::=DECLARE<structure name><data
                          description><level description>

 2) <level description>::=LEVEL(<k>)<structure
          description><data description><level description>

 3)     | LEVEL(<k>)<structure description><data description>

 4) <structure description>::=LIST(<branch label>)<1 or 2
                            way><reset option><end option>

 5)     | TREE(<branch label>)<k>-BRANCH(<branch
                     label list>)<back option><reset option>

 6)     | RING(<branch label>)<1 or 2 way><reset option><end
                                                    option>

 7)     | QUEUE(<head>,<tail>)<searchable option>

 8)     | STACK(<branch label>)<searchable option><end option>

 9)     | DEQUE(<head>,<tail>)<searchable option>

10) <searchable option>::=SEARCHABLE<1 or 2 way><reset option>
      | λ

11) <1 or 2 way>::=ONEWAY(<branch label>)

12)     | TWOWAY(<branch label>,<branch label>)

13) <reset option>::=RESET(<branch label>)  | λ

14) <back option>::=BACK(<branch label>)  | λ

15) <end option>::=END(<branch label>)  | λ

16) <branch label>::=IDENTIFIER

17) <branch label list>::=<branch label>,<branch label list>
      | <branch label>

18) <structure name>::=IDENTIFIER

19) <k>::=NUMBER

20) <tail>::=<branch label>

21) <head>::=<branch label>

22) <node location
    assignment statement>::=<node>=GET(<node>,<branch label>)

23)     | <node>=ENTER(<structure name>)

24) <operation>::=INSERT(<node>,<branch label>,
                              <structure name>)

25)     | DELETE(<node>)

26)     | DELETE(<node>,<branch label>)

27)     | DETACH(<node>,<structure name>)

28)     | DETACH(<node>,<branch label>,<structure name>)

29) <extended operations>::=COPY(<node>,<branch label>,
                                    <structure name>)

30)     | INTERCHANGE(<node>,<node>)

31)     | INTERCHANGE(<node>,<branch label>,<node>,
                                  <branch label>)

32)     | REPLACE(<node>,<node>)

33)     | REPLACE(<node>,<branch label>,<node>,<branch label>)

34)     | INSERTLIST(<node>,<structure node>)

35) <data description>::=description in the host language

36) <node>::=IDENTIFIER
```

The productions are expressed in Backus-Naur Form (BNF) with the following constants:

NUMBER is a integer value

IDENTIFIER is a character string

λ is the null or empty production

Received April 1973; revised January 1974

References

1. CODASYL—Data Base Task Group Report. Available from ACM, New York, (Apr. 1971)
2. Dijkstra, E.W. Go to statement considered harmful. *Comm. ACM 11*, 3 (Mar. 1968), 147–148.
3. Earley, J. Toward an understanding of data structures. *Comm. ACM 10*, 10 (Oct. 1971), 617–627.
4. Harary, F. *Graph Theory*. Addison-Wesley, Reading, Mass., 1969.
5. Integrated Data Store. Honeywell Information Systems, Inc., Wellesly, Mass., 1971.
6. Kieburtz, R. Steps towards verifiable programs. Tech. Rep 12, Dep. Comput. Sci., SUNY at Stony Brook, N.Y.
7. Mills, H. Top down programming in large systems. From Debugging Techniques in Large Systems. Gourant Computer Science Symposium, pp. 41–53.
8. Pratt, T., and Friedman, D. A language extension for graphs—processing and its formal semantics. *Comm. ACM 14*, 7 (July 1971), 460–467.
9. Crespi-Reghizzi, S., and Morpugo, R. A language for treating graphs. *Comm. ACM 13*, 5 (May 1970), 319–323.
10. Rosenberg, A. Data graphs and addressing schemes. *J. Comput. and Syst. Sci. 5* (1971), 193–223.
11. Rosenberg, A. Exploiting addressability in data graphs. Rep. RC-3618 IBM, T.J. Watson Res. Cent., 1971.
12. Rosenberg, A. Symmetries in data graphs. *SIAM J. Comput. 1*, (1972), 40–65.
13. Senko, M.E., Altman, E.B., Astrahan, M.M., and Fehder, P.L. Data structures and accessing in data-base systems. *IBM Syst. J. 12*, 1 (1973).
14. Shneiderman, B. Data structures: Description, manipulation and evaluation. Ph.D. Th., Dept. of Comput. Sci., SUNY at Stony Brook, N.Y., 1973.
15. Wirth, N.K. Program development by stepwise refinement. *Comm. ACM 14*, 4 (Apr. 1971), 221–227.