The Structure of "Data Structures"

Wilfred J. Hansen
University of Pittsburgh

Abstract

A data structure is defined to be a 4-tuple <D, F, S, A>. D and F are Domain and Function definitions which define the externally observable behavior; S and A are a Storage Structure and Algorithms which implement the functions. It is shown that this definition helps organize the field of data structures for presentation to students. In particular, the fact that the Storage Structure is a (lower level) data structure leads to a hierarchy of data structures. Following this hierarchy, a suitable order for presentation of the material is bits, words, arrays and records, lists, trees, and search tables.

Students deserve structured presentations. In a course on data structures this is especially important because otherwise the subject can seem like a series of unrelated tricks. Moreover, the material is on structure and will be used to build programs with structure, so the student should be given as many examples of good structure as possible.

There are a number of questions about how the material on data structures ought to be organized. Is the order arrays, stacks, records, lists appropriate with each subject being motivated by its predecessor? Or should stacks be last so their multiple implementations are derived from the alternate implementations of lists? Where should list node representations be described since one alternative uses lists themselves? This paper answers these questions by referring to the data structure definition presented in Section 1. This definition distinguishes between the externally observable behavior of the data structure and its implementation. Since the latter must be in terms of lower level data structures, the definition induces on data structures the hierarchy discussed in Section 2. Finally, Section 3 describes various topics that are omitted by this organization and why these include some of the more formal appraoches.

1. Definition of a Data Structure

A data structure DS is defined to be a 4-tuple as in [Rein82a],

<D, F, S, A>: (1)

D a set of zero or more domains defined by the data structure.

F a set of definitions of functions. These provide operations on values in D and domains defined by other data structures.

S a storage structure. Definitions of variables and relationships between them.

A a set of algorithms, one for each element of F. These implement the functions F by manipulating the storage structure S.

As part of the storage structure definition, representations must be specified for values in each of the domains of D. The algorithms A are permitted to access the components of such values, but other routines are not. Nor may external routines access any variable described within the storage structure S. To illustrate this definition, three data structure descriptions follow.

An array is a complex data structure, not least because the elements may be of any type. For simplicity and to illustrate that a storage structure can be very low level, assume that a program needs a single array of twenty real values. The domain set D is empty, but the data structure uses domains real and "subscript" as defined by other data structures. The only requirement for the subscript domain is that it be possible to compare subscripts to integers.

Although most programming languages provide special syntax for arrays, the definition is better illustrated with ordinary function syntax:

```
F = {store, retrieve}

store (real, subscript)
    { associate the real value with
      "subscript" }
retrieve (subscript) returns real
    { return the value most recently
      associated with "subscript" }
```

One possible storage structure for these functions is an array. (It would be accessible only to the algorithms for store and retrieve.) To illustrate that alternate implementations are possible, suppose we choose a storage structure of twenty real variables:

```
A1,A2,A3,A4,A5, ... A20 : real
    { locations for array. Relationship:
    Ai will contain the value associated
    with subscript value i. }
```

With this storage structure, the algorithms A for F are somewhat tedious:

```
procedure store (value, subscript)
        if subscript=1 then A1 ← value
    else if subscript=2 then A2 ← value
        . . .
    else if subscript=20 then A20 ← value
    else error

procedure retrieve (subscript) returns value
    if subscript=1 then return A1
    if subscript=2 then return A2
        . . .
    if subscript=20 then return A20
    error
```

where error performs some appropriate action.

Many extensions to this data structure are possible. For example, to permit more than one array, the domain set could include a domain:

```
D = <array20real>
```

Each function would be redefined to include an array20real parameter. Any place where a program wanted an array, it would declare a variable to be of type array20real. This variable would then be used as the array parameter to the functions. Presumably, each array20real value would be a composite value (or **record**) containing the twenty real variables.

As a second example, consider a checkbook_balance data structure. This will illustrate that a data structure can be at a high enough level to provide services directly to a user. For simplicity the example will retain the account balance but will not record individual deposits and checks. The domain set D is empty, but check_book will utilize a dollar_amount data structure which provides at least addition and subtraction of dollar_amount values.

The only operations provided are these:

```
F = (open_account, deposit, check_written,
        report_balance)
open_account
    { Initialize the account.}
deposit (dollar_amount)
    { Update to reflect a deposit. }
check_written (dollar_amount)
    { Update to reflect writing a check }
report_balance returns dollar_amount
    { Return current balance in account }
```

The storage structure is only a single dollar_amount, the current balance in the account. Each algorithm performs the appropriate action on this balance. To record the amounts of the twenty most recent checks, this program could be extended with the array data structure above.

For an example at an intermediate level, consider a table. Each entry in it will have a set of attributes, one of which is a key which distinguishes this entry from all others. The domain set is

```
D = (element_location)
```

and other domains employed are key_value, attribute_1_value, attribute_2_value, ... attribute_N_value. Each value in the element_location domain will indicate the location of the table entry for some given key. This prevents having to find the location each time any attribute is accessed. (A data structure without this domain is possible, but would require a lookup operation for every attribute access.)

It is sufficient that the functions be

```
F = (initialize_table, enter_element,
    find_element, attribute_1, attribute_2,
    . . . attribute_N)

initialize_table
    { Sets table to have no entries. Must
    be called before any other table
    routines. }

enter_element (key_value) returns
element_location
    { Creates an entry for the given key_
    value and returns an indication of
    where that entry is. }

find_element (key_value) returns
element_location
    { Finds key_value in table and returns
    the same value as returned by enter_
    element when key was first entered. }

attribute_i (element_location) returns
attribute_i_variable
    { Returns a variable of domain attri-
    bute_i_value. This variable is one
    component of the table entry for the
    key which corresponds to the element_
    location value. }
```

A number of storage structures are suitable for implementation of these functions. If arrays are chosen, element_location values are integers, while if lists are chosen they are pointers. Since the element_location domain is defined by this data structure, only the functions above can apply operations other than assignment to such values. Thus the implementor has the option of changing the storage structure at any time without affecting any facet of the remainder of the program except its running time and storage utilization.

2. A Hierarchy of Data Structures

The material on data structures can be organized by beginning with the observation that the components of S are values in domains defined by other data structures. If DS_1 is $\{D_1, F_1, S_1, A_1\}$ and S_1 uses domain D_2 from DS_2, then the key fact is that operations on S_1 in A_1 are written using functions F_2 provided by DS_2. This nesting generates a lattice with the relationship that one data structure is part of the implementation of another. One such lattice is shown in Figure 1.

At the bottom of the lattice, the bit is the primitive storage structure and serves to implement the bit data structure. This in turn serves to implement words. As shown in parentheses, a word can have a number of alternate interpretations, and any of these can be chosen when a word appears in the storage structure for a higher data structure. As examples of the rest of the chart, note that an array is implemented with consecutive words and also with an integer for subscript values. Note, too, that a data base can be either a table and a B-tree, or it can be some other form of table. This portion of the lattice is necessarily incomplete because there are many alternatives.

Figure 1 can serve as the basis for organizing the material on data structures. The presentation can proceed from the bottom of the lattice to the top:

bits

words

arrays and records

lists

trees

tables

data bases.

Three examples will illustrate how the lattice can resolve problems of ordering the material.

Example 1. Is a list always linked?
The term "list" has two common meanings. As a data structure, a list supports operations like finding the successor of an element and creating a list from a value and a (shorter) list. As a storage structure, "list" usually means a linked list, where each element includes the address of the next. By distinguishing data structures from storage structures, definition (1) clarifies this dual meaning. It is then reasonable to point out that arrays also provide a storage structure for lists. Of course this presentation should also point out that array representations may lead to less efficient algorithms.

Example 2. How is a list element implemented?
A list element is a data structure in its own right. The principle function is access to a field; often the only fields are the value in the element and the pointer to the next element. The most natural storage structure is a **record**, which directly implements field access. But there are a number of alternate storage structures, including "parallel arrays". This scheme is often used in older languages which do not support **record** values: One array is declared for each field of a list element; for each subscript value, there is a list element consisting of the array elements having that subscript. When list elements have many fields, it is possible to implement an element as a "mini-list" of "mini-elements", each of which has one field of the element. To permit having only one section on implementation of list elements, it is preferable to defer this discussion until after the introduction of lists.

Example 3. The storage structure for a stack is a list.
A stack provides the functions of insertion and deletion at only one end of a list. The two possible storage structures for lists - arrays and linked lists - then lead to two implementations for stacks. Thus stacks should be covered after lists, rather than using them as an example to motivate lists. With this order, the student learns about lists before dealing with the special restrictions of stacks.

It is valuable to note that the hierarchy of data structures is closely related to structured programming. Several of the techniques recommended for structured programming are enforced by careful attention to the data structure organization of a program:

> "Levels of abstract machines" [Dahl72a].

> "Top-down refinement" [Wirt71a].

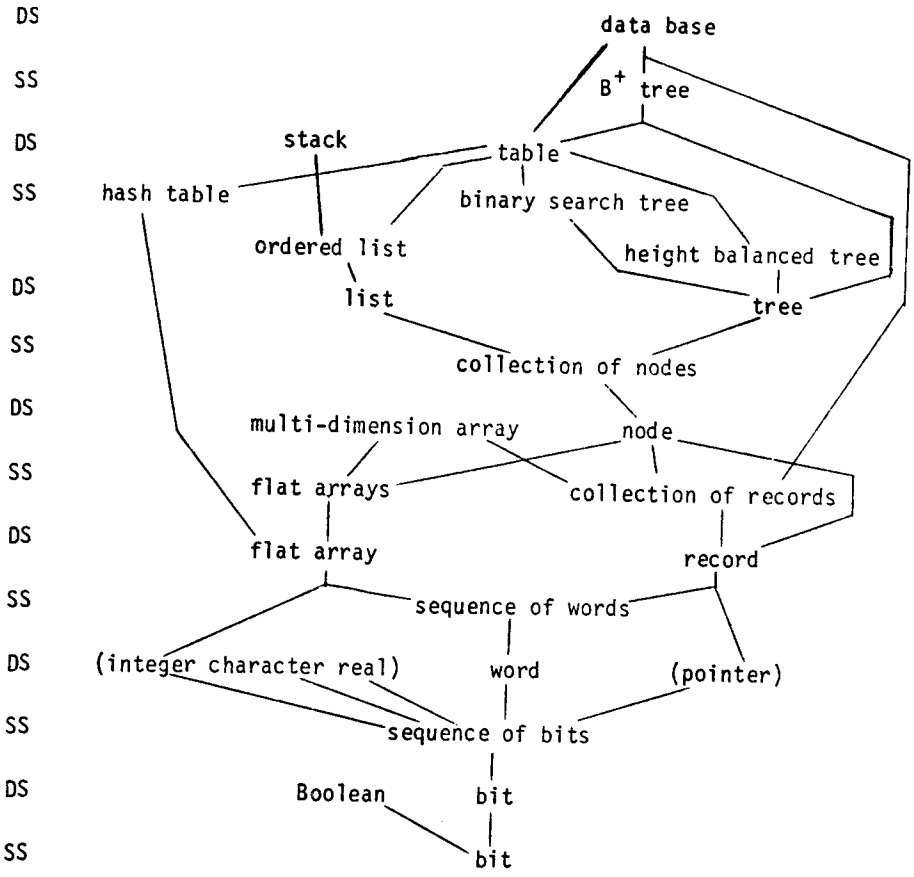

Figure 1 Lattice of data structures with the relation "implements". A forked line indicates that both lower structures are used. Multiple lines indicate that either may be used. The parentheses suggest that the alternate interpretations of words may be used in higher level structures.

In Dijkstra's conception, each "abstract machine" provides operations for machines at higher levels. This is equivalent to lower data structures providing operations to higher data structures.

Wirth's approach calls for first writing a program in a high level pseudo-language. Then the statements are refined to be closer to an implemented programming language. During refinement, storage structures are introduced as needed and operations are defined to manipulate the structure. The higher level routines utilize these operations to manipulate the storage, so the storage and operations consitute a data structure within the meaning of definition (1).

> "Reduced coupling" [Myer78a].

> "Information hiding" [Parn72a].

In these versions of structured programming, the emphasis is on the minimization of impact from one routine to another. Although the two authors have different approachs, the data structures of definition (1) are a suitable implementation of either. The variables of a storage structure are "hidden from" (inaccessible to) all routines outisde the data structure. Thus it is not possible to generate the form of "coupling" that occurs when one routine accesses or modifies variables intended solely for another routine.

3. What has been omitted?

In any description it is necessary to omit detail to clarify the overall structure, and certainly many topics have been omitted above. However, most of these same omissions should be made in any first course on data structures. Inclusion of too much material is more likely to confuse than to educate. There are two major areas of omission.

The first major area is that of formal specification. At the least, definition (1) could have been more formal with separation of the external and internal descriptions. That is, by letting DS be <E,I>, where E=<D,F> and I=<S,A>. This separation would have the conceptual advantage that the DS could be changed by replacing the Implementation I without replacing the External behavior description E. The original definition requires the formally more complex task of replacing the entire data structure with another having the same D and F. In a formal system, the latter task requires a notion of equivalence of descriptions of D and F. However, in practice it is not difficult to have an intuitive notion that two data structures can have the same external appearance. (Even though implementations are seldom really identical.) At heart, <D,F,S,A> is simpler just because it has one less level of notation and two fewer symbols.

No matter how they are introduced, however, formal specifications have a number of drawbacks at present. The major candidates for formal specification of the Domains and Functions are axiomatic and algebraic techniques [Scot76,Wulf76, Gutt78,Gogu77]. Both methods have many problems; see, for example, the discussion in [Cart80a]. Not least among these problems is that the theories are not well developed; only a few data structures have formal specifications and there is too little time in a typical course to teach undergraduates enough to understand such specifications by writing one themselves.

For specification of data structure implementation, the situation is only slightly better. There are of course programming languages which provide a somewhat formal specification of the algorithms in A. One could even use a language with formal semantics [say, PASCAL Hoar73a]. However, there is very little literature on specification of storage structures. The axiomatic and algebraic approaches have their appeal precisely because they specify as little as possible about the storage structure. To describe an implementation, however, it is necessary to describe program variables and to specify relationships between them. For example, if an array is chosen implement the table data structure above, it is desirable to specify relationships between element_location values and entries in the table. For one, every valid element_location value must be the subscript of an entry in the table that has been given a value with enter_key. Satisfactory descriptions of implementations ought to be constructive and with sufficient connection to the program that a compiler can verify that the relationships are maintained when variables are modified.

It is an interesting research question to provide a formal system for specifying how data structures are put together as storage structures. For example, a one dimensional array is concatenated words and a list is linked elements. Concatenation and linkage are two primitive operations that should be provided by such a formal system. Other approaches to this problem are referenced in [Rose81a]. One set of formal conditions for acceptable hierarchical nesting of specifications is given in [Wirs80a].

My primary rationale for omission of formal specifications is not noted above, but lies instead in the philosophical position that students should understand material before they are given a formal description of it. Understanding of a list will not come from axioms; it can only come from using lists to solve problems. This philosophy has been congently stated for grade school mathematics by Prof. M. Kline of New York University [Klin73a].

The second major area of omissions is that of programming language features for invocation of data structures. The two most common techniques are explicit invocation and identifier declaration. Both are available in Ada [DOD80a] and both are possible with definition (1). In explicit invocation, the data structure is made part of the environment for execution of a program by writing its declarations within the program text or by invoking them via something like Ada's **with** clause. Thereafter the functions of the data structure may be invoked. In the identifier declaration approach, a variable in the user program is declared to have as its type one of the domains in D. Then the operations of the data structure can be used on this variable. In principle, with data structure definition (1), a function may be called in any context where the parameters are of the correct domains. Thus if a data structure defines a function with only integers as parameters, that function may be called with any integer variables. It is preferable, however, that the data structure define its own domains so its values cannot be examined and modified by unauthorized routines.

When a data structure has internal variables like a stack pointer or the balance in a checkbook, these variables should often be initialized prior to execution of any of the functions of the data structure. Most programming languages that provide for data-structure-like goupings also provide mechanisms for initialization when a data structure is first invoked. To avoid describing such facilities, the development above simply expects that if initialization is to be done it will be done by some function that is called before all others.

Another omission is that of "generic" or "polymorphic" data structures. One example of such is an array; there should exist instances of arrays of integers, arrays of reals, and arrays of virtually any other domain. In Ada, such a data structure is declared as a form of variable declaration with a compile time parameter that specifies what type to use within the package. For definition (1), multiple types of arrays conceptually require multiple data structures.

A final problem is that of efficiency. If a procedure call is required for every function reference, the user program may be adversely affected. Consider the report_balance function which merely needs to return the value in a variable. This and other functions can be much more efficiently invoked if their internal code replaces the call in-line. Unfortunately, despite years of advocacy of this approach, little progress has been reported on compilers that actually perform this optimization. Nor are language facilities common for specifying that a routine is to be in-line rather than called. In a data structures discussion, the best that can be done is to suggest that the programmer manually insert the in-line expansion of the call where desired.

Omissions in the above two major areas are important because they reduce the topic of data structures to manageable proportions. In many of the subareas there is no agreement as to the best solution, so alternative and partial solutions are best discussed in a course directed to formal languages and abstractions. The intent of the presentation proposed above is that the student should first have a reasonably "concrete" notion of data structures before going on to a formal specification.

References

Cart80 Cartwright, R., "A Constructive Approach to Axiomatic Data Type Definitions," Proc. LISP Conference, pp.46-55 (1980).

DOD80 DOD,, Reference Manual for the Ada Programming Language, United States Department of Defense, Washington, DC (Jul 1980).

Dahl72 Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, Academic Press, New York (1972).

Gogu77 Goguen, J., J. Thatcher, E. Wagner, and J. Wright, "Initial Algebra Semantics and Continuous Algebras," JACM Vol. **24**(1), pp.68-95 (1977).

Gutt78 Guttag, J. V. and J. J. Horning, "The Algebraic Specification of Abstract Data Types," Acta Informatica Vol. **10**, pp.27-52 (1978).

Hoar73 Hoare, C. A. R. and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica Vol. **2**, pp.335-355 (1973).

Klin73 Kline, M., Why Johnny Can't Add: The Failure of the New Math, St. Martins Press, New York (1973).

Myer78 Myers, G., Composite Structure Design, Van Nostrand, Reinhold, New York (1978).

Parn72 Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules," CACM Vol. **15**(12) (December 1972).

Rein82 Reingold, E. M. and W. J. Hansen, Data Structures, Winthrop, Cambridge, 1982.

Rose81 Rosenberg, A. L., "On Uniformly Inserting One Data Structure into Another," Comm ACM Vol. **24**(2), pp.88-90 (Feb 1981).

Scot76 Scott, D., "Data Types as Lattices," SIAM J. Comput. Vol. **5**, pp.522-586 (1976).

Wirs80 Wirsing, M., P. Pepper, H. Partsch, W. Dosch, and M. Broy, "On Hierarchies of Abstract Data Types," TUM-I8007, Technical University of Munich (1980).

Wirt71 Wirth, N., "Program Development by Stepwise Refinement," Comm. ACM Vol. **14**(4), pp.221-227 (Apr 1971).

Wulf76 Wulf, W. A., Ralph London, and Mary Shaw, "An Introduction to the Construction and Verification of Alphard Programs," IEEE Transactions on Software Engineering Vol. **SE-2**(4), pp.253-265 (1976).