Algorithms and Data Structures

John Bruno Editor

The Effect of Data Structures on the Logical Complexity of Programs

FAROKH B. BASTANI and S. SITHARAMA IYENGAR

ABSTRACT: The logical complexity of a program is a measure of the effort required to understand it. We hypothesize that the logical complexity of a program increases with the increase in the opaqueness of the relationship between the physical data structures used in the program and their corresponding abstract data types. The results of an experiment conducted to investigate this hypothesis are reported. Documentation techniques for making programs easier to understand using complex data structures are discussed. Data structure diagrams, data structure invariants, stepwise transformation of data structures, and formal specification of the mapping between abstract and concrete data structures are illustrated using two nontrivial examples.

1. INTRODUCTION

Any program intended for use over a period of years must be implemented to be comprehensible for other programmers to maintain. This can be achieved by proper coding and documentation practices. To determine whether a program has been well designed and adequately documented, we must measure its "difficulty," that is, the amount of time required to understand it. Over the past decade several program complexity metrics [5] have been proposed. Those that are widely known are software science [7], control flow complexity metrics [1, 13], program knots [21], and complexities due to data flow dependencies [2, 8, 9]. Over the last five years a number of papers have been published on various aspects of program complexity. For overviews of related research on software complexity metrics see [5, 14].

We investigate the effect of data structures on program complexity. An experimental study of the effect of control flow and data structure documentation on program comprehensibility is discussed by Schneiderman [17]. He concludes that data structure information is more helpful than control flow information for understanding programs irrespective of whether the information is in textual or graphical format.

Section 2 discusses an experiment that we conducted to verify our hypothesis that data structures are not inherently complex once a certain level of knowledge is assumed, although the representation of advanced data structures (such as sets) using simpler data structures (such as arrays) can increase a program's complexity if the mapping between the two data structures is obscure. Since it is often essential to use primitive data structures for performance reasons, the relationship between the abstract and concrete versions of a data structure should be well documented. Toward this end, Section 3 discusses documentation techniques for data

This work was supported in part by the National Science Foundation under Grant MCS-83-01745.

A paper describing the experiment reported here was presented at the Symposium on Empirical Foundations of Information and Software Science, Atlanta, Ga., Oct. 1984.

structures that can aid in program clarification. These include both informal and formal approaches, such as data structure diagrams, stepwise transformation, invariant assertions, and mapping specifications. These techniques are illustrated by two detailed examples in Section 4. Section 5 summarizes the article and outlines some research directions.

2. EXPERIMENT

Several researchers have studied the effects of comments, different control structures, mnemonic names, textual and graphical documentation, etc., on program comprehension [4, 16–19]. After analyzing a number of examples, we conjectured that data structures are not inherently complex. That is, we cannot, for example, assert that a program that manipulates trees is inherently more complex than a program that uses arrays. However, if a program using trees is transformed into one using arrays, then the latter program can be more complex if the correspondence (i.e., mapping or relationship) between the tree and its array representation is obscure.

This hypothesis is important because it can be used to guide the extent to which documentation should emphasize data structure. Suitable documentation methods are discussed in Sections 3 and 4. In this section, we discuss an experiment conducted to verify this hypothesis.

Prior to conducting the experiment we selected a set of programs and an experimental method for determining their complexity. To isolate the effect of data structures on logical complexity, the programs had the same functions and algorithms but used different data structures. To estimate the complexity of the programs, we tested for comprehension by requiring a set of students to locate errors embedded in the programs. This is a reasonable criterion since it is usually the first step in program maintenance. An alternative approach is to pose some questions at the end of the comprehension period. However, this would test recollection ability which can be affected by the choice of names, degree of documentation, etc. [4]. Also, we did not require correction of errors since this would test problem solving capability in addition to comprehension.

For each program given to each student, the observed factors were the number of errors correctly located and the time taken to locate those errors. One possible metric which combines these factors is the rate of detection of errors. This can be criticized on the ground that some errors are detected more quickly than others. To overcome this obstacle we asked the students to try to find two errors in each program. Nevertheless, the analysis later in this section considers the two factors separately.

2.1 Subjects

Thirty students in two graduate courses at the University of Houston and nine students in a graduate course at Louisiana State University served as subjects in the experiment. The language used was Pascal. The experiment was conducted during regular class sessions in the middle of the semester (March 1984).

All of the subjects had knowledge of at least three programming languages (Pascal, Fortran or PL/1, other high-level or assembly languages), structured programming style, fundamental data structures, and a good background in mathematics.

2.2 Materials

The subjects were given two sets of programs coded in Pascal-like syntax, with some comments at the beginning of each program: Set 1 (part A) implements a stack PUSH operation presented in three procedures each using a different data structure (array, linked list, and combination of array and linked list). The second set of programs (part B) describes the sift procedure used in heap sort. Set 2a uses an array data structure and set 2b uses a binary tree data structure. Each program contained two or three artificially embedded errors. Additional details appear in [10].

2.3 Design

A repeated measures design with one factor was used in the first experiment (the stack PUSH operation) at both the test sites. Each subject received each of the three programs in turn. In the second experiment (heap sort), the subjects in Group I were given either the array or the binary sift procedure in a random order. Those in Group II received both the procedures in turn.

2.4 Procedure

The experiment was conducted during a regularly scheduled lecture period. In Group II, the instructor of the course had described the purpose of the testing in the previous lecture; in Group I this was done just prior to the experiment. The experiment was conducted in two parts. In part A (stack PUSH operation), the subjects were given the array program first. They were asked to find and mark up to two embedded errors in the program. They were also asked to note the time taken (in minutes) to detect the errors in the program. (The students were not required to correct those errors.) Then they were given the second program (employing linked list) and then the third program (combination of array and linked list) in turn and were asked to do the same thing. After completion of part A, the programs

were collected. In part B, the subjects were required to repeat the procedure for the heap sort procedure.

2.5 Scoring

Following the experiment, the exercises were scored to determine how many errors were correctly detected by each student. The criterion was: Would the location of the error identified by a student describe the error in the program? For the purpose of the study the errors were tallied 0, 1, or 2, the number of correct answers.

2.6 Results and Discussions

Tables I and II summarize the results of the experiment. For the number of errors successfully found in the stack PUSH operation, a significant difference in Group II was indicated by an $F(2, 16) = 3.65$, $p = 0.0495$, with further analysis by the Duncan multiple range test (alpha = 0.05) showing significantly more errors found for the linked list than for the array/linked list combination. More errors were found for the array algorithm also, but the difference did not reach significance. The differences for the number of errors in Group I were not significant ($F(2, 56) = 0.83$, $p = 0.4397$), but more errors were found for both the linked list algorithm and the array algorithm than for the combination algorithm.

A two-way analysis of variance was performed for both measures (errors and time) on the data for the stack PUSH operation, with a mean effect of data structure (array versus linked list versus array and linked list) and with "blocking" on individuals. Significant differences for the time taken to find the errors were observed for both groups ($F(2, 16) = 44.88$, $p < 0.0001$ and $F(2, 56) = 38.48$, $p < 0.0001$, respectively). Further analysis using the Duncan multiple range test (alpha = 0.05) determined that for Group I, the array and linked list combination was significantly more difficult than the linked list procedure. (See the means for time in Table I.) For Group II, the array and linked list combination was significantly more difficult than either the array or the linked list algorithm, but there was no significant difference between the latter two.

Some interesting results were obtained despite possible confounding by a learning effect due to the manner in which the procedures were assigned for the stack PUSH operation algorithm (part A). For the time to find errors, Group I's results show that the linked list algorithm was significantly easier than the array algorithm, but the learning effect confounds interpretation. That is, it would not be difficult to believe that the significant difference hinges on that alone. However, the combination algorithm (array and linked list) is the most difficult of all, and if learning plays a significant part here, it would be expected that it makes this the easiest algorithm. Therefore, it appears to be the most difficult to comprehend in any case. Similarly, Group II's results indicate that the combination algorithm is significantly more difficult than the other two algorithms.

For part B (heap sort experiment) a one-way analysis of variance shows no significant differences between algorithms for either the number of errors discovered ($F(1, 18) = 1.29$, $p = 0.2717$) or the time to find the errors ($F(1, 18) = 0.01$, $p = 0.9327$). The

TABLE I. Results of the Experiment for the Stack Push Operation

	Test site							
	Group I				Group II			
	Errors		Time		Errors		Time	
Data structure	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Array	1.310	0.891	3.828	1.311	1.889	0.333	3.664	1.144
Arrays and linked list	1.034	0.865	5.414	2.113	1.333	0.866	16.278	4.711
Linked list	1.103	0.939	2.103	0.860	2.000	0.000	5.500	1.936

TABLE II. Results of the Experiment for the Sift Procedure

	Test site							
	Group I				Group II			
	Errors		Time		Errors		Time	
Data structure	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Array	1.200	0.789	20.500	7.778	1.111	0.333	18.556	3.941
Binary tree sift	0.800	0.789	20.200	7.885	1.444	0.527	13.333	3.640

design in part B was subject to a learning effect in Group II, unlike the case in Group I where each subject received either program 2a or 2b but not both.

2.7 Observations

Although one should be cautious in drawing conclusions from the experiment in part A due to possible confounding, it seems safe to conclude that the combination of data structures was more difficult to comprehend than either data structure alone, at least for this particular program. The safety lies in the fact that the combination algorithm should have been the easiest if the confounding were the only real effect, when in actuality it was the most difficult. For the experiment in part B it apparently makes little difference which of the data structures is chosen for a heap sort algorithm.

How is it possible for programs containing advanced data structures to be easier to understand than programs performing identical functions but using simple data structures such as arrays? One likely explanation is that the understanding process in the latter case is similar to reverse compilation, namely, an attempt to synthesize the higher level (abstract) version from the lower level one. This comprehension task can be aided by presenting the data structures so as to facilitate such synthesis. This is discussed in the next section.

3. DATA STRUCTURE DOCUMENTATION METHOD

We will now outline four complementary methods for documenting data structures. These are illustrated using three different ways of implementing sets using arrays. The next section contains two examples.

Definition D1: A *data structure diagram* for a given data structure is a pictorial representation of the data structure which is constructed using a set of predefined notations, and conveys the primary relationships between the components of the data structure.

Such pictorial information greatly enhances the comprehension of data structures by human beings [17]. Brown and Sedgewick [3] propose the dynamic construction of data structure diagrams to provide snapshots of the state of the program for improving program understandability as well as for debugging purposes. Selecting the set of predefined notations requires extensive experimentation since data structure diagrams are intended wholly for conveying information to human beings. In this article, we use informal notations, as illustrated in Figure 1, for three different representations of sets using arrays. (See also [17].) We recommend that such notations evolve over several years before any standardization is attempted.

A data structure diagram is an informal representation of the state of the data structure. This can be formalized by giving the invariant assertion for the data structure [22] and it eliminates all ambiguities which may be present in the data structure diagram.

Definition D2: The *invariant assertion* for a data structure is a precise description of the legal states of the data structure. Comprehension of complex data structures can be facilitated by giving a sequence of intermediate representations. For example, the invariant assertions for the three array representation of sets are shown in Figure 1.

Definition D3: The *stepwise transformation* of a data structure starts with a high-level abstract version and, by successive transformation, ends with a lower level one. (See also [20].)

Stepwise transformation of data structures is similar to stepwise program decomposition. It informally shows how the data structure has evolved, making it easier to understand the detailed version. For example, a representation of stacks using an array/linked list combination can be understood by first understanding the representation of stacks using unbounded arrays and then the representation of unbounded arrays using a linked list of fixed-size arrays. This can be formalized by specifying the mapping. (See also [15].)

Definition D4: The *mapping* between two data structures is a formal specification of the equivalence of the information contained in the two data structures. For example, for Figure 1 we have:

```
(a) for all x: exists(s, x)
    ⇔there exists i: 1 ≤ i ≤ s.card
    such that s.table[i] = x;
(b) for all x: exists(s, x)
    ⇔there exists i: 1 ≤ i ≤ s.card
    such that
    (s.table[(i + 1)div 2]div 8^((i+1)mod2))
      mod 8 = x.
(c) for all x: exists(s, x)
    ⇔there exists i: 1 ≤ i ≤ s.card
    such that
    x = (s.table[1 + (3*i − 3) div 8
          div 2^((3*i−3)mod8)) mod 2
        + 2*[(s.table[1 + (3*i − 2)
        div 8]
          div 2^((3*i−2)mod8)) mod 2]
        + 4*[(s.table[1 + (3*i − 1)
        div 8]
          div 2^((3*i−1)mod8)) mod 2].
```

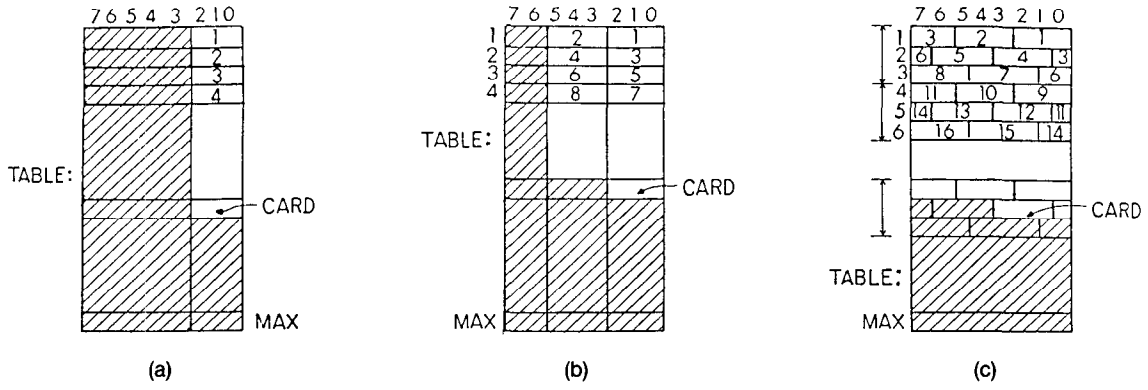

FIGURE 1. Data Structure Diagram for the Array Representation of Sets: (a) 0 ≤ s.card ≤ MAX; (b) 0 ≤ s.card ≤ 2* MAX; (c) 0 ≤ s.card ≤ (8* MAX) div 3

These four documentation techniques (D1, D2, D3, and D4) can *collectively* provide adequate information for data structures. Stepwise refinement (D3) shows the link between the final and initial data structures. It is formalized by giving the mapping (D4) between these data structures. Data structure diagrams (D1) informally show how the data structure is used, that is, they show the relationships between the various components. The data structure invariant (D2) formally shows the legal states of the data structure. Such combinations of informal and formal specifications are effective over the entire software life cycle [12].

4. ILLUSTRATIONS

In this section, we apply the documentation methods discussed in the previous section to two detailed examples.

Example: This example deals with a data structure for the symbol table module specified in [6]. The symbol table module considered here is intended for Pascal-like scope rules. A high-level data structure for this module uses a traversable stack of search table of key × attributes. Key corresponds to the search argument, say, identifier names. The data structure diagram for this high-level data structure is shown in Figure 2. The definition is:

```
var a: TraversableStack of SearchTable
       of key × attributes;
  NumScopes: integer;
initial a: = NewTraversableStack;
        NumScopes := 0;
```

Recall that in Pascal the definition of an identifier in a particular scope supersedes its definition (if any) in the enclosing scopes. The new definition holds in

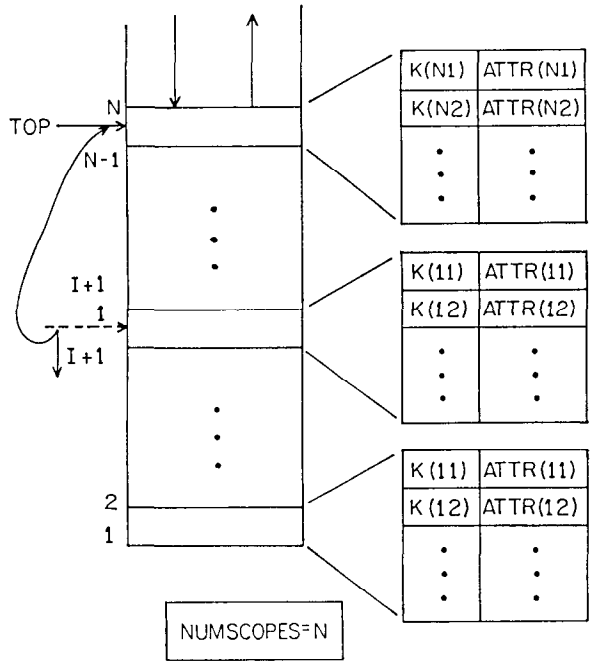

FIGURE 2. Data Structure Diagram for the Symbol Table Module

this scope and all enclosed scopes not redefining the identifier. Thus, an (almost) equivalent data structure is (see Figure 3):

```
var b: SearchTable of key × stack
       of attributes;
initial b := NewSearchTable;
```

It is not fully equivalent because for any given key we cannot determine the scope(s) in which it is declared. Thus, the `pop` and `IsInScope` functions [6] cannot be implemented. This can be rectified by as-

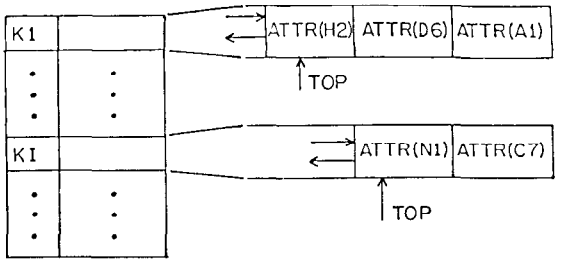

FIGURE 3. Data Structure Diagram for "SearchTable of Stack"

sociating a field ScopeNo with each attribute:

```
var c: SearchTable of key × stack of
       (ScopeNo: integer × attributes);
initial c: = NewSearchTable;
```

The IsInScope function can now be easily implemented. However, the pop function will require an examination of all the entries in the search table, a potentially inefficient operation. This can be remedied by using a separate stack to keep the keys in the order in which they are inserted in the search table (see Figure 4):

```
var d: SearchTable of key × stack of
       (ScopeNo: integer × attributes);
    e: stack of key;
initial d: = NewSearchTable;
        e: = NewStack;
```

Finally, we observe that stack e can be maintained by linking the elements in the stacks associated with each key in the search table in the correct order, that is, such that the top item has the highest ScopeNo and the bottom item has the lowest ScopeNo. This can be achieved by specifying (for every item) the key of the item which occurs immediately below it in stack e. The key of the item at the top is specified in a separate variable. The item at the bottom of stack e can specify any key since the search table will be empty when it is removed. Thus:

```
var f: SearchTable of key × stack
       of (PreviousKey: key ×
       ScopeNo: integer × attributes);
    top: key;
initial f: = NewSearchTable;
```

The final data structure uses an array g for the abstract data type SearchTable using hashing with double-hashing for collision resolution. Also, PreviousKey is replaced by PreviousIndex which contains the index of the item below this item. If PreviousIndex = n then there is no item below it, that is, it is the last item in stack e. The data structure diagram is shown in Figure 5.

```
initial NumDistinctKeys: = 0; top: = n;
  for all i: 0 ≤ i < n: (g[i].status: =
          empty; g[i].ptr: = nil);
```

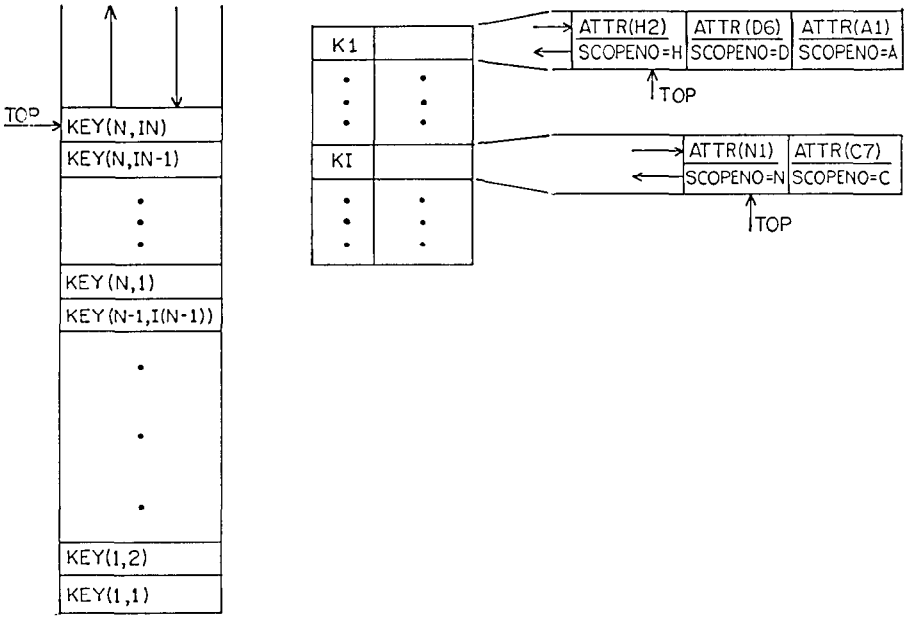

FIGURE 4. Efficient Data Structure for the Symbol Table Module

The mapping between this data structure and the original one (using a traversable stack) is as follows:

```
if top = n then assert NumScopes = 0
else assert NumScopes = g[top].ptr^.ScopeNo;
for i: = NumScopes downto 1 do
 {for all x such that x exists in top(a) do
    assert {there exists j: 0 ≤ j ≤ n − 1 such that
              g[j].k = x and g[j].ptr^.ScopeNo = i
            and g[j].ptr^.attr = attributes of x in top(a)}
  for j: = 0 to n − 1 do
    if g[j].ptr ≠ nil and then (g[j].ptr^.ScopeNo = i) do
      g[j].ptr: = g[j].ptr^.next
  pop(a)}
assert for all j: 0 ≤ j < n: g[j].ptr = nil;
```

The invariant assertions are:

```
1(a). for all i: g[i].status = occupied ⇔ g[i].ptr ≠ nil;
 (b). for all i: (g[i].status = marked) or (g[i].status = empty) ⇔ g[i].ptr = nil;
2(a). there exists i: g[i].status = occupied ⇔ top in [0 .. n − 1];
 (b). there does not exist i: g[i].status = occupied ⇔ top = n;
3.    NumDistinctKeys = j ⇔ there exists i1, i2, ..., ij such that
        for all m: 1 ≤ m ≤ j: g[im].status = occupied
        and for all i such that there does not exist m (1 ≤ m ≤ j: i = im):
          (g[i].status = marked) or (g[i].status = empty);
4.     for all i: g[i].status = occupied ⇒ there does not exist j: 0 ≤ j < n, j ≠ i:
         (g[j].status = occupied) and (g[j].k = g[i].k);
5.    for all i: g[i].status = occupied ⇒
        j: = h1(g[i].k); c: = h2(g[i].k);
        while j ≠ i do
          {assert g[j].status ≠ empty;
           j: = (j + c) mod n}
6.    for all i: g[i].status = occupied ⇒
        p: = g[i].ptr;
        assert p^.ScopeNo ≤ g[top].ptr^.ScopeNo;
        while p^.next ≠ nil do
          {assert p^.ScopeNo > p^.next^.ScopeNo;
           p: = p.next}
7.    var num: integer;
      if top in [0 .. n − 1] then num: = g[top].ptr^.ScopeNo
      else num: = 0;
      for i: = num downto 1 do
       {while (top in [0 .. n − 1])
         and then (g[top].ptr^.ScopeNo = i) do
         {assert g[top].status = occupied;
          top, g[top].ptr: = g[top].ptr^.PreviousIndex, g[top].ptr^.next}}
assert top = n;
assert for all i: 0 ≤ i < n: g[i].ptr = nil;
```

Example: This example deals with the temporary redirection of pointers for specific purposes. The idea is due to Deutch, Schorr, and Waite [11], and can be applied to several cases. We briefly develop the data structure used in their efficient algorithm for garbage collection. Assume that each memory word (or node) contains a bit `m` which is used for marking, a bit `a` which is set to `1` for *atomic* nodes and to `0` for *list* nodes. Atomic nodes contain some information and list nodes contain two addresses, viz., `ALink` and `BLink`. Assume that `nil` is the address of a node with `a = 1`.

Step 1: A simple garbage collection algorithm uses a stack and employs a depth-first marking algorithm [11]:

```
as: stack of node;
```

Step 2: The stack can be implemented using list nodes. The data structure diagram is shown in

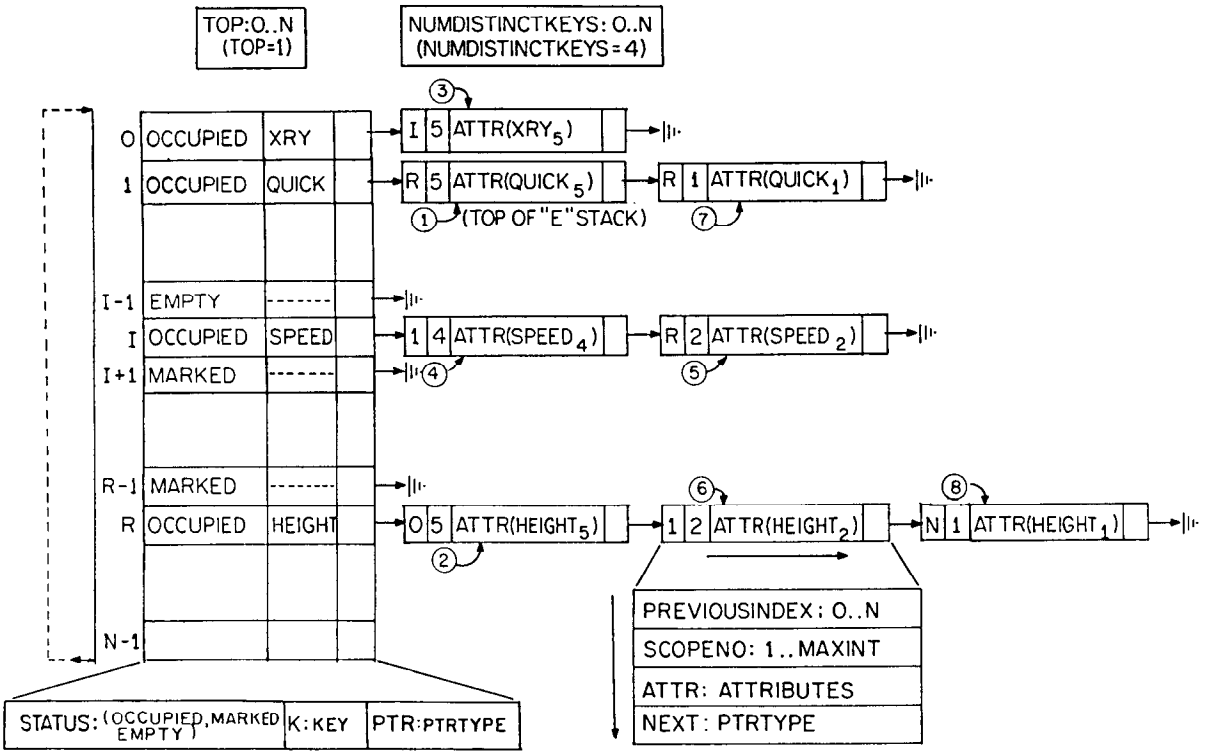

FIGURE 5. Final Data Structure for the Symbol Table Module

Figure 6. Let `cs` (*concrete stack*) be the pointer to the top of the stack.

Step 3: For this case, we assume that if `cs` is not `nil` then (i) if `csˆ.a = 1`, then `csˆ.BLink` points to the item at the top of the stack and `csˆ.ALink` points to the rest of the stack; (ii) if `csˆ.a = 0` then `csˆ.BLink` points to the rest of the stack while `csˆ.ALink` is unused. The data structure diagram is shown in Figure 7.

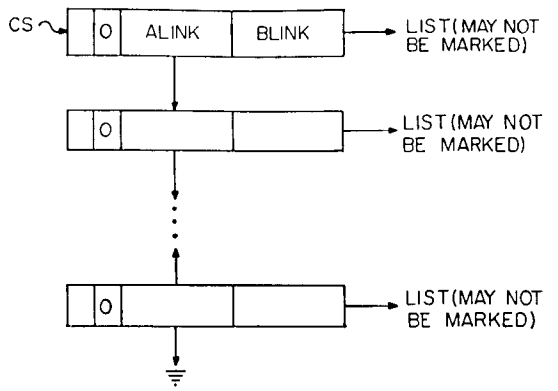

FIGURE 6. Implementation of Stack Using List Nodes

Step 4: The remarkable observation made by Deutch, Schorr, and Waite was that this stack can be

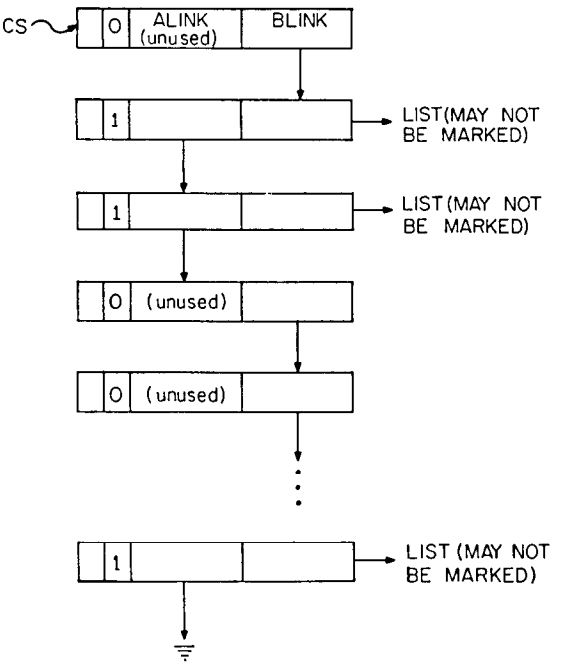

FIGURE 7. Implementation of Stack Using Modified List Nodes

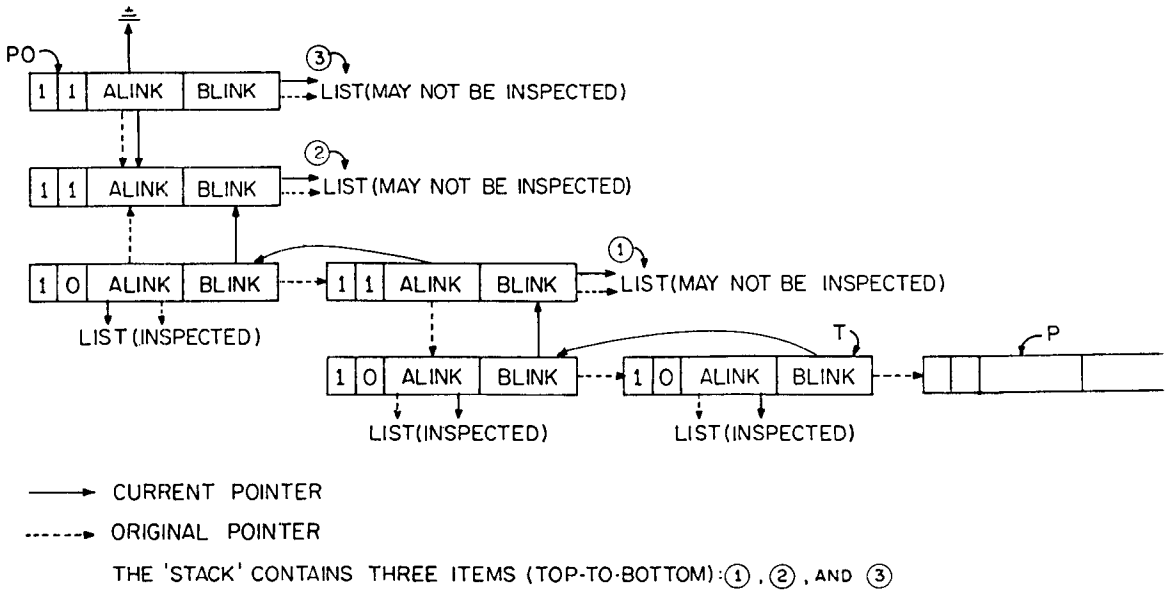

FIGURE 8. Diagram of the List Structure

maintained within the list being marked! The algorithm maintains P and T (corresponding to cs above). The data structure diagram is shown in Figure 8. PO is the initial pointer passed to the marking routine. The mapping is:

```
while not empty(as) do
  {while T^.a = 0 do
     {q: = T^.BLink; T^.BLink: = P;
      P: = T; T: = q}
   assert top(as) = T^.BLink;
   pop(as);
   q: = T^.ALink; T^.a: = 0; T^.ALink: = P;
   P: = T; T: = q}
while T^.a = 0 do
  {q: = T^.BLink; T^.BLink: = P;
   P: = T; T: = q}
assert T = nil;
assert state of the heap storage at this
  point is the same as its state in the
  abstract stack version.
```

The invariant assertion is:

```
while T ≠ nil do
  {assert S(T^.m) = 1;
     -- S(y) ≡ current state of y
   assert S0(T^.m) = 0;
     -- S0(y) ≡ initial state of y
   assert S0(T^.a) = 0;
   if S(T^.a) = 0 then
     {assert S(T^.ALink) = S0(T^.ALink);
      assert S0(T^.BLink) = P;
      marked(S(T^.ALink)); -- see below
      P: = T; T: = T^.BLink}
   else {assert S(T^.BLink) =S0(T^.BLink);
         assert S0(T^.ALink) = P;
         P: = T; T: = T^.ALink}}
```

where

```
marked(q) ≡
  {assert q^.m = 1;
   if q^.a = 0 then {marked(q^.ALink);
                     marked(q^.BLink)}}
```

For the details of the final program, refer to [11]. The above development shows that a data structure can be understood without full details of the underlying algorithm.

Observation: On the basis of the examples discussed here, we conjecture that the complexity of a data structure can be measured by estimating the complexity (e.g., size) of the mapping specification and invariant assertion. These capture the amount of knowledge required to understand programs using the given data structure.

5. SUMMARY

We have hypothesized that data structures are not inherently complex once a certain level of knowledge is assumed. For example, there are cases where it is easier to understand a program using abstract set operations rather than array operations. The complexity of a program increases with the increase in the opaqueness of the mapping between the representation and abstract data types. We have discussed the results of an experiment conducted to validate this hypothesis.

On the basis of our observation, we have discussed documentation techniques for data structures that make it easier to understand a program. These include both informal and formal methods. These were illustrated with the data structures used in the

Deutch-Schorr-Waite algorithm and a symbol table for Pascal-like scope rules.

Some related research issues are (i) the experimental verification of the efficacy of the proposed documentation techniques, (ii) the development and formalization of a set of notations for constructing data structure diagrams, and (iii) relating the mapping and invariant assertions to the axiomatic specification of the data structure.

Acknowledgments. The authors wish to thank John Fuller for helping with the analysis of the results of the experiment, and Ben Schneiderman, Harry Dunsmore, and four anonymous referees whose detailed comments and suggestions helped to enhance the clarity of this article.

REFERENCES

1. Baker, A.L., and Zweben, S.H. A comparison of measures of control flow complexity. *IEEE Trans. Softw. Eng. SE-6*, 6 (Nov. 1980), 506–512.
2. Bastani, F.B. An approach to measuring program complexity. In *Proceedings of COMPSAC '83.* (Chicago, Ill., Nov. 1983), 1–8.
3. Brown, M.H., and Sedgewick, R., Techniques for algorithm animation. *IEEE Softw.* 2, 1 (Jan. 1985), 28–39.
4. Curtis, B. et al. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Trans. Softw. Eng. SE-5*, 2 (Mar. 1979), 96–104.
5. Gilb, J. *Software Metrics.* Winthrop, Cambridge, Mass., 1977.
6. Guttag, J.V., Horowitz, E., and Musser, D.R. Abstract data types and software validation. *Commun. ACM 21*, 12 (Dec. 1978), 1048–1064.
7. Halstead, M.H. *Elements of Software Science.* Elsevier North-Holland, New York, 1977.
8. Henry, S., and Kafura, D. Software structure metrics based on information flow. *IEEE Trans. Softw. Eng. SE-7*, 5 (Sept. 1981), 510–518.
9. Iyengar, S.S., Parameswaran, N., and Fuller, J. A measure of logical complexity of programs. *Comput. Lang. 7*, 4 (Dec. 1982), 147–160.
10. Iyengar, S.S., Bastani, F.B., and Fuller, J.W. An experimental study of the complexity of data structures. *Symp. Emp. Found. Info. and Softw. Sc.*, Atlanta, Ga., Oct. 1984.
11. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms,* 2d ed. Addison-Wesley, Reading, Mass., 1973.
12. Matsumoto, Y. Management of industrial software production. *IEEE Computer 17*, 2 (Feb. 1984), 59–72.
13. McCabe, T.J. A complexity measure. *IEEE Trans. Softw. Eng. SE-2*, 4 (Dec. 1976), 308–320.
14. Oviedo, E. Control flow, data flow and program complexity. In *Proceedings of COMPSAC '80.* (Chicago, Ill., Oct. 1980), 146–152.
15. Robinson, L., and Levitt, K.N. Proof techniques for hierarchically structured programs. *Commun. ACM 20*, 4 (Apr. 1977), 271–283.
16. Schneiderman, B. Measuring computer program quality and comprehension. *Int. J. Man-Mach. Stud. 9* (1977), 465–478.
17. Schneiderman, B. Control flow and data structure documentation: Two experiments. *Commun. ACM 25*, 1 (Jan. 1982), 55–63.
18. Weissman, L. A methodology for studying the psychological complexity of computer programs. Tech. Rep. CSRG-37, Dept. of Computer Science, Univ. of Toronto, 1974.
19. Weissman, L. Psychological complexity of computer programs: An experimental methodology. *ACM SIGPLAN Not. 9*, 6 (Jun. 1974), 25–36.
20. Wile, D.S. Type transformations. *IEEE Trans. Softw. Eng. SE-7*, 1 (Jan. 1981), 32–39.
21. Woodward, M.R., Hennel, M.A., and Hedley, D. A measure of control flow complexity in program text. *IEEE Trans. Softw. Eng. SE-5*, 1 (Jan. 1979), 45–50.
22. Wulf, W.A. London, R.L., and Shaw, M. An introduction to the construction and verification of ALPHARD programs. *IEEE Trans. Softw. Eng. SE-2*, 4 (Dec. 1976), 353–365.

CR Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics—*complexity measures*; E.1 [**Data**]: Data Structures
General Terms: Experimentation
Additional Key Words and Phrases: abstract and concrete data structures, abstract data types, data structures, data structure diagrams, data structure invariants, logical complexity

Received 6/84; revised 11/86; accepted 11/86

Authors' Present Addresses: Farokh B. Bastani, Department of Computer Science, University of Houston, Houston, Tex. 77004. S. Sitharama Iyengar, Department of Computer Science, Louisiana State University, Baton Rouge, La. 70803-4020.