AN INTRODUCTION TO ROBUST DATA STRUCTURES

J. P. Black, D. J. Taylor,
and D. E. Morgan

Department of Computer Science
and
Computer Communication Networks Group
University of Waterloo
February, 1980

CS-80-08

Abstract

The addition of redundancy to data structures can be used to improve a software system's ability to detect and correct errors, and to continue to operate according to its specifications. This paper discusses some of the foundations of the study of redundancy, particularly its use in making data structures robust. The paper includes discussion of two appropriate error models, the basic assumptions required for a mathematical framework, and several definitions. It also gives a brief indication of some theoretical and empirical results which have been obtained.

1. INTRODUCTION

One approach to increasing the overall reliability and fault tolerance of a computer system makes use of redundant information in the system's data structures. However, current techniques for adding and exploiting such redundancy are not well developed, and are largely ad hoc. Our purpose in this paper is to present a formalism for discussing redundancy in data structures, its effectiveness and costs, and to discuss briefly some theoretical and practical results which have been obtained in our research.

Section 2 contains a very general discussion of redundancy in software systems, as well as an overview of related and previous work. Section 3 introduces our formalism for "robust" data structures. This is followed in Section 4 by a presentation of the theoretical framework underlying our research. Section 5 discusses the complementary experimental basis, and Section 6 provides references to more advanced work, conclusions, and directions for further study.

2. REDUNDANCY AND ITS USES

Any fault tolerant system, whether hardware or software, relies on some form of redundancy to improve reliability. Examples of software redundancy are system

checkpoints, and the rollback and recovery programs which use them; redundant programs not required for normal system operation, such as audit programs; redundant pointers, identifier fields, and node counts stored in data structures; and information about correct or expected system operation. Given appropriate redundancy, a fault tolerant software system uses it to detect and perhaps correct errors in the system.

As defined by Randell [1], the action taken by the system upon detecting an error may be classified as either backward error recovery or forward error recovery. Backward error recovery attempts to restore the system to some previous correct state, while forward error recovery attempts to modify or correct the erroneous system state and place the system in a new correct state The following discussion will make these terms more precise.

The classic example of backward error recovery is the checkpoint, rollback, and recovery method which has been used in transaction processing systems for many years. In this case, the redundancy consists of the checkpoint, transaction log, and the program for the recovery process. The method is quite general as it assumes nothing about the transaction in error, assumes only that the checkpoint file represents a correct system state, and that it is possible to decide whether or not a completing transaction leaves behind a correct system state.

A more structured technique along the same lines is that of the recovery block [1]. A recovery block consists of an acceptance test and a set of alternate blocks of code. At the end of the primary alternate, the acceptance test is applied to determine whether execution was successful. If it was, the recovery block terminates. If it was not, the system state is reset to its value on entry to the block, and the second alternate is attempted. Successive failures of the acceptance test cause successive alternates to be attempted, until the list is exhausted, in which case the recovery block returns an error indication, possibly to an enclosing recovery block. Recovery blocks can be efficiently implemented by a recovery cache which stores the previous value of each memory word modified inside the block [4]. Meaningful acceptance tests for recovery blocks must rely on some form of redundancy in order to determine whether an alternate has succeeded. Thus, while a massive checkpoint can be avoided by use of the recovery cache, applying a strong acceptance test may involve a significant execution time and/or storage overhead.

In some systems, backward error recovery is not acceptable, and so the less well understood approach of forward error recovery is used. The classic example is electronic telephone switching systems, where calls in progress cannot be cleared because of an error. This has led Bell Telephones to use "audit programs" whose purpose is

to detect and correct errors in the system's data structures without unnecessarily aborting existing calls unaffected by the error [2]. Waldbaum [11] has also proposed audit programs as a means of increasing operating system fault tolerance. In the case of audit programs, we consider the redundancy to be the programs themselves, the "knowledge" they embody about correct system states, as well as redundant information contained in the data structures being audited.

Compared to backward error recovery, forward error recovery can have the advantage of less system degradation or unavailability during recovery. On the other hand, it would seem to require a more detailed knowledge of the application and its data structures than backward error recovery.

3. ROBUST DATA STRUCTURES

A robust data structure is one which contains sufficient redundancy to permit the detection of errors in stored instances of the data structure. (Strictly speaking, this is an abuse of the terminology: given a data structure, such as a binary tree, we are interested in finding a storage structure or implementation of a binary tree which is robust in the face of errors. This terminology is due to Tompa [10].) Some robust data structures contain sufficient redundancy to allow correction

of errors. This detection and correction is performed by routines which operate only on the erroneous instance of the storage structure. Thus we are not concerned with detection and correction schemes which make use of, for example, a backup copy of the data structure. The error detection methods we consider are applicable to both forward and backward error recovery. The correction methods use forward error recovery to restore the system to a correct state.

There are two terms which are used to characterise the correctness of a data structure: semantic integrity and structural integrity. Semantic integrity [5] concerns the meaning of the information contained in the data structure: does it accurately represent a possible configuration of those real entities which it is supposed to model? Structural integrity [12] refers to the more general characteristics of the way in which the data are organised: Do pointers have the proper values, are all nodes properly linked into the structure?

Because of the difficulty of making generalisations about the semantic integrity of data structures, our work to date has concentrated on the structural integrity or correctness of an instance of a data structure. That is, we are concerned with the effects of errors on the overall structural characteristics of the data. We consider the following types of information in an instance as being structural: pointers, fields containing counts of the

number of nodes in the instance, and identifier fields. An identifier field in a correct instance contains a value which is unique over all node types and all instances of the same or different storage structures in the system. Thus, given an arbitrary node containing an identifier field, one can determine to which particular instance in the system the node belongs.

We consider storage structures whose instances each consist of a set of nodes with a distinguished "header". The nodes and header are assumed to be connected by pointers, with all nodes accessible by following some sequence of pointers from the header. (In graph-theoretic terms, our instances are rooted, directed graphs.) Relative to a particular instance, all nodes in the system are either inside or outside the instance, depending on whether or not they are accessible from its header. We assume that the data structure is subject to a source of errors, and we seek to make precise statements about the number and kinds of errors which may be detected or corrected.

Our research is concerned with the effect on data structure robustness of the addition of redundant structural information. In order to make precise statements about the robustness of a given structure, it is necessary to state our assumptions about the errors occurring in the structure. We consider two types of error source: an "intelligent adversary" source, and a random source, both introducing

errors into the structural information contained in an instance.

Our theoretical results assume that the errors are due to an intelligent adversary attempting to do a maximum amount of damage with a minimum of effort. For instance, we wish to answer a question such as the following for a given data structure: what is the minimum, over all possible instances, of the number of structural changes required to transform one correct instance into another? Or equivalently, what is the maximum number of changes, made to a correct instance, which may be guaranteed to produce an incorrect instance? (In order not to obscure this presentation, we will postpone strict definitions of these terms until the next section.) This intelligent adversary model permits us to make strong statements about the robustness of given structures, but it deals only with worst case results.

In order to obtain an estimate of the "average" or "effective" robustness of a data structure, we assume a (pseudo-) random error source, and perform experiments in which errors are introduced into data structures by a "mangler". As explained more fully below, we have shown the effective robustness to be significantly greater than the worst case results indicate.

Thus, we make very few assumptions regarding the causes of errors in data structures. They may be due to

misbehaving programs; programs which are forced to abort in the midst of an operation due, for example, to an operating system crash; errors in the supporting hardware; or mistakes by users.

The next section expands on the above intuitive ideas by giving a more precise overview of the theoretical framework required to study robust data structures.

4. A THEORETICAL FRAMEWORK FOR ROBUST DATA STRUCTURES

As indicated above, we are seeking storage structures which are robust in that some number of (erroneous) changes applied to an instance of the structure may be detected as producing an incorrect instance. A further goal is to use forward error recovery to modify the changed instance so that it is once again correct. This motivates our use of two types of procedure: detection procedures and correction procedures.

A detection procedure, when applied to a storage structure instance, either accepts or rejects the instance as being (structurally) correct. Thus, we are using the text of the procedure as an implicit definition of what constitutes a correct instance. Once an instance has been found incorrect by a detection procedure, one may wish to use a correction procedure to modify the instance so that it becomes acceptable to the detection procedure. Trivially, an "empty" instance satisfies this requirement, but this is

clearly not what we want. Rather, the correction procedure should modify the instance so that it is "almost the same as" the original instance, while also being correct.

In order to quantify the detection and correction capabilities of storage structures, we need to be able to quantify modifications to them, whether erroneous or not. For this purpose, the term change is defined to be the alteration in type and/or value of a single elementary memory item. This definition may be adjusted to suit a particular application, but would normally be taken to be any modification resulting from the execution of a single store instruction. Examples of single changes to the structural information in an instance are replacing a pointer by null or by a pointer to a different node, incrementing a count of the number of nodes in an instance, or placing a new value in an identifier field.

During normal system operation, update routines introduce changes into structural information to produce new, correct instances from old ones. However, such routines may periodically introduce erroneous changes due to the presence of a bug, or may fail because of an operating system crash, in which case the partially updated instance is left some number of changes away from both the initial configuration and the desired final configuration.

Given this definition of change, quantifying the robustness of a storage structure is straightforward. If a

single change can transform one correct instance into another, the storage structure has no detection capabilities. In general, if at least n + 1 changes are required to transform one correct instance into another, n change detection is possible, and such a storage structure is called n-detectable. (Strictly speaking, the n-detectability is a property of a particular detection procedure; the procedure rejects all correct instances modified by n or fewer changes.) Similarly, if n changes applied to a well-formed instance may be corrected (by some correction procedure), the storage structure is said to be n-correctable.

Before stating the kinds of results which may be obtained with these definitions as a starting point, we need to discuss an important assumption which is required for a rigorous mathematical framework. The "valid state hypothesis" (VSH) has two parts: one for identifier fields, and one for pointers. For identifier fields in a valid system state, the hypothesis is: for each identifier field in each kind of node in each storage structure instance, there is a unique identifier field value which is stored in that field; this value is not stored in any other location which could be used to store that identifier field. Secondly, for pointers, the only pointers to a node are found in other nodes which belong to the storage structure instance containing that node.

The valid state hypothesis is quite restrictive as it requires, in particular, that the system memory outside an instance be "clean", that is, it contains neither pointers pointing back into the instance, nor identifier field values belonging to the instance. It is possible to relax the hypothesis by taking into consideration the amount of "invalidity" in the system [7]. However, the complication which this introduces would obscure our presentation here.

This completes the basic framework on which our theoretical results rest. In order to make the ideas presented more clear, to introduce some additional concepts related to detectability, and to indicate the type of analysis possible, we present a simple example: the linear list.

Figure 1 shows a common implementation of a linear list: the header contains a pointer to the first node in the list, each node contains a pointer to the next, and the final node contains a pointer back to the header. This storage structure has no detection capabilities (is 0-detectable): changing a single pointer can shorten the list and still leave a correct instance.

Figure 2 shows a slightly more robust storage structure for linear lists: the header now also contains a count of the number of nodes in the instance, and each node contains an identifier field. This structure is 1-detectable, although still 0-correctable. Intuitively, the count field

permits detection of pointer changes which shorten or lengthen the apparent list, while the identifier fields permit detection of pointer changes which cause foreign nodes to appear as part of the changed instance. In fact, it can be shown that any storage structure which fits our model and which contains only one pointer to every node can be made 1-detectable by the addition of count and identifier fields.

Figure 3 shows a double linked list implementation which is not only 2-detectable, but also 1-correctable. This storage structure differs from that of Figure 2 by the addition of a redundant set of back pointers. Although we could demonstrate the 2-detectability by an exhaustive case analysis of all pairs of changes, there are at least two better approaches. We will discuss one here, which involves determining the minimum number of changes required to transform one correct instance into another by evaluating the three quantities ch-same, ch-repl, and ch-diff.

Ch-same is the minimum number of changes required to transform one correct instance over a set of nodes into a different correct instance over the same set of nodes. Any such rearrangement for the double linked list requires changing at least two forward and two back pointers, as is easily verified, giving ch-same = 4.

Similarly, ch-repl is the minimum number of changes to form a new correct instance by replacing one or more nodes

with the same number of foreign nodes. Assuming a fortuitously correct structuring of these foreign nodes, at least two incoming and two outgoing pointers need to be changed, as there can be no pointers already present from the foreign nodes into the instance under VSH. Furthermore, proper identifier fields must also be created for the foreign nodes, giving a minimum of ch-repl = 5, in the case of one foreign node.

Thirdly, ch-diff is defined as the minimum number of changes required to change one instance into another with a different number of nodes. As identifier field values must be supplied for foreign nodes, the minimum clearly occurs for deletion. Deleting any node requires changing two pointers and the count, yielding ch-diff = 3.

Taken together, ch-same, ch-repl, and ch-diff exhaust the ways of changing one correct instance into another, and as the detectability is defined to be one less than this value, we have shown that a double linked linear list is 2-detectable (i.e., min(4,5,3)-1).

It is possible to increase the detectability of a double linked list to three by restructuring the back pointers. Figure 4 shows a linear list where each back pointer points to the second preceding node. A second "header" node has also been added. This is called a modified(2) double linked list. (It belongs to the class of modified(k) double linked lists, where the back pointer in

each node points to the kth preceding node. The detectability of such a list with k = 3 is four, but increasing k further has no effect on the detectability.) In order to show the 3-detectability, we will use the concepts "k-determined" and "k-count-determined" to bound ch-same, ch-repl, and ch-diff from below, although we could use a similar argument to the one used above.

We say a set of pointers in a storage structure determines the structure if all count fields, identifier fields, and other pointers can be reconstructed given only the determining set. A structure is then k-determined if there exist k disjoint sets of pointers, each of which determines the structure. Similarly, a storage structure is k-count-determined if there exist k disjoint sets of pointers, each of which can be used to determine the number of nodes in the structure. A modified(2) double linked list is 2-determined since either forward or back pointers can be used to reconstruct an instance, and it is 3-count-determined as the forward pointers or either of the two sets of interleaved back pointers may be used to calculate the number of nodes.

Now, for a k-determined storage structure, ch-same $\geq$ 2k since a single change in any one of the k sets cannot rearrange the nodes in that set. If m of the k sets must have an equal number of pointers entering and leaving a node, and there are n identifier fields, ch-repl $\geq$ k + n + m

under VSH. These give ch-same $\geq$ 4 and ch-repl $\geq$ 5 for a modified(2) double linked list. Neither of these bounds is tight in this case, but they are sufficient, since the bound on ch-diff is smaller and tight: if a k-count-determined implementation contains j stored counts, ch-diff $\geq$ k + j, which gives ch-diff $\geq$ 4. In particular, the bound is obtained when an arbitrary list is changed into the empty list, in which case the count, a forward pointer, and two back pointers must be changed in the header nodes. Together, these show that the modified(2) double linked list is 3-detectable. (The proofs of these results may be found in [9]).

Returning to the ordinary double linked list, we now wish to show that this storage structure is also 1-correctable. Again, this could be done by exhaustive analysis; indeed, the reader is encouraged to do so. However, we prefer to apply a more general result, called the General Correction Theorem [6]. The theorem states that a storage structure which is 2r-detectable and has r + 1 edge-disjoint paths to each node is r-correctable. The requirement for r + 1 edge-disjoint paths to each node precludes the disconnection (and loss) of any node by r or fewer changes. (This result is proven in [6].)

Applying the theorem to double linked lists with r = 1, we find they are 1-correctable. Note that the theorem applies equally to modified(2) double linked lists; while

they have higher detectability, their correctability is still one. Although not presented here, a 1-correction routine has been developed for this structure whose execution time is linear in the size of the list. This routine can be adapted for modified(2) double linked lists. We have also developed a storage structure for binary trees which is 2-detectable and 1-correctable.

Other theoretical results provide bounds on detectability, ch-same, ch-repl, and ch-diff , and discuss a restricted class of "compound" storage structures [9]. As mentioned, the results have also been extended to cases where the valid state hypothesis cannot be used.

5. EXPERIMENTAL EVALUATION OF ROBUST DATA STRUCTURES

The theoretical results provide only a partial view of data structure robustness, as they are concerned with the worst case results which an intelligent adversary could produce. A more complete view can be obtained by investigating the "effective" robustness in the face of a random error source. This section discusses the types of experiments we have performed which indicate that the effective robustness is much higher than the theory suggests.

All of the experiments make use of a "mangler" to inject pseudo-random errors into data structures as they are written to external storage. One of the mangler's

parameters is the probability that any one record will be changed when written. If the mangler decides to mangle a record, it uses some probabiltiy distribution to determine which word in the record to change. Finally, a small integer (positive or negative) is added to the chosen word. A small integer is used rather than an arbitrary value in order to make the changes more subtle; an arbitrary value is often too easy to detect. Thus the experiments consist of executing a program which uses a data structure on external storage, injecting errors into the structure with the mangler, and observing the program's reaction. Needless to say, the program expects errors to occur, and contains some logic for error detection and correction.

One set of experiments was performed on storage structures whose detectability could be determined from the theoretical analysis. The experiments were designed to investigate the detectability as well as the probability of node disconnection. The experiments compared three different storage structures for linear lists. The experiments showed that when the detectability was greater than one, no undetected errors occurred, even though up to twenty errors were injected by the mangler into a fifty-node structure. Modified(2) double linked lists were significantly more resistant to random disconnection than double linked lists: for five injected errors, the observed probability of disconnection was 65% for double linked lists

compared to 8% for modified(2), even though they have the same number (2) of edge-disjoint paths to each node.

The second set of experiments was concerned with a sample data base system of moderate complexity. This "example system" provides random retrieval by key through an externally chained hash table, and a query facility based on inverted lists. The theoretical results suggested the use of a 1-correctable double linked list for each hash chain, but were less helpful for the inverted lists and their implementation. We thus added redundant count and identifier fields to the "index file", which contained the dynamic set of inverted lists, on an ad hoc basis. In order to provide error detection and recovery, a complete set of audit routines was written, including the linear time 1-correction algorithm mentioned above for the hash chains.

A set of experiments was performed using a standard script of commands, and different seeds for the mangler's random number generator. In order to make the experimentation feasible, both the mangling probability and the frequency of audit execution were set unrealistically high. Our results confirmed that it was possible to achieve high levels of system reliability with an acceptable overhead cost: over 92% of errors to fields used in normal program operation were corrected, and 94% of errors to redundant fields were corrected. We estimate that reducing the audit execution frequency in view of a reasonable error

rate would allow this level of reliability to be achieved with 7% I/O overhead, and 16% execution time overhead.

6. SUMMARY, CONCLUSIONS, AND FURTHER WORK

Our purpose has been to present some of the concepts underlying our research into robust data structures. As the subject is still in the developmental phase, we welcome all comments on the ideas we have presented here. More information on the theoretical and practical results obtained to date may be found in [3, 7, 8, 9].

Section 2 situated our research in the larger area of software fault tolerance. Section 3 provided an introduction to our concept of redundancy and robust data structures. In Section 4, we gave a broad outline of the theoretical basis for our research, which was complemented in the following section by a brief description of some experimental results.

Much further work is easily identified. An obvious extension is to introduce semantic correctness into the framework. While this appears very challenging in general, it should be possible to obtain limited results for common semantic notions such as keys. Formal specification of what constitutes a correct instance appears to offer some aid to determining the robustness of a storage structure as well as to constructing special purpose detection and correction routines. We have as yet no results for "parts" of

instances, nor for discussing the robustness of a structure such as "list of trees", given that the robustness of "list" and "tree" are known. It would be quite interesting to apply the techniques mentioned here to a large software system in production use, in order to gain a better idea of the costs and effectiveness of robust data structures.

The major goal of our research is to find where and how to apply redundancy to yield cost-effective fault tolerant systems.

BIBLIOGRAPHY

1. Anderson, T., and B. Randell (eds.). Computing Systems Reliability. Cambridge University Press, 1979.

2. Beuscher, Hugh J., George E. Gessler, D. Wayne Huffman, Peter J. Kennedy, and Eric Nussbaum. Administration and maintenance plan. Bell System Technical Journal, vol. 48 (October 1969). pp2765-2815.

3. Black, J. P., D. J. Taylor, and D. E. Morgan. A case study in fault tolerant software. In preparation.

4. Lee, P. A., N. Ghani, and K. Heron. A recovery cache for the PDP-11. Digest of Papers, The Ninth Annual International Symposium on Fault-Tolerant Computing. Madison, Wisconsin, June 20-22, 1979. pp3-8.

5. Minsky, N. Files with Semantics. Proceedings, ACM SIGMOD International Conference on the Management of Data. Washington, D. C., June 2-4, 1976. pp65-73.

6. Taylor, David J. Robust data structure implementations for software reliability. Ph.D. Thesis, Department of Computer Science, University of Waterloo, Ontario, 1977.

7. Taylor, D. J. Theoretical foundations for robust data structure implementations. Available as Computer Science Research Report CS-78-52, University of Waterloo, Ontario, Canada. December 1978.

8. Taylor, D. J., D. E. Morgan, and J. P. Black. Redundancy in data structures: Improving software fault tolerance. Submitted to IEEE Transactions on Software Engineering. Also available as Computer Science Research Report CS-79-34, University of Waterloo, Ontario, Canada. September, 1979.

9. Taylor, D. J., D. E. Morgan, and J. P. Black. Redundancy in data structures: Some theoretical results. Submitted to IEEE Transactions on Software Engineering. Also available as Computer Science Research Report CS-79-35, University of Waterloo, Ontario, Canada. September, 1979.

10. Tompa, Frank W. Data structure design. Data Structures, Computer Graphics, and Pattern Recognition, edited by A. Klinger, et al. New York,

Academic Press, 1977. pp3-30.

11. Waldbaum, G. Audit programs--a proposal for improving system availability. IBM Research Report, Yorktown Heights, February 26, 1970 (RC2811).

12. Wilkes, M. V. On preserving the integrity of data bases. The Computer Journal, vol. 15, no. 3 (August 1972). pp191-194.

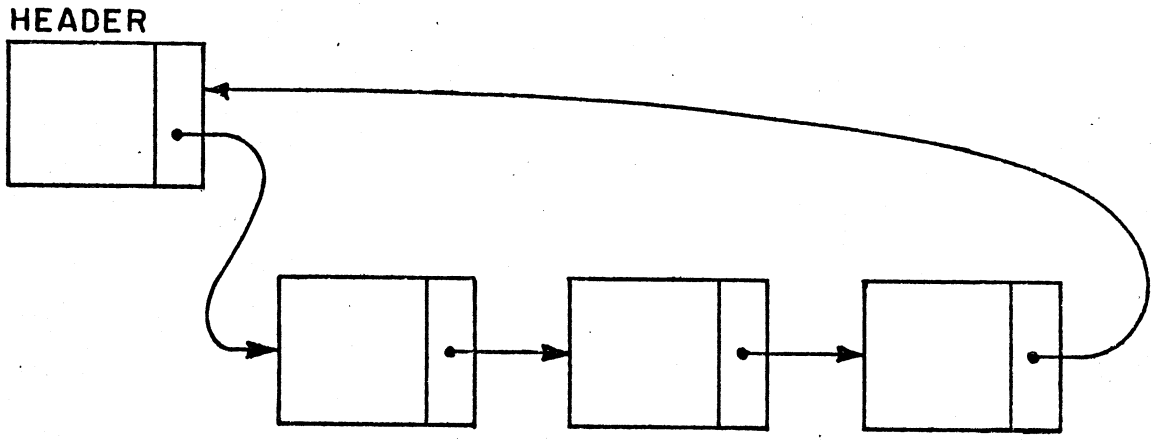

FIGURE 1. A simple linear list implementation.

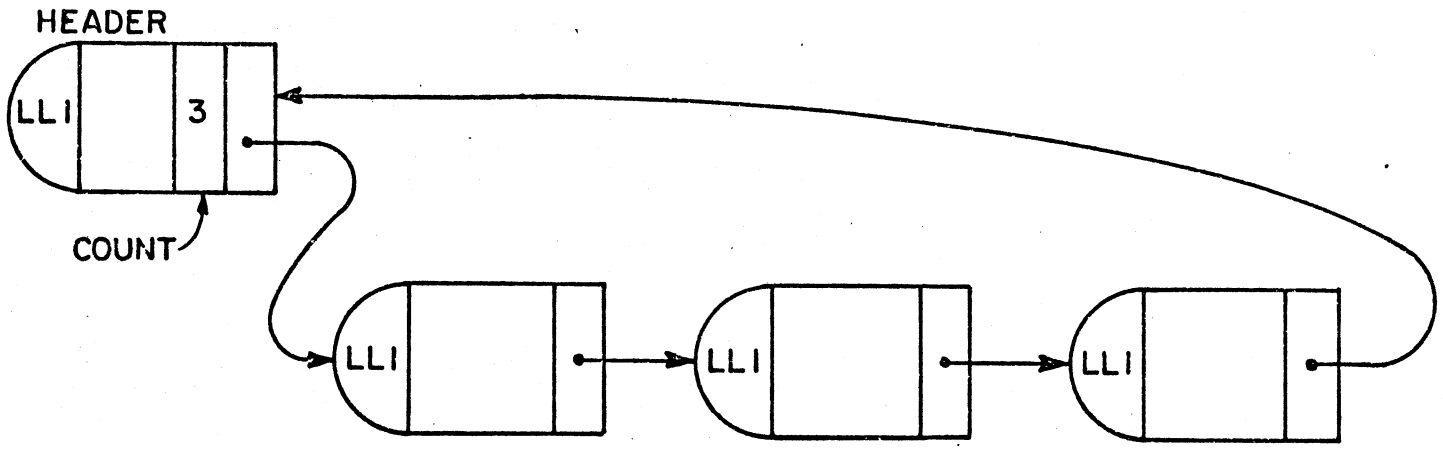

FIGURE 2. A 1-detectable linear list.

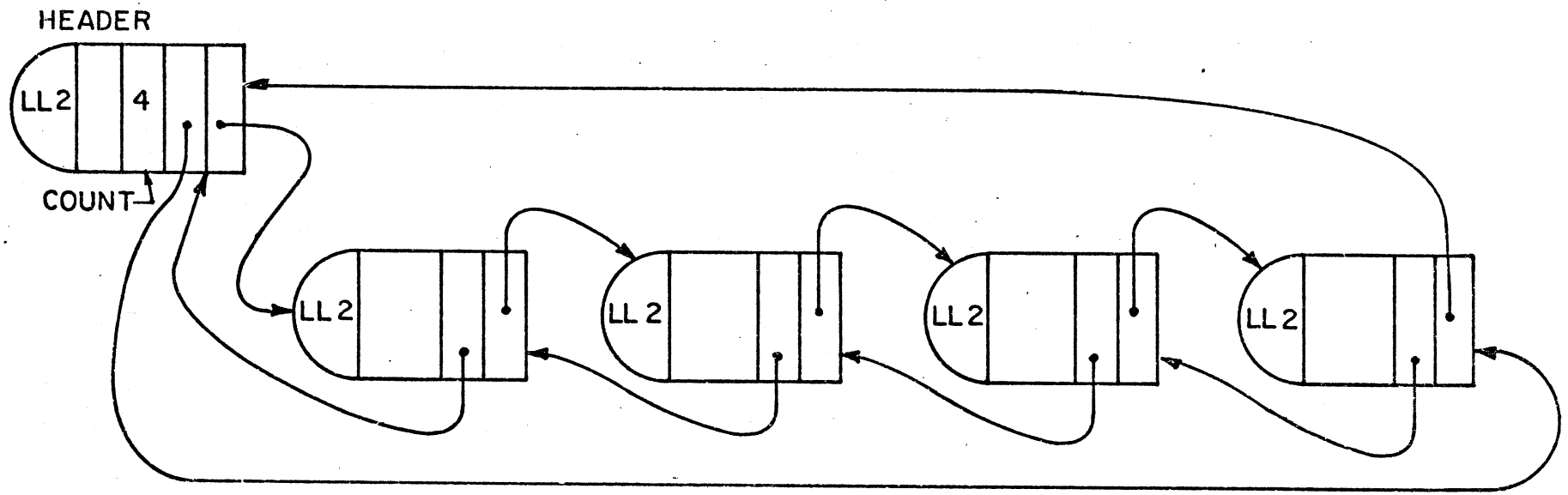

FIGURE 3. A double linked list.

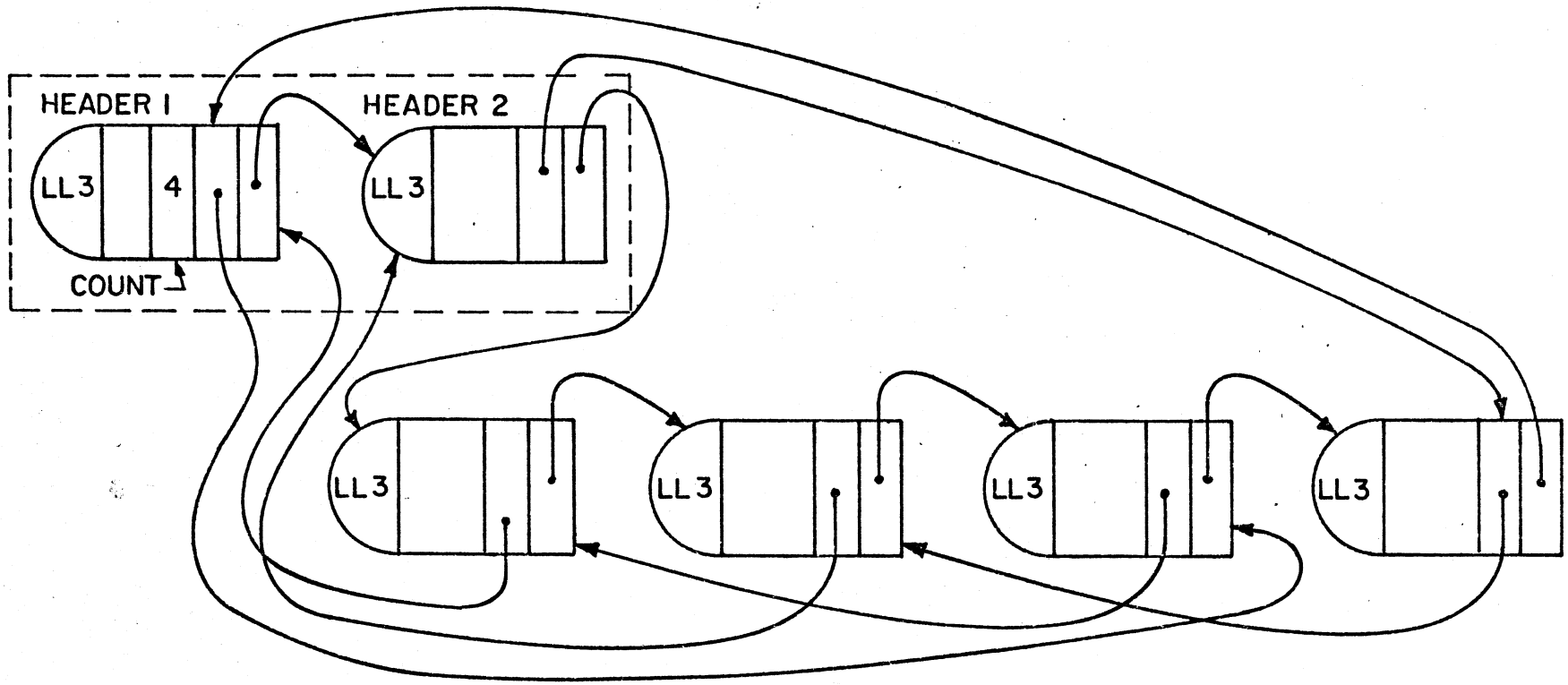

FIGURE 4. A modified (2) double linked list.