REDUNDANCY IN DATA STRUCTURES: IMPROVING SOFTWARE FAULT TOLERANCE

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

Department of Computer Science
and
Computer Communications Networks Group
University of Waterloo
September, 1979

CS-79-34

1. INTRODUCTION

The increase in complexity and size of modern software systems and the increase in society's dependence on computer systems has been accompanied by an increase in the costs associated with their failure. This has in turn created an interest in achieving reliable, fault tolerant systems. In this pair of papers, we discuss one particular approach to increasing fault tolerance: the detection and correction of errors in stored data structures. The first paper presents a survey of related work, terminology, and an informal development of the approach, including some experimental results. The second paper complements this with a more rigorous treatment of the basic results, as well as extensions to them, and concludes with some remarks on the synthesis of robust data structures.

Avizienis [3] defines two complementary approaches to achieving software reliability: fault intolerance and fault tolerance. The former includes techniques applied during system development to ensure that the running system satisfies all reliability criteria a priori: examples are proofs of program correctness, structured design and programming methodology, and development aids for systematic testing and debugging. This approach cannot cope with residual design flaws, bugs, hardware malfunctions, or user errors, all of which suggest using the complementary

approach of fault tolerance. This approach attempts to increase reliability by designing the system to continue to provide service in spite of the presence of faults.

The need for software to cope with its own errors, errors introduced by undetected hardware faults and mistakes by users seems to have been recognised first by designers of real-time control systems, notably telephone switching systems [4, 8]. More recently, designers of operating systems and data base systems have also recognised that detection and correction of software-induced errors are important. (An example of this concern is shown in the design of IBM's OS/VS2-2 [22, 23], which contains error detection and recovery routines not included in previous IBM operating systems.)

Redundancy is the key to error detection, correction, and recovery [2, 3, 20]. Redundant data in the system are essential in order to detect and recover from many types of hardware or software malfunctions. Special software is required to maintain and make effective use of the redundant data; some redundant data may be essentially coded into this software. Other types of malfunctions can only be detected by observing the behaviour of the system, and comparing this with known or expected behaviour derived from analysis of a system model or experience using the system. There are, in fact, four forms of redundancy which can be used to enhance system fault tolerance: redundant hardware, redundant

software, redundant data, and redundant information about the system's behaviour. A compromise between the cost of failures and the cost of the facilities necessary to cope with them determines the amount and type of redundancy which should be used to improve reliability.

Recovery relies on the use of redundancy to reconstruct damaged data. The redundancy can be in the form of backup copies or can be achieved by using redundancy in the representation of the data. A systematic recovery technique is the use of "recovery blocks" and the associated "recovery cache" [21]. For each recovery block, a sequence of alternate sections of code is provided. Each alternate is tried in turn until the "acceptance test" associated with the recovery block is satisfied. After each unsuccessful alternate, the system state is reset from the recovery cache to its contents on entry to the block. Useful acceptance tests depend on the existence of appropriate redundancy.

Various ad hoc techniques have also been developed for using redundant representations of data for recovery. For example, Waldbaum [27] summarizes some techniques for use with a set of linked control blocks, and Lockemann and Knutsen [15] describe a particular technique for use in correcting disk allocation data after a system crash.

A general description of error recovery in a data base environment is given by Fry and Sibley [9]. Many recovery techniques for data base systems make use of backup copies

of portions of the data base. Some ways of making use of such backup copies in recovery are described in [6, 16].

Ideally, there should be systematic techniques for synthesising the software components of a system to achieve specified levels of reliability and recoverability. There is also a need to analyse software systems to predict or measure their reliability. Attempts are being made to satisfy these needs: a good survey may be found in [2]. Unfortunately, there is little underlying theory to explain why one technique is better than another or to assist in the development of new techniques. Even when it is realised that redundancy is the key to performing error detection, diagnosis, and recovery, there is no systematic technique for adding redundancy to stored data, or for exploiting such redundancy.

The major goal of our research is to find where and how to apply redundancy to yield cost-effective fault tolerant systems. A little redundancy, thoughtfully deployed and exploited, can yield significant benefits for fault tolerance; however, excessive or inappropriately applied redundancy is pointless. These papers illustrate one way of measuring and comparing the effectiveness of alternative ways of structuring data, the goal being to find storage structures that are robust in the face of errors and failures. The work is in a sense parallel to (and complements) that of Gotlieb and Tompa [11] which provides a

technique for selecting a storage structure from a set of alternatives based on efficiency considerations.

2. TERMINOLOGY

In discussing fault tolerance, we will use some definitions suggested by Melliar-Smith and Randell [18]. A failure occurs when a system does not meet its specifications: it is an externally observable event. An erroneous state is a system state that can lead to a failure which we attribute to some aspect of that state. An error is that part of an erroneous state which can lead to a failure. A fault is a mechanical or algorithmic cause of an error. A fault tolerant system is one which attempts to prevent erroneous states from producing failures. This paper discusses the effect on fault tolerance of using redundancy in the representation of data structures.

The following definitions will be used in discussing data structures. A data structure is defined to be a logical organisation of data. A storage structure is a representation of a data structure. The representation specifies whether nodes are to be adjacent or connected by pointers, what pointers are used, and so on. An encoding of a storage structure is its representation on a particular storage medium. The encoding specifies how pointers are represented (absolute, relative, etc.), what fields are packed into a single word, and so on. Thus, "binary tree"

is a data structure; a representation in which there are pointers from each node to the left and right sons of the node is a storage structure for a binary tree; and if we also specify that pointers are stored as absolute addresses, that is an encoding of a binary tree. (This terminology is adapted from Tompa [26].)

We define a data structure instance to be a particular occurrence of a data structure. When the context makes the meaning clear, we will also use "data structure instance" to refer to the storage structure for the instance or its encoded form.

We define a change to be an elementary modification to the encoded form of a data structure instance. (The meaning of "elementary modification" can be specified to suit the environment; here it will mean the modification of a single word.) Since a change modifies a data structure instance at the encoding level, the effect of the change on the storage structure depends on the encoding used. There is a mapping which transforms a change into one or more "changes" in the storage structure. For simplicity, we will assume that each change corresponds to only one "change" in the storage structure unless the encoding is explicitly noted as packing two or more fields into one word. (Here, the only example of the latter case occurs in Section 3.2.) We do not specify the types of faults which cause the changes, but the following are possibilities: hardware faults; "wild" stores

by incorrect programs; incorrect update procedures; and incomplete execution of update procedures, possibly resulting from an unrelated event (e.g., an operating system crash). For this last possibility, the instance which has been partially updated is left some number of changes "away" from the initial configuration and from the final desired configuration.

To illustrate our definition of change, consider the following storage structure for a linear list. Suppose the list contains four items, each of the first three has a pointer to the next, and the last contains a null pointer:

A -> B -> C -> D -> NULL

If somewhere in storage there is a node which contains X and a null pointer, then a single change in the pointer of node C can produce:

A -> B -> C -> X -> NULL

This single change effectively replaces D by X.

In this paper, only changes affecting structural information (such as pointers, counts, and identifier fields) will be considered. That is, we are concerned here with structural integrity rather than semantic integrity. Semantic integrity [11, 16, 19] concerns the meaning of the data being represented: does it correspond to a possible configuration of the real world entities being described? Structural integrity [9, 28] concerns the correctness of the representation of the data: whether pointers have values in

the right range, whether internal structural redundancy is consistent, and so forth.

In order to discuss error detection and correction in instances of data structures, we must give our definition of a "correct" instance. For the purposes of this paper, we define an instance of a data structure to be "correct" if a "detection procedure" applied to the instance returns the value "correct".

Detection properties of a data structure encoding are stated in terms of changes. If a single change can transform a correct data structure instance into another correct instance, as in the linear list example above, the encoding has no detection capabilities. If at least two changes are required to transform any correct instance into another, then single change detection is possible. In general, if at least N changes are required to transform any correct instance into another, any set of one to N-1 changes can be detected.

If all sets of N or fewer changes can be detected, we say the encoding is N-detectable, (i.e., its detectability is N). We say an encoding is N-correctable (i.e., its correctability is N) if there is a procedure which, for all sets of N or fewer changes, can take a correct instance modified by that number of changes and recreate the correct instance. Thus, we are interested in computing the minimum number of changes to produce, say, an undetectable error.

(Note that N-detectability implies K-detectability for K < N, and similarly for correctability.) These definitions of detectability and correctability are related to Hamming's definitions for binary codes [13].

Although detectability and correctability are properties of the encoding of a data structure, it is often convenient to refer to them as properties of the storage structure. In general, the encoding itself will only be significant when it specifies that more than one field at the storage structure level is to be placed in a single word.

3. ROBUST STORAGE STRUCTURES

A robust storage structure is one containing redundant data which allow erroneous changes to be detected, and possibly corrected as well. Three commonly used forms of structural redundancy in data are: a stored count of the number of nodes in a structure instance, identifier fields, and additional pointers [27].

A count of the number of nodes in an instance is often useful for purposes other than reliability. It is also one of the most commonly-used techniques for improving the robustness of storage structures, since it is a simple technique to use and usually introduces little overhead.

An identifier field is a group of one or more words, usually at the beginning of a node, which explicitly

signifies the type of the node. The type of a node is usually defined implicitly by which pointers in what types of nodes point to it. We will assume that the type of each node, and hence proper identifier field values, can be determined from pointer data; that is, we will only consider identifier fields which provide redundant identification of node type.

Algorithms which work with a storage structure usually require certain pointers between nodes in order to perform their functions. Additional pointers whose values could be deduced from other pointers may be added to the structure. These redundant pointers are a powerful tool in increasing storage structure robustness. In addition, they may sometimes make algorithms which work with the storage structure simpler or more efficient. In passing, we point out our assumption that such redundant pointers are "independent". That is, when one is modified, another will not be modified in an identical way. If a field is duplicated, some types of faults (software bugs) will almost certainly have identical effects on all copies of the fields. However, this consideration applies only to the effective detectability observed during system operation; the theoretical results are not affected.

In this paper we consider storage structures which consist of a header and a (possibly empty) set of nodes. The header contains pointers to certain nodes of the

instance or to parts of itself and may also contain one or more counts and identifier fields. Each node contains data items and structural information, which may be pointers and node type identifier fields. (If the header contains more than one part, we assume that all parts are accessible without following intra-header pointers. This is generally accomplished by storing the parts of the header as a contiguous vector.)

When identifier fields are used in an instance, we assume that they contain a value which appears in no other identifier fields in the system. This is an aspect of the "valid state hypothesis", which is a basic assumption used throughout these papers. The hypothesis is that the only pointers to nodes of an instance occur in the instance, and that the unique identifier value(s) for the instance appear only in its own identifier fields. Insofar as the theorems in Part II are concerned, we point out that it is possible to relax the valid state hypothesis, at the price of complication of their statement and proof [25].

We present below examples of several storage structures for linear lists and binary trees, and informally discuss their robustness. We also give some practical implementation considerations. Our purpose is to clarify the terms presented above, give an intuitive motivation for the theoretical results presented in Part II, and demonstrate that these techniques can be easily applied in

practice.

3.1 Linear Lists

The easiest way of implementing a linear list is simply to store a pointer in each node to the next node of the list, placing a null pointer in the last node. Inserting nodes in, and deleting nodes from, instances of such a storage structure is quite simple and efficient, but the storage structure is not at all robust. Specifically, it is 0-detectable and 0-correctable: changing one pointer to null can reduce any list to the empty list. Such a storage structure contains no explicit redundancy and uses only one word of structural data in each node (the pointer field). Inserting a node in the list requires two changes, one in the inserted node and one in the preceding node.

A commonly-used storage structure which is more robust adds an identifier field to each node, replaces the null pointer in the last node by a pointer to the header of the list, and stores a count of the number of nodes on the list. In this "single-linked" implementation, an additional word is added to each node of the list, and four changes are required to insert a node: two pointers, an identifier field, and the count. It also has the effect of making the storage structure 1-detectable, although it is still 0-correctable.

The 1-detectability is easily seen. A change to the count may be detected by comparing it against the number of

nodes found by following pointers. An identifier change is trivial to detect. A pointer change may be detected either because the count does not agree, or because the changed pointer now points to a foreign node, which cannot have a valid identifier field under the valid state hypothesis. The reader can easily devise a pair of changes which produces a correct instance, thus proving that the storage structure is exactly 1-detectable. The 0-correctability is shown by the following example: modifying a pointer to shorten the apparent list makes it impossible to decide whether the count is wrong or whether some nodes have been deleted.

The most robust of commonly-used list storage structures is the double-linked list. A double-linked list is a single-linked list with a pointer added to each node, pointing to the predecessor of the node on the list. This adds one more word of storage per node and increases the number of changes for inserting a node to six: two forward pointers, two backward pointers, an identifier field, and the count. This storage structure is 2-detectable and 1-correctable, essentially because it has two independent, disjoint sets of pointers, each of which may be used to reconstruct the entire list. (This result is proven in Part II.)

Finally, we may consider a novel storage structure, which is similar to the double-linked one, but in which the

"backward" pointers point to the second preceding node rather than the immediately preceding node. The storage required per node is clearly the same as for a double-linked list, but one more change is required when inserting a node. (Three backward pointers must be changed, rather than two.) This storage structure, which is referred to as a "modified(2) double-linked list," is 3-detectable and 1-correctable. Figure 3.1 shows a modified(2) double-linked list of 5 nodes. (The parameter, 2, in the name is the distance spanned by the back pointer. A modified(3) double-linked list is 4-detectable, but still only 1-correctable. Increasing the parameter beyond 3 has no further effect on detectability or correctability.)

This last storage structure illustrates that the "standard" double-linked list implementation may not always be the best way of using two pointers per node in a linear list. If one is willing to pay a slight price in terms of update time, it is possible to achieve greater detectability using the modified(2) double-linked implementation.

We can summarize the robustness and the performance costs of these four storage structures in a "cost and effectiveness graph" (Figure 3.2).

How could these detectability and correctability results for double linked lists be expoloited in a real system? Could this be done at reasonable cost? Besides the obvious costs of increased storage and update time, what

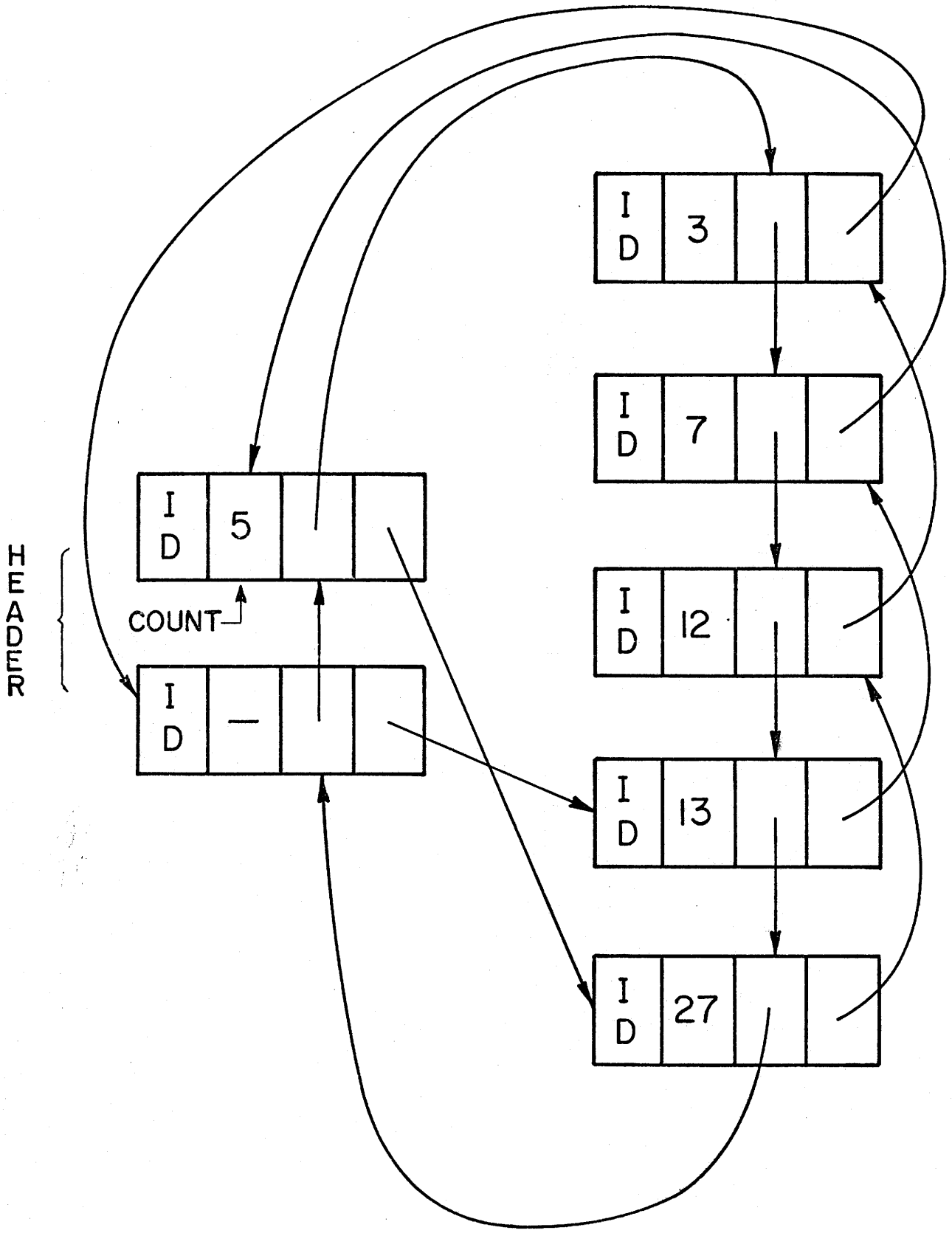

Figure 3.1 Modified (2) Double-Linked List

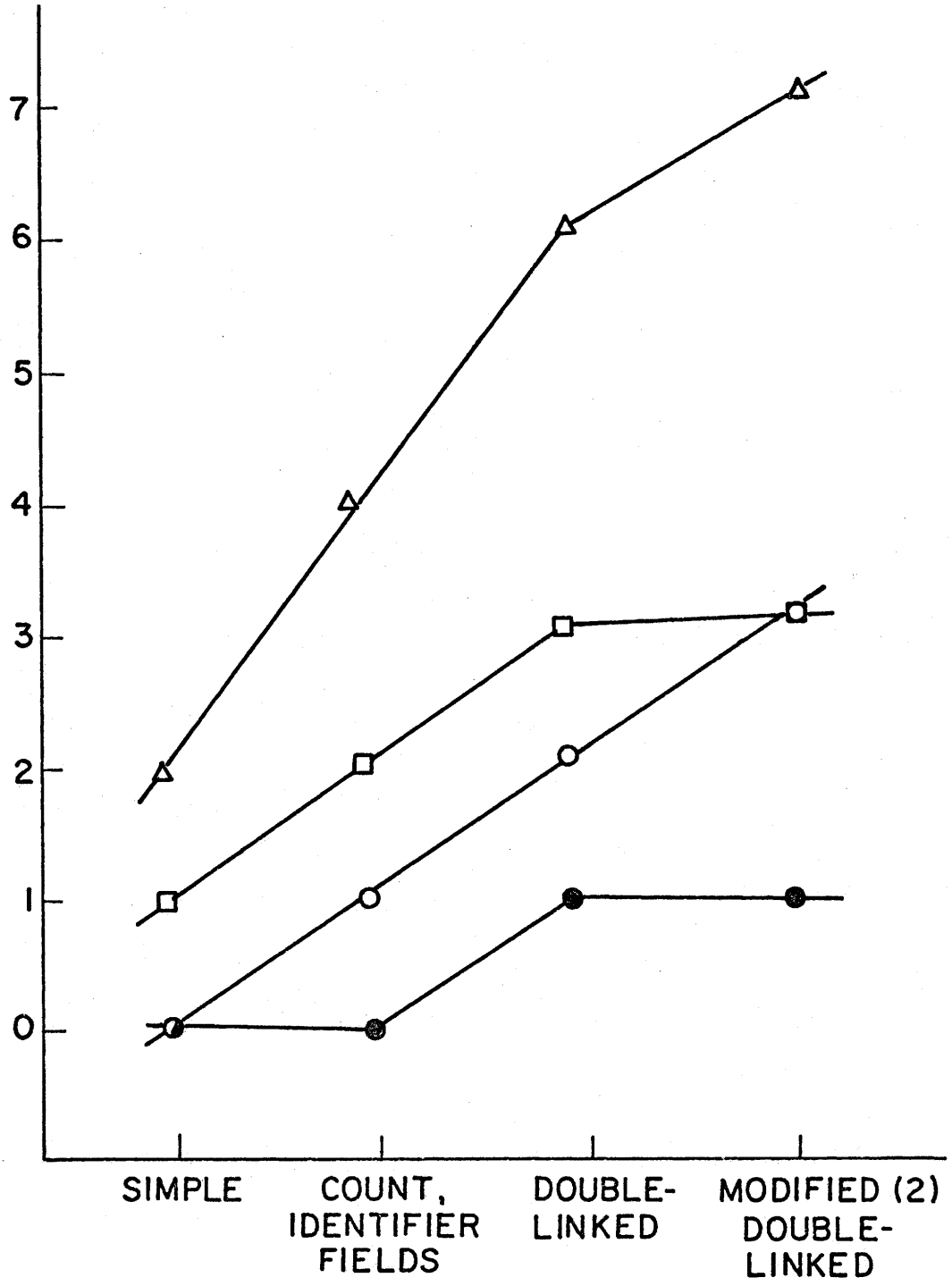

Figure 3.2 Cost and Effectiveness Graph for Linear List Implementations

- □ storage cost (words)
- ○ detectability
- ● correctability
- △ execution time for insert (changes)

Abstract

The increasing cost of computer system failure has stimulated interest in improving software reliability. One way to do this is by adding redundant structural data to data structures. Such redundancy can be used to detect and correct (structural) errors in instances of a data structure. The intuitive approach of this paper, which makes heavy use of examples, is complemented by the more formal development of the companion paper, "Redundancy in Data Structures: Some Theoretical Results".

Key Words and Phrases: Software reliability, software fault tolerance, robust data structures, redundancy, error detection, error correction, linear lists, binary trees.

else is required to perform the error detection and correction?

In-line checks may be introduced into normal system code to perform error detection, and possibly correction, during regular operation. This introduces an unvarying amount of overhead. Alternatively, detection/correction programs (sometimes called "audit" programs [1]) may be run periodically, or when trouble is suspected. The advantage is that their frequency may be varied according to criteria such as frequency of errors, system load, application criticality, etc. The classical example is automated telephone switching [1]. Some additional considerations may be found in [5, 27].

In order to indicate that the cost of detecting and correcting errors is not prohibitive, Figure 3.3 shows a single error correction/double error detection procedure for double-linked lists. In Part II, we give a General Correction Theorem, and exhibit a general correction procedure which has a polynomial, although rather excessive execution time. However, the procedure of Figure 3.3 has O(n) execution time for an n-node list. The procedure scans the list in the forward direction until an identifier field error or forward/back pointer mismatch is detected. When this occurs, a reverse scan is initiated until a similar error is encountered, at which point repair is attempted. Figure 3.4 gives an example, showing the applicable

```
procedure LIST-CORR(H, N)
begin
    pointer H, integer N, pointer P,
    pointer PREV-P, integer J;
    J <- 0;
    PREV-P <- H;
    P <- FORWARD(H);
    while (P ≠ H) do
    begin
        J <- J + 1;
        if (BACK(P) = PREV-P and ID(P) correct) then
        begin
            PREV-P <- P;
            P <- FORWARD(P);
        end
        else
        begin
            BACK-SCAN(H, P, PREV-P);
            return;
        end
    end
    if (BACK(H) ≠ PREV-P or ID(H) incorrect) then
    begin
        BACK-SCAN(H, P, PREV-P);
        return;
    end
    if (J ≠ N) then
        N <- J;
end

procedure BACK-SCAN(H, P, PREV-P)
begin
    pointer H, pointer P, pointer PREV-P,
    pointer Q, pointer PREV-Q;
    PREV-Q <- H;
    Q <- BACK(H);
    repeat
    begin
        if (FORWARD(Q) = PREV-Q and ID(Q) correct) then
        begin
            PREV-Q <- Q;
            Q <- BACK(Q);
        end
        else
        begin
            L-REPAIR(P, PREV-P, Q, PREV-Q);
            return;
        end
    end
end
```

```
procedure L-REPAIR(P, PREV-P, Q, PREV-Q)
begin
    pointer P, pointer PREV-P, pointer Q, pointer PREV-Q;
    if (P = Q and ID(P) incorrect) then
        ID(P) <- correct i.d.
    else
        if (P = PREV-Q) then
            BACK(PREV-Q) <- PREV-P
        else
            if (PREV-P = Q) then
                FORWARD(PREV-P) <- PREV-Q
            else
                "multiple error";
end
```

Figure 3.3 Linear list correction procedure

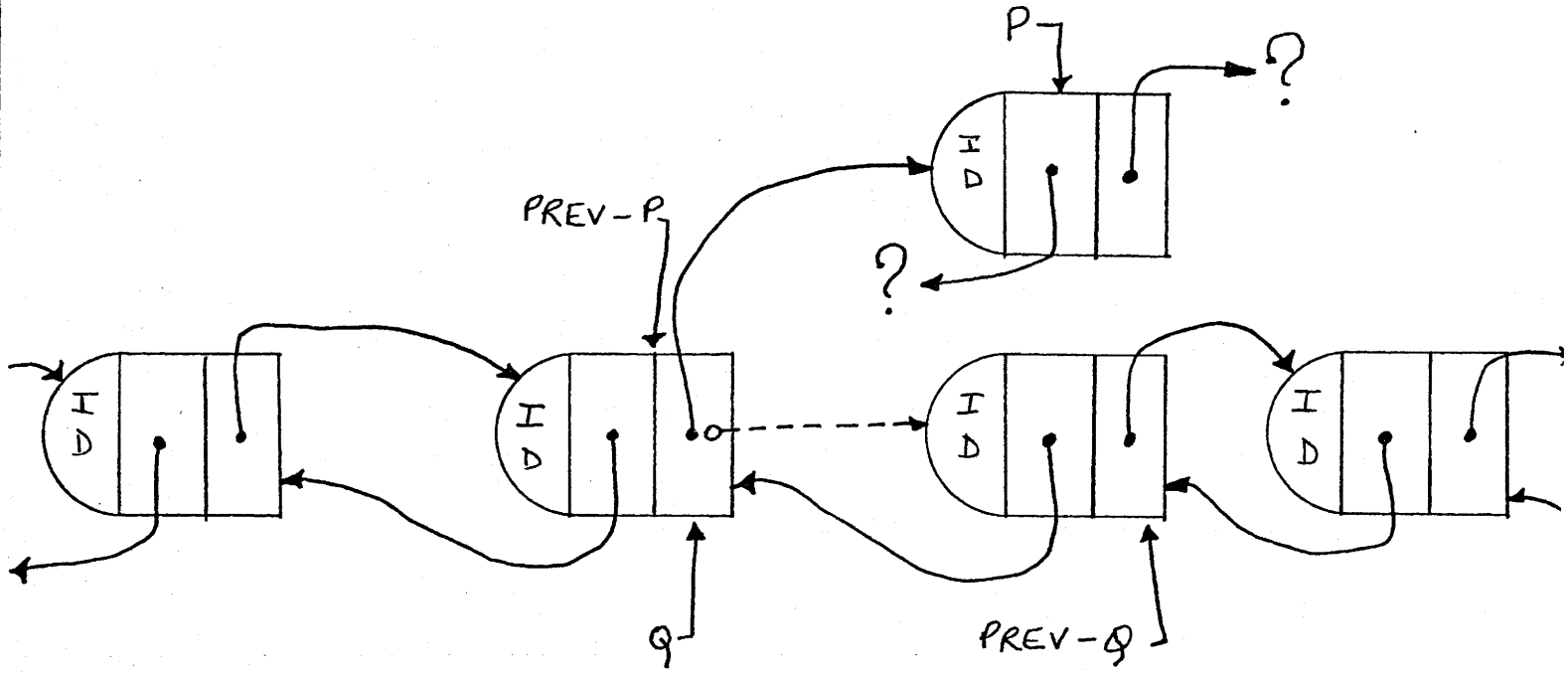

Figure 3.4 Linear List Correction. L-Repair will make the Correction Indicated by o---→

variables when L_REPAIR is called. The reader should be able to convince himself that the procedure corrects any single error to count, identifier, or pointer fields, and detects all double errors and some multiple errors. (This is proven in [24].) While LIST_CORR is shown in the form of an audit, it could easily be adapted to perform any "normal" processing required during list traversal, thus making it into an in-line check procedure.

3.2 Binary Trees

Binary trees are very commonly-used data structures, but ad hoc detection and correction techniques for them are not as well developed as for linear lists. This section presents new techniques for achieving the same level of robustness in binary trees as is provided by the common techniques used for linear lists.

The usual storage structure for binary trees will be considered, namely one in which each node of the tree contains two pointers, one to its left son and one to its right son. If either son does not exist, the corresponding pointer will have a null value. Procedures for traversing binary trees form the basis for detection and correction procedures. Generally, an in-order traversal will be used. In-order may be defined simply by: traverse the left subtree (in in-order), "visit" the root, traverse the right subtree (in in-order). This suggests an obvious recursive implementation; a simple non-recursive implementation using

a stack is also possible. For more details on tree traversal see [14, pp315-332].

Two obvious kinds of redundancy to add to a binary tree storage structure are identifier fields and a count of the number of nodes in the tree. As with linear lists, this yields 1-detectability and 0-correctability. Without this redundancy, the structure is 0-detectable and 0-correctable. Performing change detection is difficult because of problems associated with detecting the change of a pointer so that it points to a different subtree of the same size. It appears that all detection procedures which do not modify the tree structure instance require O(n log n) time and O(n) working storage, for a tree of n nodes. (These are worst case results; better average case behaviour could be obtained.)

A simple example will illustrate the source of the difficulty. Consider a three node tree: a root A and two sons of A, denoted B and C. Suppose the pointer from A to C is changed to point to B. The only difference from the original tree is that one node, B, now appears to be in two different locations in the tree. If modification is allowed we may set a flag in each node visited to detect duplication. However, if the tree may not be modified, we must compare each node against all other nodes in order to detect this type of change. The only effective way of detecting duplication is to store all node addresses in a structure which has O(log n) search and insertion times,

thus producing the results cited above. An alternative is to re-scan the previous part of the tree structure, thus eliminating the O(n) storage space but increasing the execution time to $O(n^2)$. It is only necessary to test for duplication at leaf nodes; however, since in a balanced tree of n nodes there are approximately n/2 leaves, the above order notations are not changed.

Another kind of redundancy which is sometimes added to binary trees to improve efficiency is a "thread link" [14, pp319-320]. In the case of "right threading," which will be used here, each null right link is replaced by a pointer to the in-order successor of the node containing the thread link. A recursive characterization is that, for a node, X, the final in-order node in X's left subtree contains a thread to X. A flag in each node is used to indicate whether the right pointer is a normal link or a thread. We assume that the encoding packs the right pointer and its associated flag into one word. As shown in [24, Section 5.4], this implementation is 1-detectable and a detection procedure exists which, for a tree of n nodes, requires O(n) time and space proportional to the height of the tree (O(log n) for a balanced tree).

The threaded tree implementation is not 2-detectable, as illustrated in Figure 3.5. The instance on the right differs in the count and one link from the instance on the left, and both are properly threaded binary trees.

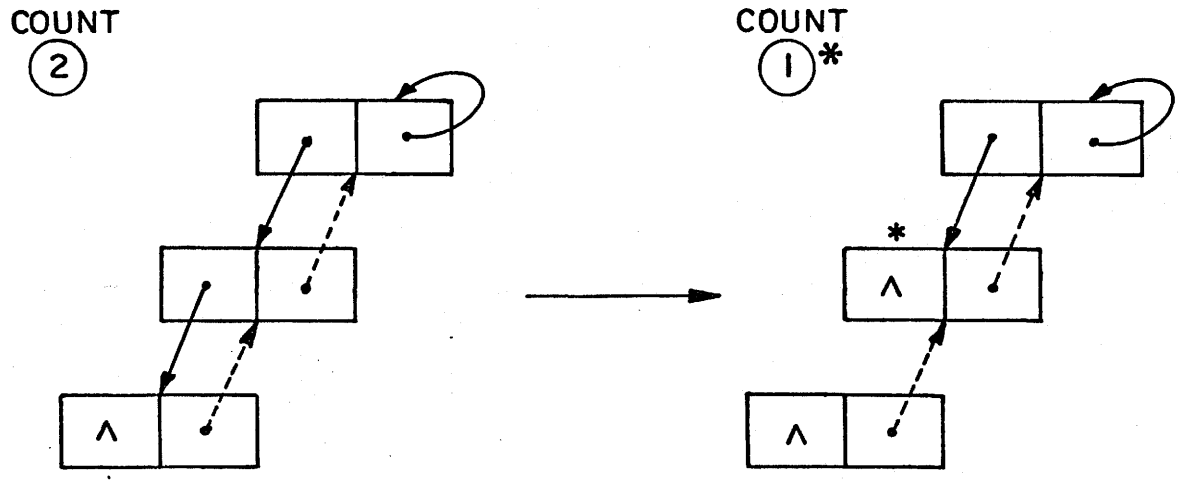

Figure 3.5 An Undetectable Pair of Changes to a Threaded Tree

We would like to obtain a 2-detectable, 1-correctable storage structure for a binary tree. As mentioned above for linear lists, the General Correction Theorem of Part II requires at least two edge-disjoint paths to each node of an instance, in order to prove that the structure is 1-correctable. Intuitively, this precludes disconnection of one or more nodes from the instance by a single change. This clearly implies that there be at least two pointers to each node, a condition which does not hold for threaded trees. We note that each node has exactly one non-thread link pointing to it and either zero or one thread links pointing to it. In fact, a node has a thread link pointing to it iff it has a non-null left subtree. (If the left subtree is non-null, the final in-order node in the subtree contains a thread to the node in question.) Thus, nodes with null left links have only one incoming edge, so an obvious possibility is to link these nodes together, using the left link field. A tag must be added to each node indicating the use of the left link, and the list head must now contain a pointer to the "first" node with a null left link. The nodes could be linked in any order, but for obvious reasons, in-order will be most convenient. (We again assume that pointer and flag are packed into a single word.)

This structure will be called a chained and threaded binary tree (CT-tree). The nodes with logically null left

links, joined in in-order, will be called the chain, and the links joining them will be called chain links. Figure 3.6 shows an example of a chained and threaded binary tree.

Figure 3.7 is a detection procedure for chained and threaded binary trees. Its execution time is O(n) for an n-node tree. The routine CHECK operates as follows. Given the header of a purported subtree, T, and a pointer CH to the first node which should be on T's chain, CHECK sets CH to the last in-order chain link in the subtree, and sets TH to the thread link of the final in-order node in the subtree. Calls to CHECK use the new values of CH and TH to verify that the tree is properly formed. CHECK also counts each node encountered and terminates if this count exceeds N, the expected number of nodes in the tree. The enclosing routine CHECK-CT simply uses the header to initialise the call to CHECK, and to verify that the tree is properly chained and threaded to the header.

The detection procedure is presented to make more formal our definition of a chained and threaded binary tree. However, exhibiting a detection procedure implies nothing about the robustness of this storage structure, which we claim is 2-detectable and 1-correctable. Given the 2-detectability, and the two edge-disjoint paths to every node (one using only chains and threads, the other using the normal tree pointers), the General Correction Theorem states that the structure is 1-correctable.

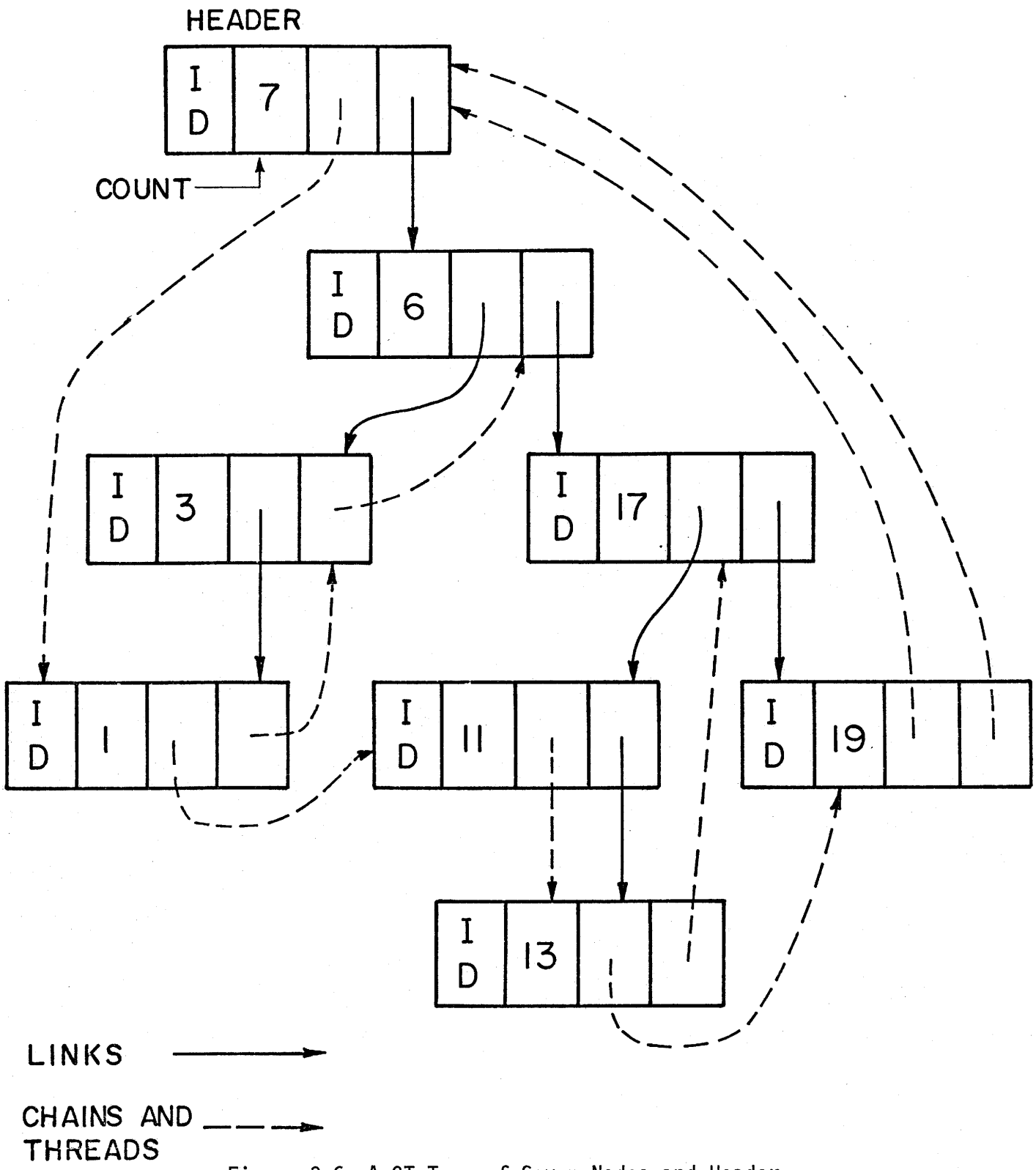

Figure 3.6 A CT-Tree of Seven Nodes and Header

```
procedure CHECK_CT(T, N);
    pointer T;
    integer N;
    begin

    /*  Given a purported tree root T, expected count N, and
    expected chain pointer CH, CHECK increments K by the
    number of nodes in the tree, and advances CH and TH to
    the outgoing chain and thread pointers.  It makes
    recursive calls to itself to check each of its subtrees.
    */

    procedure CHECK(T, N, K, CH, TH);
        pointer T, CH, TH;
        integer N, K;
        begin
        K <- K + 1;
        if ID(T) incorrect then error_exit;
        if K > N then error_exit;
        if LTAG(T) = 'CHAIN' then

            /* Check expected chain pointer and update CH.*/

            if CH = T then CH <- LEFT(T)
            else error_exit
        else begin
            CHECK(LEFT(T), N, K, CH, TH);

            /*Verify proper threading of subtree.*/

            if TH ~= T then error_exit;
            end;
        if RTAG(T) = 'THREAD' then

            /*Advance thread pointer.*/

            TH <- RIGHT(T)
        else CHECK(RIGHT(T), N, K, CH, TH);
        end CHECK;

    integer K;
    pointer CH, TH;

    /*Initialise call and verify subtree chain and thread.*/

    if T = null or ID(T) incorrect then error_exit;
    K <- 0;
    CH <- LEFT(T);
    CHECK(RIGHT(T), N, K, CH, TH);
    if N ~= K or CH ~= T or TH ~= T then error_exit;
    end CHECK_CT;
```

Figure 3.7. CT-tree detection procedure.

The following is an intuitive justification of the 2-detectability. Any set of changes which transforms one correct instance into another must either change the number of nodes in the instance, re-arrange existing nodes, or replace one or more nodes in the instance with foreign nodes.

a) Change in number. Deletion clearly requires fewer changes than insertion, as new identifier fields are required for new nodes under the valid state hypothesis. Any subtree may be deleted by changing the count, the incoming chain pointer, and either the left pointer to a chain or the right pointer to a thread, for a minimum of three changes. See Figure 3.8(a).

b) Re-arrangement. Clearly, both the normal tree structure and the chain/thread structure must be re-arranged for the result to be a correct CT-tree. The case requiring the fewest changes occurs when a null and a non-null subtree are exchanged, leaving the chain structure intact: the old thread becomes a right pointer, the incoming pointer to the subtree becomes a thread, and the outgoing thread of the subtree is updated, for a minimum of three changes. Updates which re-arrange non-null subtrees, which require changing the chain structure, or which re-arrange interior nodes all require a larger number of changes. See Figure 3.8(b).

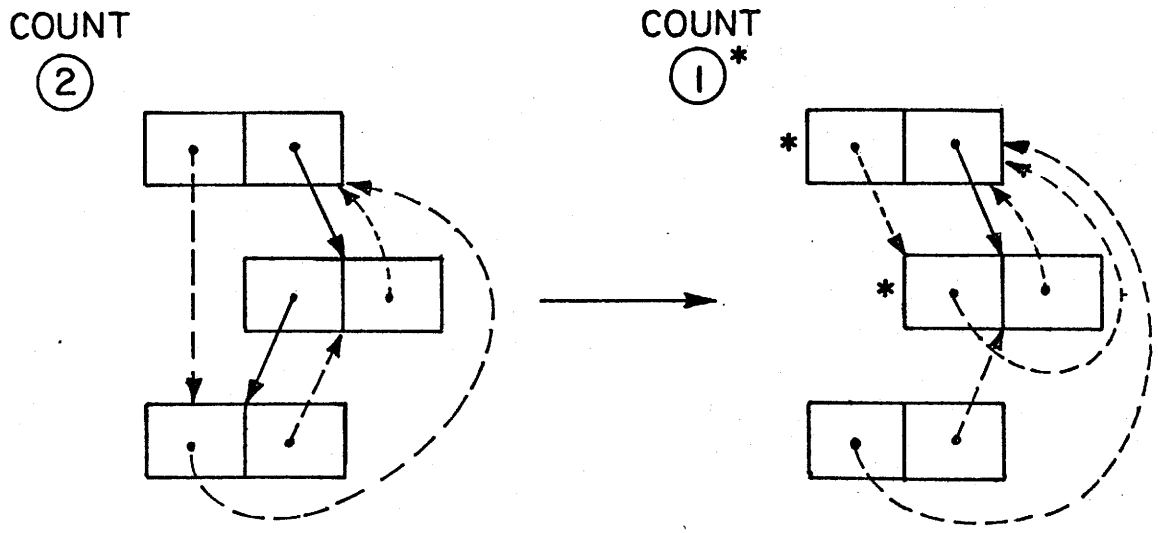

Figure 3.8(a) Change in Number of Nodes in a CT-Tree

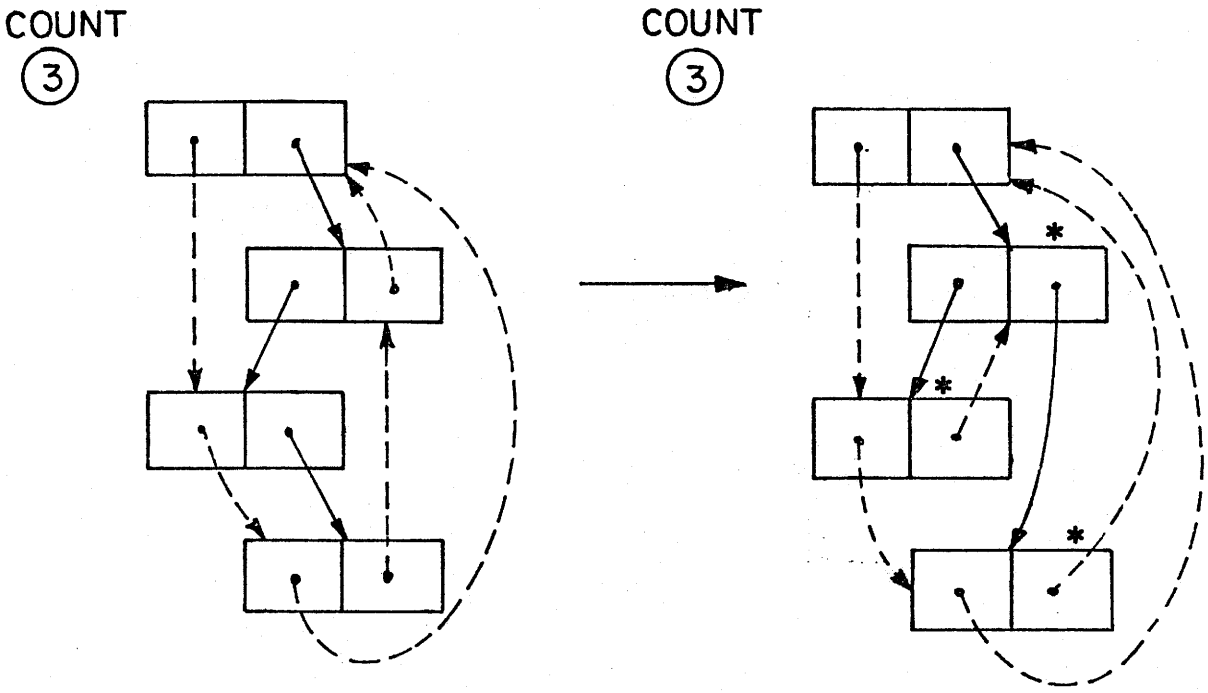

Figure 3.8(b) Re-arrangement of CT-Tree

c) Replacement. As the valid state hypothesis requires changing an identifier field for each replacement node, the minimum clearly occurs when a single foreign node replaces a single node of the instance. Besides the identifier field, we must change one incoming pointer, one incoming thread or chain, one outgoing chain or left pointer, and one outgoing thread or right pointer. This gives a minimum of five changes for replacing one or more nodes.

Thus, the minimum number of changes to transform one correct CT-tree into another is three for re-arrangement and change in number, and five for replacement. Note that such changes do not necessarily leave the system in a valid state, as they may leave identifier fields and pointers in memory external to the changed instance. However, this is undetectable except by exhaustive memory search, which we exclude by assumption. Since at least three changes are required to transform one correct CT-tree into another, the detectability is exactly two.

The results obtained here for binary tree storage structures are summarized in a cost and effectiveness graph (Figure 3.9). It should be noted that the simple storage structure with a count is 1-detectable but does not have a linear time, read-only detection procedure, whereas all the other detectabilities can be achieved by linear time procedures.

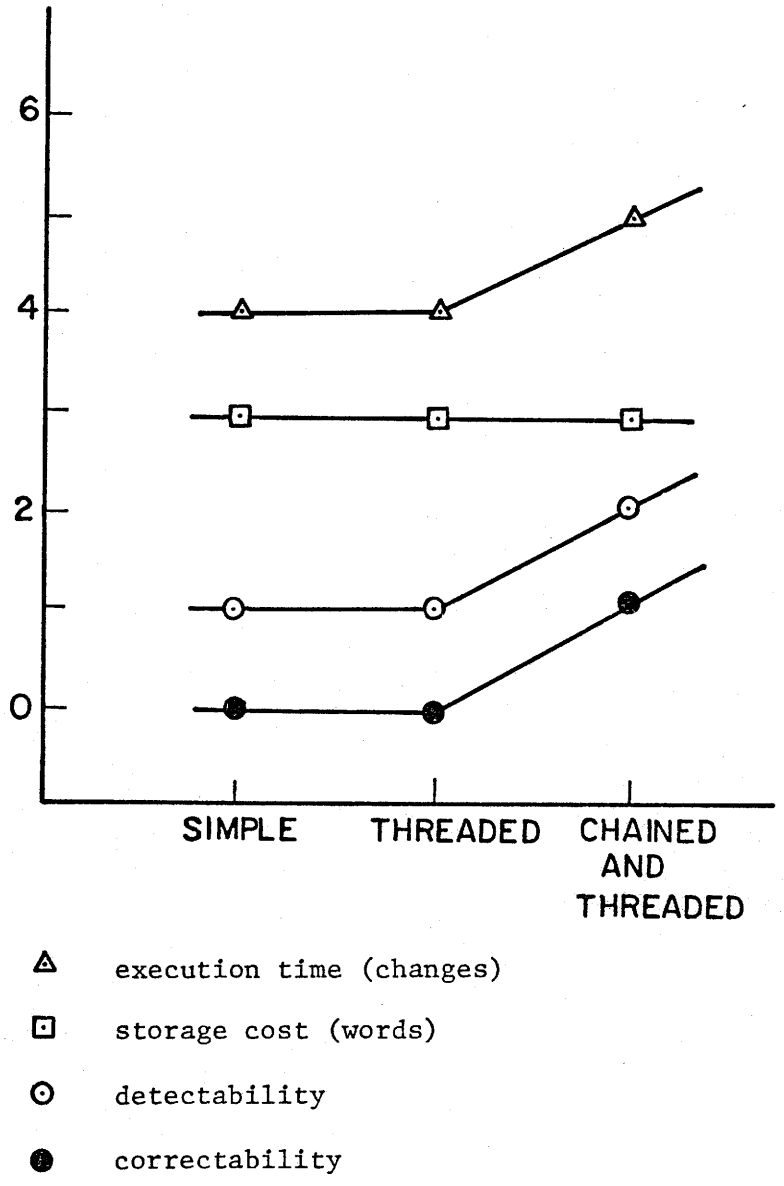

Figure 3.9 Cost and Effectiveness Graph for Binary Tree Implementations

4. EMPIRICAL RESULTS

The preceding section assumed an "intelligent" source of changes, that is, we calculated the minimum number of changes to produce an uncorrectable or undetectable error. In this section we discuss some experiments which were performed to determine the effect of applying "random" changes to the encoding of a data structure instance.

Although the change sources are different, the analytical results do partially predict the results of the experiments. For example, if a storage structure is exactly 2-detectable, we know that any randomly-selected change or pair of changes cannot produce an undetectable error, but that a set of three changes can. The experiments provide an indication of the probability that a set of three changes will produce an undetectable error.

It may be possible to calculate such probabilities directly from the specification of a storage structure, but at present only empirical results are available. (It should also be noted that the effect of applying random changes depends on various parameters which do not have to be considered when using an "intelligent adversary" model. How one selects "random changes" is clearly such a parameter. Less obviously, the size of the instance may also be a relevant parameter.)

4.1 Methodology

The basic experimental technique used was to introduce changes to encodings of data structure instances in a pseudo-random manner and observe the behaviour of detection and correction routines applied to the changed instances.

The experiments work with data structures that appear to be on external storage. The most important reason for this is that external data structures constrain a program to perform all accesses through read and write routines, simplifying the experiments.

The data structures are actually kept in main storage, a large buffer being used to simulate a random-access file. A set of routines called the "IOSYS Pseudo File System," developed in order to perform the experiments, provides support for such simulated files, and provides various auxiliary services such as long-term storage of simulated files on real external storage. An important facility provided is the "mangler" which allows pseudo-random changes to be inserted in a simulated file.

There are many ways of introducing erroneous changes in a data structure instance (i.e., mangling it) in order to test its robustness. Alternative methods of mangling range from inserting random values into randomly selected locations to making subtle changes to carefully selected locations in the instance. If no use is made of knowledge of the storage structure, subtle combinations of changes

that could be caused by software containing errors will occur with very small probability. If full knowledge of the data structure is used, it is likely that the mangler will only introduce those errors that the programmer thought of. A full discussion of manglers is beyond the scope of this paper.

The mangler used for these experiments is a compromise intended to minimize the disadvantages of either extreme. It is implemented as part of the write function of IOSYS. It pseudo-randomly chooses whether or not to change the record being written, which word to change, and by what amount to change the word. Small increments or decrements are used for changes rather than arbitrary replacement of a word, since the chosen method tends to introduce more "subtle" changes.

For flexibility, the mangler is driven by a set of user-specified parameters which determine: the probability of mangling a record, the probability density of changes over the words of a record, and the maximum value to be used as an increment or decrement. There are presently two distributions available: uniform, and skewed towards the beginning of the record (where structural information is typically stored). The increment to be used is chosen uniformly from the integers in the range -max to max, excluding zero. All parameters can be specified individually for the separate simulated files.

4.2 Detectability results

The purpose of the experiments was to estimate the probability of random changes producing undetectable errors in linear list storage structures. A routine "pretended" to delete records from a linear list by reading and writing those records which a delete routine would read and write. As records were written, words in the list nodes were "randomly" altered by adding or subtracting a small value. When a specified number of changes had been made, a detection procedure was executed to determine if the resulting instance could be detected as in error.

Three storage structures were tested in this experiment: the single-linked, double-linked, and modified(2) double-linked implementations described above. These have detectabilities of 1, 2, and 3, respectively, as described previously. For single-linked lists, 3000 sets each of one up to five changes were applied. For exactly two changes, five pairs of changes produced undetectable errors; no other number of changes produced undetectable errors. For both double-linked lists and modified(2) double-linked lists, 3000 sets of one to twelve changes were applied and no undetectable errors occurred.

Probably the most surprising aspect of these results is that single-linked lists seem more resistant to triples or quadruples of changes than to pairs of changes. It is hypothesized that this results from the tendency of sets of

more than two changes to include destruction of an identifier field in addition to an otherwise undetectable set of changes.

4.3 Correctability results

In order to study correctability, two additional concepts are needed. The first is called the accessible set of a data structure instance. It is the set of all nodes which can be accessed by following a sequence of pointers from the header of the structure. For a correct instance which does not contain pointers to other instances, the accessible set is simply the set of nodes which are (intuitively) "part of" the structure. We define the correctability radius to be one less than the minimum number of changes which can cause any node to become inaccessible.

No attempt was made to correct the instances found to be in error, but all the changed instances were checked to see if there was still a path to each node of the unchanged instance, which is a prerequisite for correction. We are particularly interested in determining how frequently disconnections occur once the correctability radius is exceeded. (In all the examples of Section 3, the correctability radius is equal to the correctability.) The following table shows the probabilities of disconnecting an instance (destroying all paths to a node):

Number of Changes	Single-linked	Double-linked	Modified(2) Double-linked
1	.424 (.406, .442)	0.00 (0.00, .001)	0.00 (0.00, .001)
2	.675 (.658, .692)	.143 (.131, .156)	.008 (.006, .012)
3	.841 (.828, .854)	.332 (.315, .349)	.020 (.015, .025)
4	.942 (.933, .950)	.510 (.492, .528)	.044 (.038, .052)
5	.978 (.972, .983)	.655 (.638, .672)	.079 (.070, .090)

(The parenthesized figures are 95% confidence intervals.)

We can observe that for the first two storage structures there is a direct practical significance for the correctability radius (which is 0 for single-linked lists and 1 for double-linked lists). If the correctability radius is exceeded we immediately encounter a significant number of disconnections, precluding correction. The modified(2) double-linked implementation also experiences some disconnections as soon as the correctability radius is exceeded, but there are not nearly as many. Another much more robust storage structure, not described here, was also tested. It has a correctability radius of four, but in the experiment no disconnections were observed for sets of fewer than fourteen changes, and even with as many as twenty

changes applied, the number of disconnections was very small, not exceeding seven disconnections in 3000 trials in any of the test runs.

5. CONCLUSIONS

In the first of these two papers, we have introduced the reader to concepts of robust data structures, and have given an informal analysis of the detectability and correctability of various implementations of linear lists and binary trees. Empirical results indicate that the effective detectability of a storage structure can be higher than that which is analytically shown to be possible.

We have seen that commonly-used techniques, in the case of linear lists, can be quite effective. However, the modified(2) double-linked implementation suggests that the commonly-used techniques may not necessarily be the best way of exploiting redundancy.

For binary trees, the authors are aware of no commonly-used storage structures which are 1-correctable. The chained and threaded implementation described here is 1-correctable, uses no additional storage (assuming space is available for tag bits), and can still be updated in time proportional to the height of the tree.

There are two potential problems with highly redundant structures which we have not discussed. One is simply that the increased complexity of the update routines may make

programming errors more likely. The other is the propagation of erroneous changes by correct update routines. It appears that in many highly redundant structures the rate of error propagation is directly proportional to the detectability.

We have attempted to give an informal description of our approach to improving data structure robustness, including definitions and examples. While the examples were concerned only with two simple data structures, we present more generally applicable formal results in Part II. The second paper also extends the basic framework to a restricted class of "compound" data structures, and discusses some of the design issues related to robust data structure synthesis.

BIBLIOGRAPHY

1. Almquist, R. P., J. R. Hagerman, R. J. Hass, R. W. Peterson, and S. L. Stevens. Software protection in No. 1 ESS. Proceedings of the International Switching Symposium, 1972. pp565-569.

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

3. Avizienis, Algirdas. Fault-tolerance: The survival attribute of digital systems. Proceedings of the IEEE, vol. 66, no. 10 (October 1978). pp1109-1125.

4. 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.

5. Chang, H. Y. Hardware maintainability and software reliability of electronic switching systems. Infotech State of the Art Report 20: Computer Systems Reliability, 1974. pp455-479.

6. Dearnley, P. A. An investigation into database resilience. Computer Journal, vol. 19, no. 2 (May 1976). pp117-121.

7. Denning, Peter J. Fault-tolerant operating systems. Computing Surveys, vol. 8, no. 4 (December 1976). pp359-389.

8. Downing, R. W., J. S. Nowak, and L. S. Tuomenoksa. No. 1 ESS maintenance plan. Bell System Technical Journal, vol. 43 (September 1964). pp1961-2019.

9. Fry, James P. and Edgar H. Sibley. Evolution of data-base management systems. Computing Surveys, vol. 8, no. 1 (March 1976). pp7-42.

10. Giannotti, Gene. Data base integrity. Data Management, vol. 12, no. 5 (May 1974). pp22-25.

11. Gotlieb, C. C. and F. W. Tompa. Choosing a storage schema. Acta Informatica, vol. 3 (1974). pp297-319.

12. Hammer, Michael M. and Dennis J. McLeod. Semantic integrity in a relational data base system. Proceedings of the International Conference on Very

Large Data Bases. Framingham, Massachusetts, Sept. 22-24, 1975. pp25-47.

13. Hamming, R. W. Error detecting and error correcting codes. Bell System Technical Journal, vol. 26, no. 2 (April 1950). pp147-160.

14. Knuth, Donald E. The Art of Computer Programming, volume 1: Fundamental Algorithms, Second edition. Addison-Wesley, 1973.

15. Lockemann, Peter C. and W. Dale Knutsen. Recovery of disk contents after system failure. Communications of the ACM, vol. 11, no. 8 (August 1968). p542.

16. Lorie, Raymond A. Physical integrity in a large segmented database. ACM Transactions on Database Systems, vol. 2, no. 1 (March 1977). pp91-104.

17. McLeod, Dennis J. High level expression of semantic integrity specifications in a relational data base system. Masters thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1976. MIT/LCS/TR-165.

18. Melliar-Smith, P. M. and B. Randell. Software reliability: the role of programmed exception handling. Proceedings of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, March 28-30, 1977. (Published as SIGPLAN Notices, vol. 12, no. 3, March 1977.) pp95-100.

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

20. Randell, Brian. Operating systems: The problems of performance and reliability. Information Processing 71, Proceedings of IFIP Congress 71, Ljubljana, Yugoslavia, August 23-28, 1971. pp281-290.

21. Randell, Brian. System structure for software fault tolerance. International Conference on Reliable Software, 21-23 April, 1975, Los Angeles. Proceedings. pp437-449.

22. Scherr, A. L. The design of OS/VS2 Release 2. Proceedings of the National Computer Conference, 1973 (vol. 42). pp387-394.

23. Scherr, A. L. Functional structure of IBM virtual storage operating systems, part II: OS/VS2-2 concepts and philosophies. IBM Systems Journal, vol. 12, no. 4. pp382-400.

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

25. Taylor, David J. Theoretical foundations for robust data structure implementations. Submitted to Journal of the ACM. Also available as Computer Science Research Report, CS-78-52, University of Waterloo, Waterloo, Ontario, Canada.

26. 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.

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

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

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO
TECHNICAL REPORTS 1979

Report No.	Author	Title
CS-79-01*	E.A. Ashcroft W.W. Wadge	Generality Considered Harmful - A Critique of Descriptive Semantics
CS-79-02*	T.S.E. Maibaum	Abstract Data Types and a Semantics for the ANSI/SPARC Architecture
CS-79-03*	D.R. McIntyre	A Maximum Column Partition for Sparse Positive Definite Linear Systems Ordered by the Minimum Degree Ordering Algorithm
CS-79-04*	K. Culik II A. Salomaa	Test Sets and Checking Words for Homomorphism Equivalence
CS-79-05*	T.S.E. Maibaum	The Semantics of Sharing in Parallel Processing
CS-79-06*	C.J. Colbourn K.S. Booth	Linear Time Automorphism Algorithms for Trees, Interval Graphs, and Planar Graphs
CS-79-07*	K. Culik, II N.D. Diamond	A Homomorphic Characterization of Time and Space Complexity Classes of Languages
CS-79-08*	M.R. Levy T.S.E. Maibaum	Continuous Data Types
CS-79-09	K.O. Geddes	Non-Truncated Power Series Solution of Linear ODE's in ALTRAN
CS-79-10*	D.J. Taylor J.P. Black D.E. Morgan	Robust Implementations of Compound Data Structures
CS-79-11*	G.H. Gonnet	Open Addressing Hashing with Unequal-Probability Keys
CS-79-12	M.O. Afolabi	The Design and Implementation of a Package for Symbolic Series Solution of Ordinary Differential Equations
CS-79-13*	W.M. Chan J.A. George	A Linear Time Implementation of the Reverse Cuthill-McKee Algorithm
CS-79-14	D.E. Morgan	Analysis of Closed Queueing Networks with Periodic Servers
CS-79-15*	M.H. van Emden G.J. de Lucena	Predicate Logic as a Language for Parallel Programming
CS-79-16*	J. Karhumäki I. Simon	A Note on Elementary Homorphisms and the Regularity of Equality Sets
CS-79-17*	K. Culik II J. Karhumäki	On the Equality Sets for Homomorphisms on Free Monoids with two Generators
CS-79-18*	F.E. Fich	Languages of R-Trivial and Related Monoids

* Out of print - contact author

CS-79-19*	D.R. Cheriton	Multi-Process Structuring and the Thoth Operating System
CS-79-20*	E.A. Ashcroft W.W. Wadge	A Logical Programming Language
CS-79-21*	E.A. Ashcroft W.W. Wadge	Structured LUCID
CS-79-22	G.B. Bonkowski W.M. Gentleman M.A. Malcolm	Porting the Zed Compiler
CS-79-23*	K.L. Clark M.H. van Emden	Consequence Verification of Flow-charts
CS-79-24*	D. Dobkin J.I. Munro	Optimal Time Minimal Space Selection Algorithms
CS-79-25*	P.R.F. Cunha C.J. Lucena T.S.E. Maibaum	On the Design and Specification of Message Oriented Programs
CS-79-26*	T.S.E. Maibaum	Non-Termination, Implicit Definitions and Abstract Data Types
CS-79-27*	D. Dobkin J.I. Munro	Determining the Mode
CS-79-28	T.A. Cargill	A View of Source Text for Diversely Configurable Software
CS-79-29*	R.J. Ramirez F.W. Tompa J.I. Munro	Optimum Reorganization Points for Arbitrary Database Costs
CS-79-30	A. Pereda R.L. Carvalho C.J. Lucena T.S.E. Maibaum	Data Specification Methods
CS-79-31*	J.I. Munro H. Suwanda	Implicit Data Structures for Fast Search and Update
CS-79-32*	D. Rotem J. Urrutia	Circular Permutation Graphs
CS-79-33*	M.S. Brader	PHOTON/532/Set - A Text Formatter
CS-79-34*	D.J. Taylor D.E. Morgan J.P. Black	Redundancy in Data Structures: Improving Software Fault Tolerance
CS-79-35	D.J. Taylor D.E. Morgan J.P. Black	Redundancy in Data Structures: Some Theoretical Results
CS-79-36	J.C. Beatty	On the Relationship between the LL(1) and LR(1) Grammars
CS-79-37	E.A. Ashcroft W.W. Wadge	R_x for Semantics

* Out of print - contact author

CS-79-38	E.A. Ashcroft W.W. Wadge	Some Common Misconceptions about LUCID
CS-79-39	J. Albert K. Culik II	Test Sets for Homomorphism Equivalence on Context Free Languages
CS-79-40	F.W. Tompa R.J. Ramirez	Selection of Efficient Storage Structures
CS-79-41*	P.T. Cox T. Pietrzykowski	Deduction Plans: A Basis for Intelligent Backtracking
CS-79-42	R.C. Read D. Rotem J. Urrutia	Orientations of Circle Graphs

* Out of print - contact author

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO

RESEARCH REPORTS 1980

Report No.	Author	Title
CS-80-01	P.T. Cox T. Pietrzykowski	On Reverse Skolemization
CS-80-02	K. Culik II	Homomorphisms: Decidability, Equality and Test Sets
CS-80-03	J. Brzozowski	Open Problems About Regular Languages
CS-80-04	H. Suwanda	Implicit Data Structures for the Dictionary Problem
CS-80-05	M.H. van Emden	Chess-Endgame Advice: A Case Study in Computer Utilization of Knowledge
CS-80-06	Y. Kobuchi K. Culik II	Simulation Relation of Dynamical Systems
CS-80-07	G.H. Gonnet J.I. Munro H. Suwanda	Exegesis of Self-Organizing Linear Search
CS-80-08	J.P. Black D.J. Taylor D.E. Morgan	An Introduction to Robust Data Structures
CS-80-09	J.Ll. Morris	The Extrapolation of First Order Methods for Parabolic Partial Differential Equations II
CS-80-10*	N. Santoro H. Suwanda	Entropy of the Self-Organizing Linear Lists
CS-80-11	T.S.E. Maibaum C.S. dos Santos A.L. Furtado	A Uniform Logical Treatment of Queries and Updates
CS-80-12	K.R. Apt M.H. van Emden	Contributions to the Theory of Logic Programming
CS-80-13	J.A. George M.T. Heath	Solution of Sparse Linear Least Squares Problems Using Givens Rotations
CS-80-14	T.S.E. Maibaum	Data Base Instances, Abstract Data Types and Data Base Specification
CS-80-15	J.P. Black D.J. Taylor D.E. Morgan	A Robust B-Tree Implementation
CS-80-16	K.O. Geddes	Block Structure in the Chebyshev-Padé Table
CS-80-17	P. Calamai A.R. Conn	A Stable Algorithm for Solving the Multi-facility Location Problem Involving Euclidean Distances

* In preparation

CS-80-18	R.J. Ramirez	Efficient Algorithms for Selecting Efficient Data Storage Structures
CS-80-19	D. Therien	Classification of Regular Languages by Congruences
CS-80-20	J. Buccino	A Reliable Typesetting System for Waterloo
CS-80-21	N. Santoro	Efficient Abstract Implementations for Relational Data Structures
CS-80-22	R.L. de Carvalho T.S.E. Maibaum T.H.C. Pequeno A.A. Pereda P.A.S. Veloso	A Model Theoretic Approach to the Theory of Abstract Data Types and Data Structures
CS-80-23	G.H. Gonnet	A Handbook on Algorithms and Data Structures
CS-80-24	J.P. Black D.J. Taylor D.E. Morgan	A Case Study in Fault Tolerant Software
CS-80-25	N. Santoro	Four O(n**2) Multiplication Methods for Sparse and Dense Boolean Matrices
CS-80-26	J.A. Brzozowski	Development in the Theory of Regular Languages
CS-80-27	J. Bradford T. Pietrzykowski	The Eta Interface
CS-80-28	P. Cunha T.S.E. Maibaum	Resource = Abstract Data Type Data + Synchronization ...
CS-80-29	K. Culik II Arto Salomaa	On Infinite Words Obtained by Interating Morphisms
CS-80-30	T.F. Coleman A.R. Conn	Nonlinear Programming via an Exact Penalty Function: Asymptotic Analysis
CS-80-31*	T.F. Coleman A.R. Conn	Nonlinear Programming via an Exact Penalty Function: Global Analysis
CS-80-32	P.R.F. Cunha C.J. Lucena T.S.E. Maibaum	Message Oriented Programming - A Resource Based Methodology
CS-80-33	Karel Culik II Tero Harju	Dominoes Over A Free Monoid
CS-80-34*	K.S. Booth	Dominating Sets in Chordal Graphs
CS-80-35*	Alan George J. W-H Liu	Finding Diagonal Block Envelopes of Triangular Factors of Partitioned Matrices

* In preparation