DETECTABILITY AND CORRECTABILITY OF DATA STRUCTURE IMPLEMENTATIONS

David J. Taylor
David E. Morgan

Research Report CS-78-51

Department of Computer Science

University of Waterloo
Waterloo, Ontario, Canada

November 1978

Key Words and Phrases: data structures, data representation, robust systems, redundant encoding, fault tolerance, error detection, error correction, linear lists, binary trees

CR Categories: 4.34, 4.6

ABSTRACT

This paper describes one method for enhancing software fault tolerance, the implementation of robust data structures. A robust data structure implementation is one containing redundancy which can be used to detect and correct erroneous changes made to an instance of the data structure. Three commonly used forms of redundancy are considered here: storing a count of the number of nodes in a structure instance, using node type identifier fields, and using additional pointers.

Some basic concepts are defined which allow data structure robustness to be studied, and some general results are presented which allow the detection and correction properties of implementations to be determined. Four possible implementations of linear lists are examined to determine their robustness and the costs of the robustness.

Analytic techniques are used to establish the levels of detectability and correctability that are possible for a particular implementation. Empirical results indicate that the effective detectability of an implementation can be higher than that which is analytically shown to be possible. This is true because the combinations of changes that are required to produce an undetectable error rarely occur in practice, but must be considered in a proof. Thus, one implementation of a linear list that is 2-detectable in theory appears to be better than 12-detectable in practice.

However, an implementation which is 1-detectable in theory also appears to be 1-detectable in practice.

1. INTRODUCTION

Software systems encounter faults from a number of sources: design flaws, program bugs, hardware malfunctions, and incorrect user actions. A desirable property of software systems is "fault tolerance," the ability to continue to provide services in spite of such faults. For some real-time applications, fault-tolerance may be an extremely important property.

In discussing fault-tolerance, we will use some definitions suggested by Melliar-Smith and Randell [3]. 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, and a potential fault is a construction which under some circumstances could produce an error. A fault tolerant system is one which attempts to prevent erroneous states from producing failures. This paper presents a specific aspect of fault tolerance: the identification and removal of errors in data structure instances.

The following definitions will be used in discussing data structures. A data structure is defined to be a logical organisation of data. We define a data structure implementation to be a representation, on some storage

medium, of a data structure. A data structure instance is a particular occurrence of a data structure implementation. 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 an implementation of a binary tree; and if a particular set of data is stored according to this implementation, that is a data structure instance.

A change is defined to be an elementary modification to a data structure instance. To illustrate this definition, consider the following implementation of 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. The effect of changes on the "data content" of an instance is also an important problem but is outside the scope of the present discussion.

Detection properties of a data structure implementation are stated in terms of changes. If a single change can

transform a correct data structure instance into another correct instance, as in this example, the implementation 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, N-1 change detection is possible.

If all sets of N or fewer changes can be detected, we say the implementation is N-detectable. Similarly, if all sets of N or fewer changes can be corrected, we say the implementation is N-correctable. These definitions of detectability and correctability are closely related to Hamming's definitions for binary codes [1]. (Note that N-detectability implies K-detectability for K < N, and similarly for correctability.)

A robust data structure implementation is an implementation containing redundant data which allows erroneous changes to be detected, and possibly corrected as well. Thus, the robustness of a data structure implementation is defined in terms of its detectability and correctability. The purposes of this paper are to provide ways of determining the robustness of implementations, and to suggest some ways of designing robust implementations.

The theme of this paper is that the fault-tolerance of a system can be enhanced by increasing the robustness of the data structure implementations it uses. Naturally,

fault-tolerance will be improved only if appropriate routines for performing detection and correction are also included.

Section 2 of this paper presents some useful results about the detectability and correctability of data structure implementations. The methods used to establish these results are sketched in this paper; complete proofs for these and other results can be found in [4]. Section 3 discusses the results of applying the theory to linear lists and binary trees. In Section 4, some interesting empirical results are presented that relate the robustness that is theoretically possible to the effective robustness that can be achieved in practice. The empirical techniques used are outlined. The final section draws some conclusions and outlines some areas needing further study.

2. DETECTABILITY AND CORRECTABILITY

2.1 Detectability

The purpose of this section is to provide means of determining upper and lower bounds on the detectability of a data structure implementation. The first result provides a means of determining an upper bound on detectability. Two techniques for finding lower bounds on detectability are developed. One allows detectability to be calculated directly. The other makes use of the intermediate properties ch-same and ch-diff (defined below). Section 2.3

contains another useful upper bound result.

The results will be illustrated by an example, the double-linked implementation of a simple list (Figure 2.1). In this example, the two lower bound techniques will provide the same result. In other cases, one technique will provide a greater lower bound than the other.

Theorem 1: If an implementation allows an "empty" instance, and n is the number of pointers in the list head which do not permanently point to a fixed location in the list head, and there are j stored counts, then the detectability of the implementation is at most n + j - 1.

(A fixed list head pointer which points to the list head may seem unlikely, but does occur in practice. For example, see the threaded tree implementation in [2, p322].)
Proof: The list head of an empty instance differs by at most n pointer changes from any other instance, so n pointer changes and j count changes can transform any instance to the empty instance. Thus the detectability is at most n + j - 1.■

Figure 2.1(a) shows a double-linked list implementation. Since the list head contains two pointers and there is one stored count, the "empty" list (2.1(b)) can be obtained from the other (2.1(a)) by three changes, thus showing that double-linked lists are at most 2-detectable, as proven by the theorem.

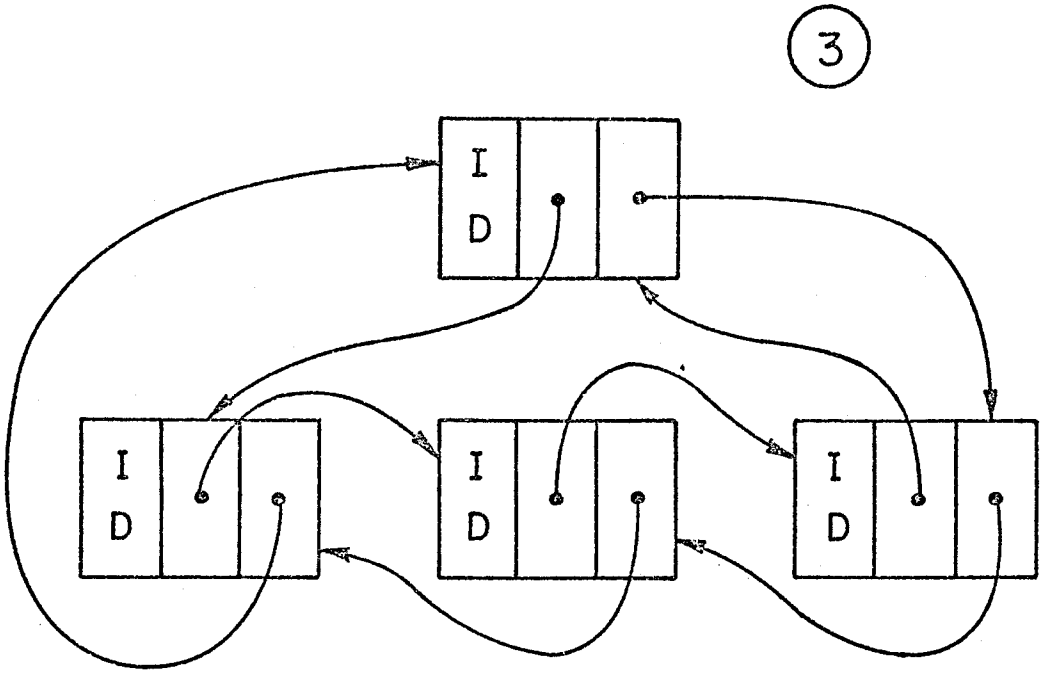


Figure 2.1(a)

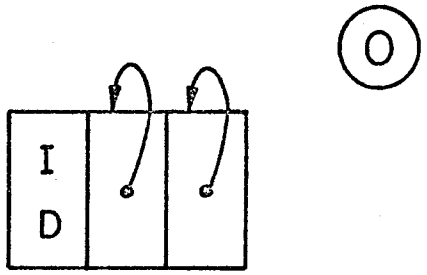


Figure 2.1(b) Double-Linked List Implementation

Several detectability results are related to the concepts k-determined and k-count-determined. These may be defined informally as follows. An implementation is k-determined if the pointers in each instance of the implementation can be partititioned into k disjoint sets, such that each set of pointers can be used to reconstruct all counts, identifier fields, and other pointers. (It must also be possible to determine which pointers are in a particular set without reference to any other pointers.) An implementation is k-count-determined if the pointers in each instance of the implementation can be partitioned into k disjoint sets, such that each set can be used to calculate the number of nodes in the instance.

These definitions can be used in stating a number of detectability results.

Theorem 2: A k-determined implementation is (k-1)-detectable.

Proof: We may detect errors in such an implementation by using each of the k sets of pointers to determine the values which all counts, identifier fields, and other pointers should have, and then comparing these with the actual values. If at most k-1 changes have been made, then one set of pointers contains no changes. Thus, when it is used to check the rest of the structural data, an error will be detected.■

Figure 2.2 shows the list of Figure 2.1(a) twice, once with the backward links and the count omitted, and once with the forward links and the count omitted. In either case, all omitted structural data can be determined from that remaining. Thus, double-linked lists are 2-determined, and by Theorem 2 are (at least) 1-detectable.

Theorem 3: If a k-determined implementation contains identifier fields and a stored count, and if one or more of the k sets of pointers contains only one pointer to each node, then the implementation is k-detectable.

Proof: The proof of the previous theorem can be used except in the case of exactly k changes, one to each of the k sets of pointers.

In this case, consider a set of pointers which has only one pointer to each node. In this set, one pointer has been changed, so the node it pointed to has disappeared. Thus either the stored count cannot agree with the actual number of nodes in the list or a "foreign" node has been added to the instance. (That is, an area of storage which is not a node now appears to be one.) In the latter case, each of the k sets of pointers would need to contain a changed pointer to the foreign node, and a change is also required to place a proper identifier field value in that node. This is a total of k+1 changes, so the implementation is k-detectable.■

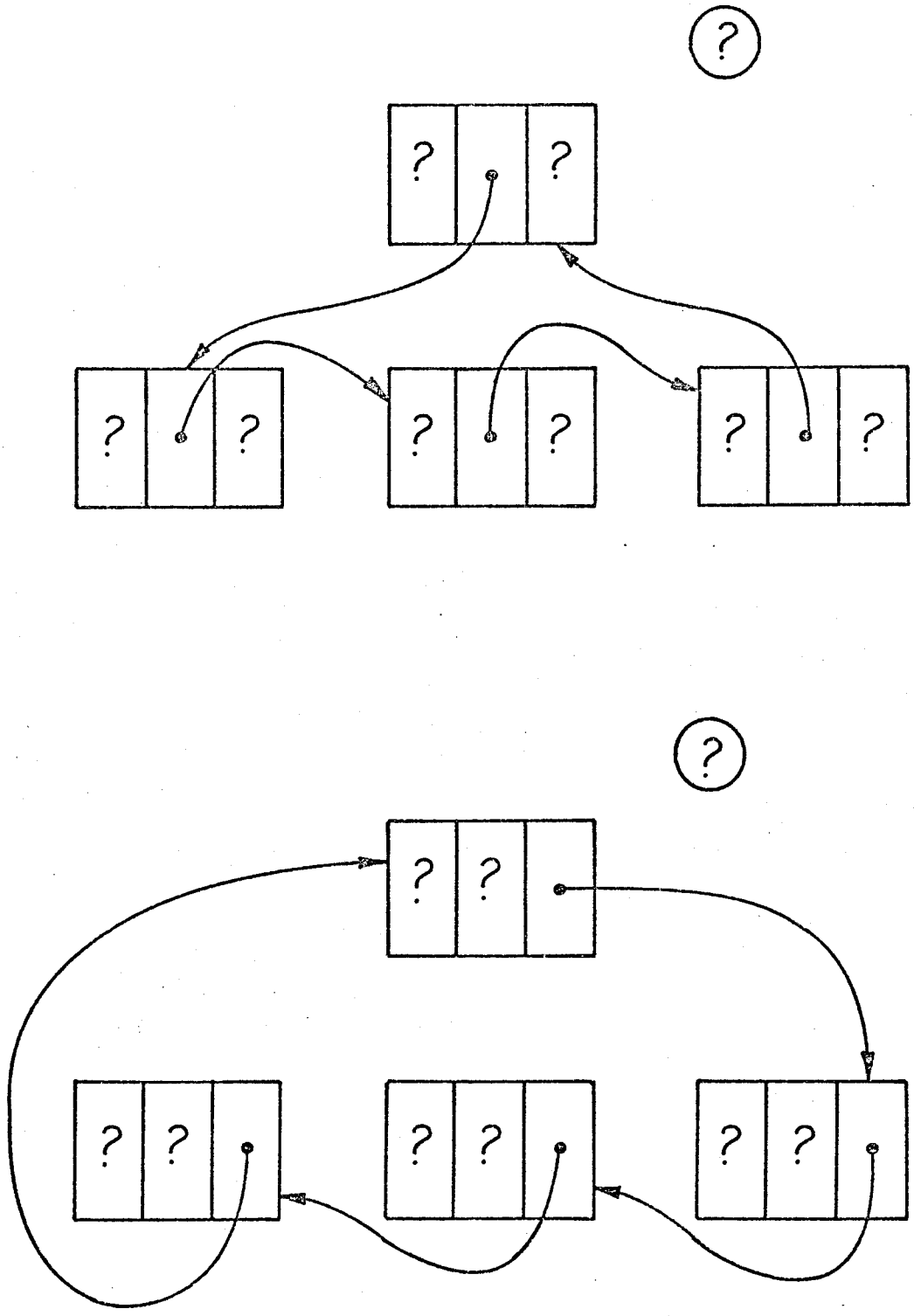


Figure 2.2 2-Determined Implementation

We conclude from Theorems 1 and 3 that double-linked lists are exactly 2-detectable.

Since the entire set of pointers in a structure instance determines the count (or counts) and all identifier fields, every structure is 1-determined. Thus we have the following as a simple corollary of Theorem 3:

Corollary: If an implementation uses identifier fields and a stored count, and there is only one pointer to each node, then the implementation is 1-detectable.

Two properties of a data structure implementation are now defined which provide an alternative method of determining lower bounds on detectability. The minimum number of changes that transforms a correct structure instance into another correct instance with the same number of nodes is defined to be ch-same. Similarly, ch-diff is defined to be the minimum number of changes that transforms a data structure instance into another correct instance with a different number of nodes.

Theorem 4: The detectability of an implementation is min(ch-same, ch-diff) - 1.

Proof: This result is very simple to prove. The minimum number of changes which transforms one correct instance into another is simply min(ch-same, ch-diff), and the detectability is defined to be one less than this value.■

Theorem 5: If each of the k sets of pointers in a k-determined implementation contains only one pointer to each node, if m of those sets have exactly one pointer in each node, and if there are a minimum of n identifier fields per node, then

$$ch\text{-}same \geq k + \min(n + m, k).$$

Proof: Consider an undetectable sequence of changes which leaves the number of nodes unchanged. There are two possibilities: the same set of nodes exists, differently structured, or one or more nodes have been replaced by "foreign" nodes. To insert a foreign node, we must change at least one pointer in each of the k sets and we must insert n identifier fields in the foreign node. For m of the k sets there must be an equal number of pointers entering and leaving any set of nodes, so there must be at least m pointers from foreign nodes to nodes in the unchanged structure. So, the minimum number of changes to insert one or more foreign nodes is k + n + m.

If no foreign nodes have been inserted, then there must be at least two changes in each of the k sets of pointers. If only one change is made, there are two cases: (1) The unchanged value was null and the changed value is non-null. In this case, some node must now have two pointers pointing to it, which violates the hypothesis of the theorem and can be detected. (2) The unchanged value was non-null. In this case the node formerly pointed to does not now have a

pointer to it in this set, which can be detected. (If both values are null, the pointer has not been changed.) Therefore at least 2k changes are required.

We thus have

$$\text{ch-same} \geq \min(k + n + m, 2k)$$
$$= k + \min(n + m, k).\blacksquare$$

It might seem that we could take m as the minimum number of non-null pointers in a node, thus increasing ch-same in some cases. For a counterexample to this, see [4, Example 4.4.4].

For double-linked lists, k = 2, m = 2, n = 1; thus ch-same $\geq$ 4. In fact, ch-same = 5 for double-linked lists, but a lengthy argument, from first principles, would be required to show this.

Theorem 6: If a k-count-determined implementation has j stored counts, ch-diff $\geq$ k + j.

Proof: If we change the number of nodes in an instance, we must change one pointer in each of the k sets of pointers and also each of the j stored counts.■

For double-linked lists k = 2, j = 1 so ch-diff $\geq$ 3. (In fact, ch-diff = 3.) Applying Theorem 4 to the values obtained for ch-same and ch-diff we conclude that double-linked lists are (at least) 2-detectable. This is independent of the previous method which showed the same result. Note that although the bound on ch-same is not

maximal, the resulting detectability is, since in this case ch-diff is the limiting factor. Very often, the detectability of an implementation may most easily be computed by first computing ch-same and ch-diff, then applying Theorem 3.

It is possible to give examples which show that the results of Theorems 2, 3, 5, and 6 are numerically maximal and that none of the hypotheses can be removed from these theorems.

2.2 Correctability

The following theorem allows the correctability of an implementation to be determined if the detectability is known, provided the implementation contains identifier fields.

Theorem 7: If a data structure implementation employing identifier fields is 2r-detectable and there are at least r+1 edge-disjoint paths to each node of the structure, then the implementation is r-correctable.

Proof: To prove this result we need an algorithm which can perform the indicated correction. The algorithm has two main phases: first, collection of all nodes in the data structure instance and, second, the restoration of the instance to a correct state.

The collection phase essentially consists of a depth-first search of the instance. Since pointers may have

been modified, this search may lead outside the instance, but checking of the identifier fields places a bound on the number of nodes which can be examined which are not part of the actual data structure instance. The algorithm terminates its scan along any path which includes more than r bad identifier fields.

Because there were originally r+1 paths to each node, this procedure must be able to find all nodes which were in the instance before it was changed. It may also "find" other nodes, but there is a bound on the number of such nodes it will locate.

Once a superset of the nodes has been found, correction can be performed on a trial and error basis. Sets of r or fewer changes are used; since the desired instance is the only correct instance this "close" to the given one, this procedure will always yield the desired result. Unfortunately, the execution time behaviour of this algorithm is very poor. Typically, to perform r-correction on an instance of n nodes will take time O(n**(2r+1)).■

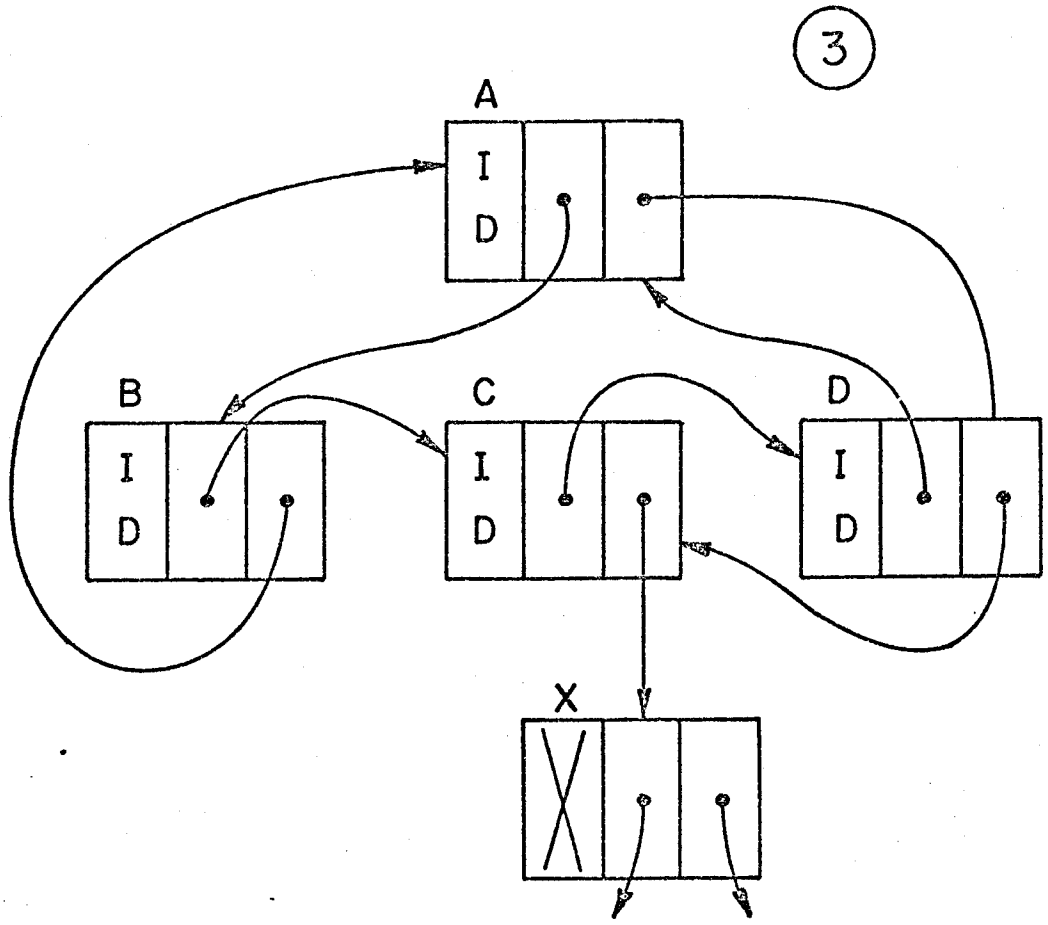
We have shown that double-linked lists are 2-detectable. We clearly have two edge-disjoint paths to each node: one using forward pointers and one using backward pointers. Thus the hypothesis of the theorem is satisfied for r equal to 1, and we conclude that double-linked lists are 1-correctable.

We may also use double-linked lists to illustrate the

operation of the correction algorithm. Figure 2.3 shows a double-linked list in which one pointer has been changed, so that it now points to an area of storage which is not a node of the list. The figure also shows the contents of NODE.TABLE (the set of nodes collected by the correction algorithm), which consists of node addresses and "bad identifier field" counts. In this case, "node" X, which is not part of the correct list, is the only one with a non-zero "bad identifier field" count. We cannot conclude that X must be removed--if the change had damaged an identifier field rather than a pointer, the appropriate correction would be to change the identifier field value. The correction procedure will try a number of corrections, passing each "corrected" instance to the detection routine, until one is accepted. It will attempt: setting ID(A) to the correct identifier value; setting the forward pointer in A to point to A itself; then setting this pointer to point to B, and so on until we reach "Back(C) = B". This will produce a correct instance (one accepted by the detection routine), so the correction procedure will terminate with this correction applied to the instance.

For a linear list of n nodes, this correction procedure will take time $O(n^3)$; however, it is also possible to write a special-purpose correction procedure for linear lists, which takes time $O(n)$.

This result allows correctability to be determined in



A	O
B	O
C	O
D	O
X	I

Figure 2.3 Operation of General Correction Procedure

most cases of practical interest. Unfortunately, no analogous result is known which allows detectability to be determined in general.

2.3 Costs and Effectiveness

The addition of redundancy to stored data combined with software to make effective use of that redundancy can make a system more fault-tolerant and will also likely affect performance. In some cases, the added redundant data will allow simplification of some processing, thus improving efficiency, but adding redundancy will usually degrade performance. Thus, a tradeoff typically exists between robustness and performance.

In the past, such tradeoffs have been made on an ad hoc basis, because there was no appropriate theoretical foundation for studying them. The purpose of this section is to elucidate the relationships between robustness and performance and to show that it is possible to establish a balance between them. It is suggested that proper choice of redundancy can yield a high effective degree of robustness at low cost.

The benefits and costs of a robust data structure implementation cannot be stated in absolute terms. They depend on the particular environment in which the data structure is to be used. In this section, we therefore provide only an outline of the costs and the effectiveness

of the techniques considered here. A graphical technique for displaying costs and effectiveness, which may help visualise the nature of parameterised implementations, is also developed.

The benefits are here considered only in terms of detectability and correctability, which are measures of the robustness of a data structure implementation. Benefits such as simplification of processing due to the presence of additional pointers will not be considered.

Various costs are associated with the robust implementations considered here. We identify three: additional storage requirements, increased processing time for insertions and deletions, and processing time to perform change detection. The final one will not be considered further. In practice, the execution time of a detection procedure always seems to be linear in the size of the structure instance being checked. ("Size" can generally be taken as the number of nodes in a structure instance. If nodes can vary in length with no fixed upper bound, "size" should be taken as the total storage area occupied by the instance.) The processing time used in checking can also be adjusted by varying the interval between executions of the detection procedure, but consideration must also be given to the fault rate.

The storage cost of an implementation can be considered to be made up of three parts: data content, structural

information, and redundancy. The redundant data used in implementations discussed here is largely of a structural nature. In many cases, it is an arbitrary decision as to what is structural information and what is redundancy. Thus, we consider the storage cost to be the number of words needed to store structural data (including redundant structural data), per node. Fixed storage costs, which do not vary with the number of nodes are generally of minor importance unless the "typical" structure instance is very small.

Because insertion and deletion are inverse operations, their costs are closely related. Thus we consider the cost of an insertion as being representative of both. For reasons similar to those discussed above, we will consider the cost of an insertion to be the number of changes which must be made to pointers, counts, and identifier fields when inserting a node.

Two theorems are now presented which provide lower bounds on storage and insertion costs.

Theorem 8: If an implementation is k-detectable, then any correct update of an instance must make at least k + 1 changes to structural and redundant data.

Proof: By a "correct" update, we mean one which transforms one correct instance into another. Thus, this is a direct consequence of the definition of detectability (as stated informally in the introduction), since any sequence of k or

fewer changes to an instance of a k-detectable implementation must produce an incorrect instance.■

This result directly bounds the number of changes which must be made in updating an instance. In the case of double-linked lists, we have 2-detectability, so at least three changes are required in any correct update.

In practice, the bound of Theorem 8 is likely to be significantly smaller than the number of changes performed by a "typical" update routine. For many families of related implementations, the number of changes performed by an update routine will always be much greater than the bound, but the variation between implementations will closely follow the variation in the bound. Thus, detectability may be of even greater significance in determining update cost than the result of the theorem directly indicates.

Theorem 9: If a data structure implementation is r-correctable then there are at least r+1 edge-disjoint paths to each node of the instance.

Proof: We make use of a graph-theoretic result, which states that the maximum number of edge-disjoint paths is equal to the minimum number of edges whose deletion destroys all paths (Theorem 11.4 in [1]). Thus if there are fewer than r+1 edge-disjoint paths leading to some node in an instance of an implementation, there is a set of r changes which destroys all the paths to that node. (For example,

change all those pointers to nulls.) This would be a sequence of r or fewer changes which would completely disconnect the node from the rest of the instance, making it impossible for a correction routine to perform correction, because it is completely unable to find the node in question. (The last part of the argument may sound unconvincing, but it is easy to demonstrate formally once the proper foundation is available.)■

This result means that, minimally, there must be r+1 pointers to each node in an r-correctable implementation. Of course, this does not mean that each node must contain at least r+1 pointers (some nodes may not contain any pointers), but does mean that the total number of pointers in an instance of N nodes must be at least N*(r+1). Thus, the correctability determines a lower bound on the storage cost, and this lower bound rises linearly with the correctability.

We may present the storage and insertion costs and the detectability and correctability of a set of implementations on a single graph to help visualise the effects of selecting different implementation options. Such graphs will be used in the next section to summarise the results obtained for linear lists and binary trees. The independent variable may be a parameter of the implementation, or members of a set of related implementations may simply be assigned arbitrary locations on the abscissa. The dependent variable for

detectability and correctability is the number of changes. For storage cost, it is the number of storage locations and for update cost it is the number of changes.

3. APPLICATIONS

In this section, the theory is applied to four implementations of linear lists and three implementations of binary trees. Both robustness and performance implications are examined for each implementation.

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. Adding nodes to, and deleting nodes from, instances of such an implementation is quite simple and efficient, but the implementation is not at all robust. Specifically, it is 0-detectable and 0-correctable. Such an implementation 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 implementation which is more robust adds an identifier field to each node, replaces the null pointer in the last node by a pointer to the "list head"

node of the list, and stores a count of the number of nodes on the list. This adds an additional word to each node of the list, and requires four changes to insert a node: two pointers, an identifier field, and the count. It also has the effect of making the implementation 1-detectable, although still 0-correctable. (Let us call this a "single-linked" implementation.)

The most robust of commonly-used implementations is the double-linked list. In a double-linked list, there is an additional pointer from a node to its predecessor 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 implementation is 2-detectable and 1-correctable, as we have already shown.

Finally, we may consider a novel implementation, 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 implementation, which is referred to as a "modified(2) double-linked list," is 3-detectable and 1-correctable.

Using the results of Section 2, we can sketch a proof of the 3-detectability and 1-correctability. The

implementation is 2-determined and 3-count-determined (we can use the forward pointers or either of the two interleaved sets of back pointers to calculate the count). Thus, by Theorem 5 (with n=1 and k=m=2) we conclude that ch-same $\geq$ 4. By Theorem 6 (k=3, j=1) we conclude that ch-diff $\geq$ 4. Inserting these values in the result of Theorem 4, we obtain the 3-detectability result. Then Theorem 7 indicates that the implementation is 1-correctable.

This last implementation 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 implementations in a "cost and effectiveness graph" (Figure 3.1).

3.2 Binary Trees

The binary tree is a very commonly-used data structure, but ad hoc detection and correction techniques for such structures 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.

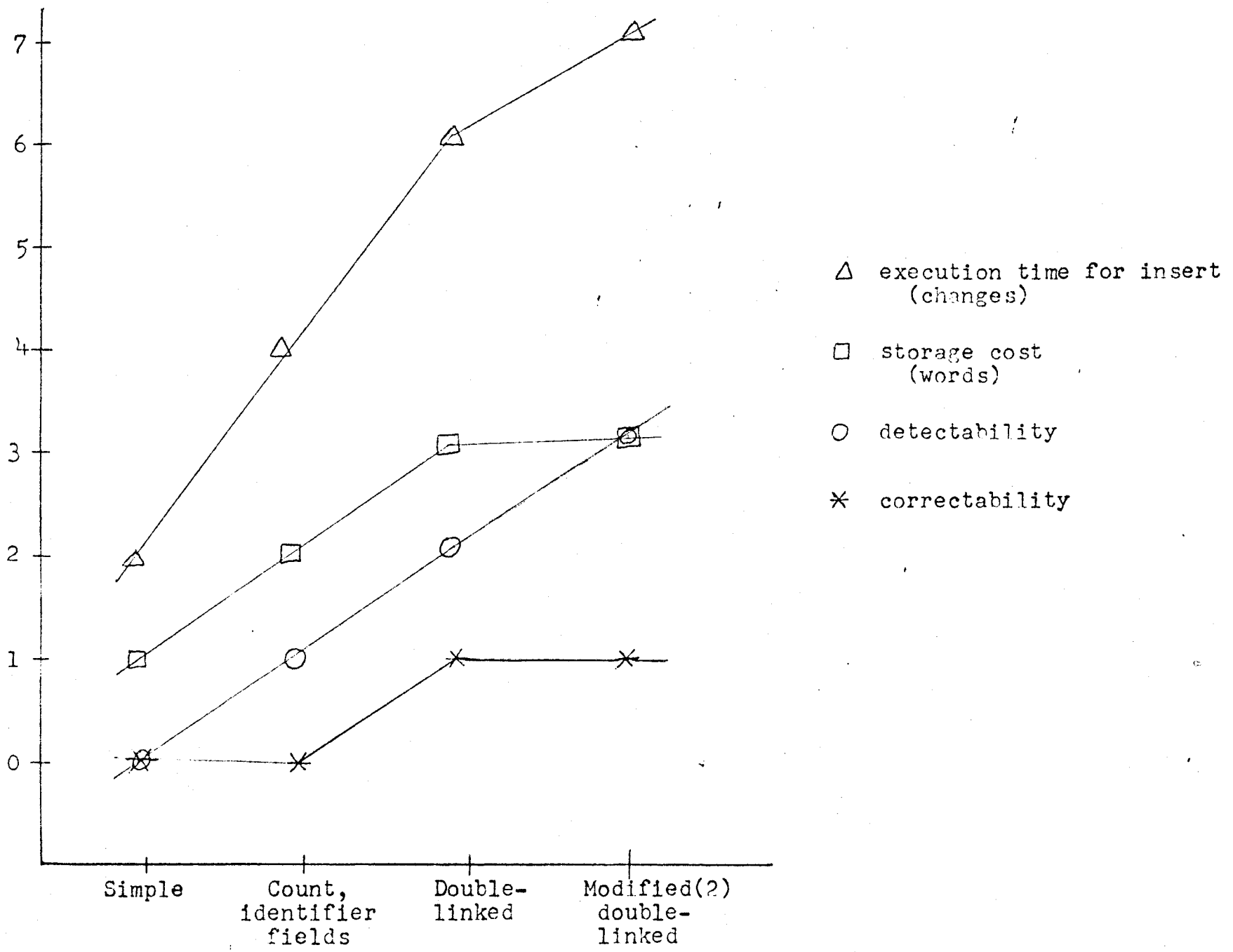


Figure 3.1 Cost and effectiveness graph for linear list implementations

The usual implementation of 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 [2, pp315-332].

Two obvious kinds of redundancy to add to a binary tree implementation are identifier fields and a count of the number of nodes in the tree. By the corollary to Theorem 3, this yields 1-detectability and 0-correctability. 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 $O(n^2)$. If we are considering only single changes 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 statistics are not changed.

Another kind of redundancy which is sometimes added to binary trees to improve efficiency is a "thread link" [2, pp349-320]. 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 flag in each node is used to indicate whether the right pointer is a normal link or a thread. As shown in [4, 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 structure is not 2-detectable, as shown by Theorem 1 and illustrated in Figure 3.2. 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.

We would like to obtain a 1-correctable implementation of a binary tree. To satisfy the hypothesis of Theorem 7, there must be two edge-disjoint paths to each node of a structure. 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 inorder 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, inorder will be most convenient.

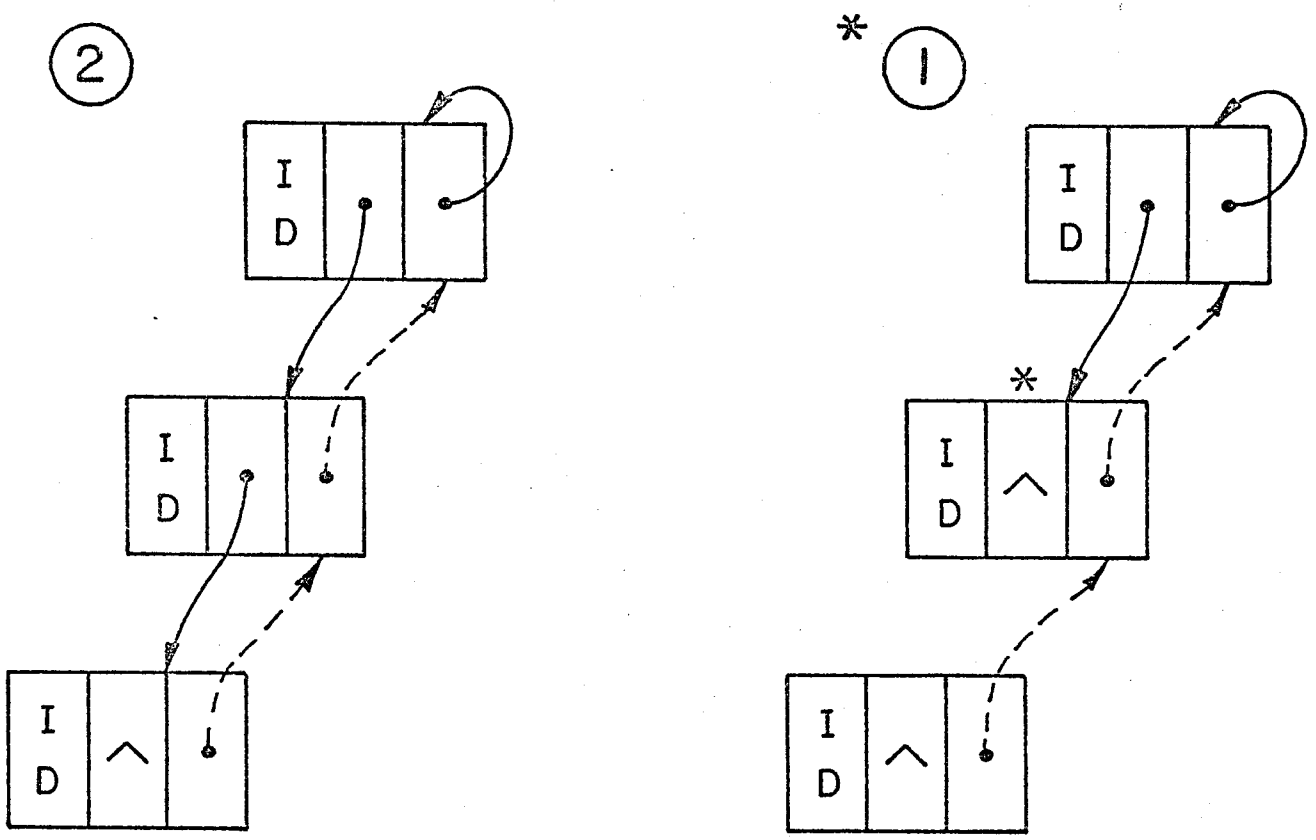


Figure 3.2 Undetectable Double Change

This structure will be called a chained and threaded binary tree. The nodes with logically null left links, joined in inorder, will be called the chain, and the links joining them will be called chain links. It can be shown [4, Section 5.4] that this implementation is 2-detectable and 1-correctable.

The results obtained here for binary tree implementations are summarized in a cost and effectiveness graph (Figure 3.3). It should be noted that the simple implementation with a count is 1-detectable but does not have a linear time, read-only detection procedure, whereas all other detectabilities can be achieved by linear time procedures.

4. EMPIRICAL RESULTS

The purpose of the experimentation is to determine the effect of applying "random" changes to a data structure instance. This investigation is distinct from the theoretical results, which assume an "intelligent" source of changes.

An analogy from network design may be helpful in understanding the distinction. When selecting a topology intended to keep a network connected in spite of line or node failures, one of two assumptions about failures must be selected: failures are random, or failures are caused by an intelligent adversary who knows the structure of the network

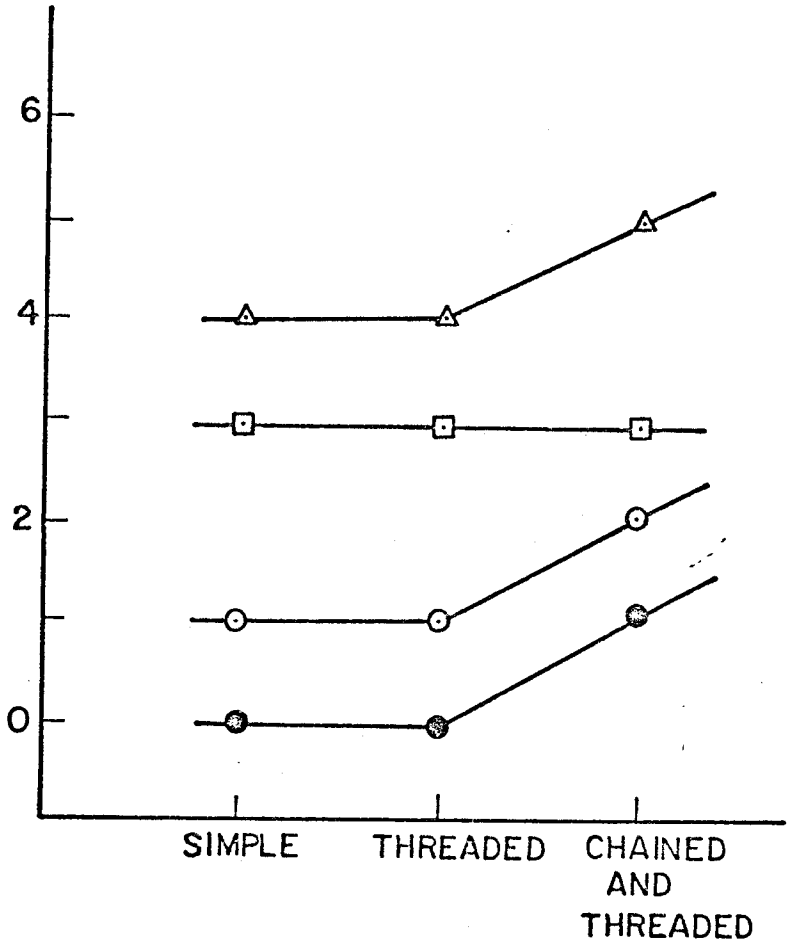


Figure 3.3

Cost and effectiveness graph for binary tree implementations

[6]. The situation in data structure robustness is similar. The theoretical results determine the minimum number of changes which are necessary to make the changes undetectable or uncorrectable, and thus correspond to an "intelligent adversary" model for the source of the changes. The experiments, in contrast, use a random source of changes.

Although the change sources are different, the theoretical results do partially predict the results of the experiments. For example, if an implementation 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 an implementation, but at present only empirical results are available. (It should also be noted that the question, as stated, is quite vague. The answer depends on various parameters, such as how one selects "random changes.")

4.1 Methodology

The basic experimental technique was to introduce changes to 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. There are two reasons for this. The first is that the theoretical results, although applicable to both internal and external data structures, will in many cases be more important for external data structures. The second 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, provide support for such simulated files, and 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 implementation, 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. 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 one set of experiments was to estimate the probability of random changes producing undetectable errors in linear list implementations. 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 implementations 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 a reasonable procedure which is given 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 implementations 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 implementation, 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. SUMMARY, CONCLUSIONS, AND FURTHER WORK

From the theoretical results, we observe that achieving very high detectability or correctability has serious performance implications. It is thus comforting to observe that, in practical situations, comparatively modest degrees of detectability and correctability can prove to be quite effective. One might state, that for typical applications requiring fairly robust data structure implementations, a 2-detectable, 1-correctable implementation should initially be assumed to be sufficient. Only if such an implementation proves to be insufficiently robust in practice should a more robust implementation be sought.

We have seen that commonly-used techniques, in the case of linear lists, can be quite effective. The modified(2) double-linked implementation suggests that the commonly-used techniques may not necessarily be the best. The increased detectability could be useful in some cases; the decreased probability of disconnection is important because it helps in performing correction.

For binary trees, the authors are aware of no commonly-used implementations 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.

An important practical conclusion from the experiments is that introducing random changes to a stored data structure instance is an excellent method for testing detection and correction procedures, and update algorithms which are intended to be fault-tolerant. Although manglers have been used occasionally for many years, they do not seem to be widely known. They are highly recommended for testing fault-tolerant software.

In Sections 2 and 3, we frequently assumed that we did not find nodes "outside" an instance which have pointers to nodes of the instance or identifier values belonging to the instance. A discussion of the restrictions which must be placed on a system to make this assumption true is in [4]. A more general approach to detectability which does not require this assumption can be found in [5].

There are a number of areas related to the work described in this paper which require further study. The most obvious of these is the development of a unified approach to robustness encompassing both content and structural data. It may well be impossible to prove completely general results, but even limited results would be quite helpful.

A second area which has also already been alluded to is

the theoretical analysis of random changes. This is likely a very challenging problem since it is necessary to identify the significant parameters which may affect the result and then to perform analysis, allowing as many as possible of these parameters to vary. Underlying the desire to perform probabilistic analysis is a concern about the nature of changes which a robust system must cope with in a practical situation. It is presently unknown whether such changes can best be modelled by a "random" model, the deterministic model implicitly used in the present theoretical analysis, or some intermediate model.

The results can be extended to more complex structures by noting that complex structures can be factored (decomposed) into basic structures such as trees and lists. These basic structures may be made more robust in order to improve the robustness of the complex structure, thereby enhancing the robustness of the software system.

BIBLIOGRAPHY

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

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

3. 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.

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

5. 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.

6. Wilkov, R. S. Analysis and design of reliable computer networks. IEEE Transactions on Communications, vol. 20, no. 3 (June 1972). pp660-678.