ROBUST STORAGE STRUCTURES FOR DATA STRUCTURES

David J. Taylor

Research Report CS-80-36

Department of Computer Science
and
Computer Communication Networks Group
University of Waterloo

July 1980

Key Words: data structures, data representation, robust systems, redundant encoding, fault tolerance, error detection, error correction

ABSTRACT

Systems which are required to operate reliably must usually contain mechanisms for detecting, and possibly correcting, errors. One such detection and correction technique makes use of redundancy in a system's data structures. This paper describes techniques for analysing the benefits obtainable from a redundantly-encoded data structure.

1. INTRODUCTION

There are two complementary approaches to software reliability: fault tolerance and fault intolerance. Fault intolerance embodies such techniques as: structured design and coding, proof of correctness, and debugging. Fault tolerance embodies the detection of, analysis of, and recovery from faults so that they do not lead to failures. This paper describes one particular approach to fault tolerance: the detection and correction of errors in stored representations of data structures.

The terms just introduced are in general use but are not always used with precisely the same meaning. Here, definitions proposed by Melliar-Smith and Randell [7] will be used: 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.

In discussing data structures, the following terms will be used: A data structure is defined to be a logical organisation of data. We define a storage structure to be 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 <u>encoding</u> 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, a 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 it is specified that the pointers are stored as absolute addresses, that is an encoding of a binary tree. (This terminology is adapted from Tompa [10].)

A <u>data structure instance</u> is a particular occurrence of a data structure. Since context will make the meaning clear, "data structure instance" will also be used to refer to an occurrence of a storage structure or an encoding.

Here, an encoding of a data structure is considered to be defined by its <u>detection procedure</u>. A detection procedure is an algorithm which is given an alleged instance of a data structure and returns a binary value indicating whether the instance is acceptable. If a data structure encoding is described in some other form, it is possible to show equivalence with a detection procedure using arguments similar to a proof of program correctness.

Although all results in this paper formally apply to encodings of storage structures, in practice the details of the encoding are often irrelevant. Hence, it is frequently convenient to assume that a detection procedure defines a storage structure rather than a particular encoding of the storage structure. Thus, throughout the paper, "<proc>" will be used to denote a detection procedure, and usually no distinction will be made between the associated storage structure and encoding.

The ultimate objective of the research described here is to provide guidance in constructing storage structures for fault tolerant systems. Ideally, given a data structure and a fault tolerance requirement, one would like to have a method for producing an appropriate storage structure. The work described here accomplishes a more limited goal. It allows error detection and correction properties of storage structures to be determined, which provides a basis for choosing among a set of alternative storage structures. The work is in a sense parallel to (and complements) that of Gotlieb and Tompa [4] which provides a technique for selecting a storage structure from a set of alternatives based on efficiency considerations.

The work described here is also restricted to the "structural" aspects of data structures, as opposed to their "data content." Superior robustness can likely be achieved in many cases by a unified treatment of content and structure, but this course has not yet been pursued because the possible information content of data structures varies so widely.

The subject of this paper may be called "data structure robustness," where "robustness" is used (informally) to denote error detection and correction capabilities. Section 2 provides the basic mathematical foundations for the study of data structure robustness. Sections 3 and 4 present results related to error detection and error correction, respectively. The last section provides some conclusions and outlines areas requiring further study. The results presented in this paper have been applied to various storage structures for linear lists, binary trees, and B-trees. These applications are described in other papers [2, 8].

2. BASIC CONCEPTS

Before discussing the mathematical model used in studying robustness, the forms of redundancy which will be considered should be mentioned. Three forms of redundancy are studied: stored counts of the number of nodes in an instance, additional pointers, and identifier fields. It may be useful to define "identifier field" fairly precisely: it is a group of one or more words, usually at the beginning of a node, whose value is sufficient to determine the unique instance in the system to which the node belongs. As well, there may be different identifier field values associated with different types of nodes in an instance. Usually, given a particular pointer in a particular type of node, we know (by the rules governing the storage structure) that it must point to a node of a particular type. Thus, the path followed to a particular node is sufficient to determine its type and the value which should be stored in its identifier field. This situation, in which all identifier fields are redundant, is the only one which will be considered in this paper.

To illustrate the forms of redundancy just introduced, an example of a redundant storage structure is shown in Figure 2.1. The example is a double-linked storage structure for a simple list. Each node contains an identifier field, each node has a "back" pointer which is not essential, and a count of the number of non-list-head nodes is stored.

In this paper, we consider storage structures which consist of a <u>header</u> and a (possibly empty) set of <u>nodes</u>. 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.)

We would like to quantify the error detection and correction properties of storage structures. In order to do

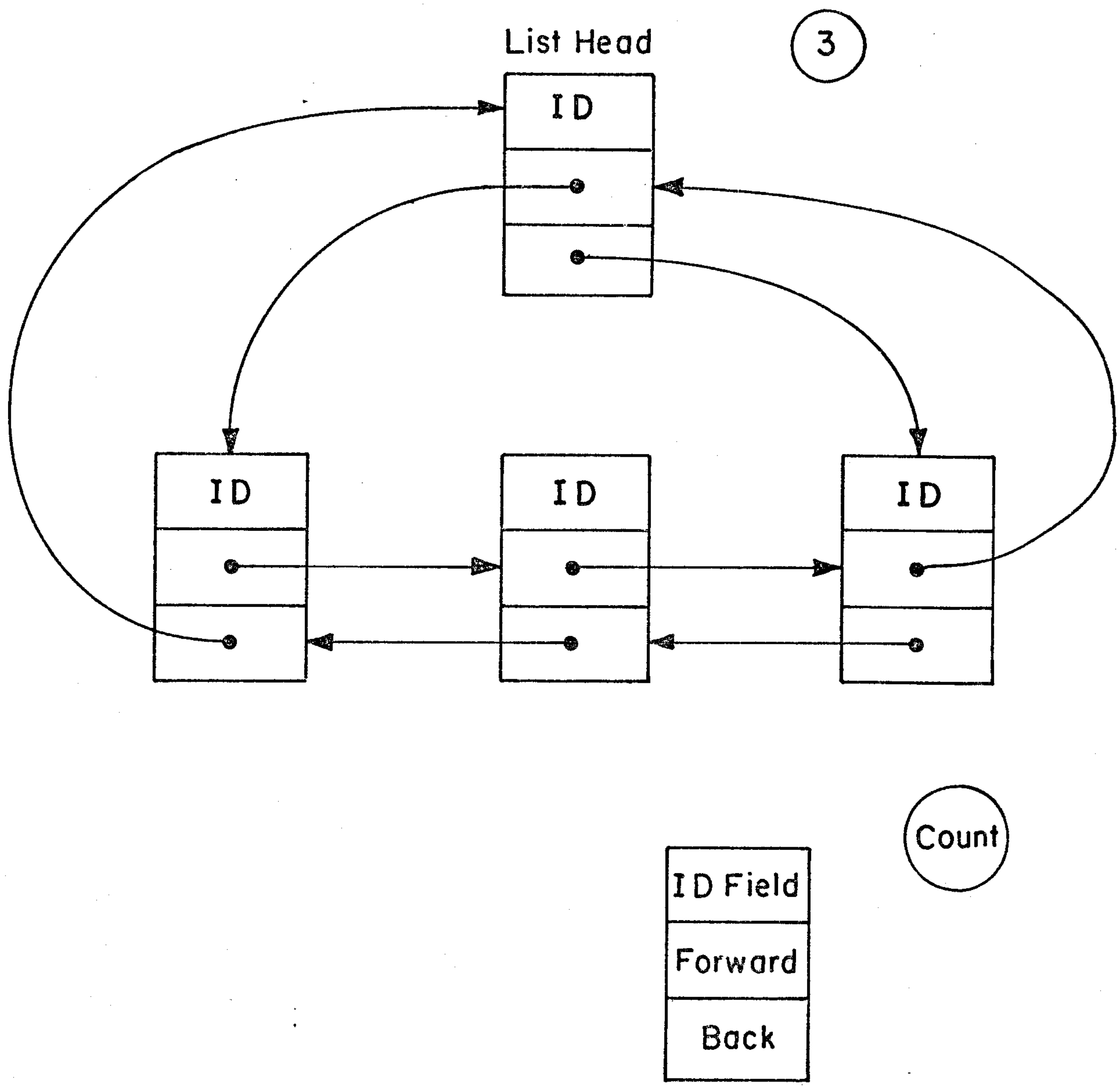

Figure 2.1 Double-linked list storage structure

this, we need first to quantify modifications to data structures. For this purpose, the term <u>change</u> is defined to be the alteration in type and/or value of a single elementary item in a memory state. We consider changes as they affect data structure instances contained in such memory states. Thus a single "error" in a software routine can easily introduce multiple erroneous changes if the routine is executed several times, or even if it is only executed once.

The size of the "elementary item" in the preceding definition should be selected in terms of the application being considered. Normally, a change will be any modification which can be the result of a single store instruction. (Here, it is assumed that a change alters the value of exactly one word.)

To illustrate the concept of change, we may consider a simple example. If we have a list of four items, A, B, C, D, each of the first three containing a pointer to the next and D containing a null pointer:

A -> B -> C -> D -> null

and 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.

The assumption which motivates the work described in this paper is that an incorrect instance differs from the correct instance it "should" be by some (small) number of changes. There are many possible sources for these changes. They might result from undetected hardware errors. They could be "wild stores" not intended to affect this instance at all. They could result from an update routine containing a bug. An important possible source of erroneous changes is the premature termination of an update routine, for example by a system crash. In this case, the instance which is left at premature termination of the update routine is an intermediate, incorrect instance which differs from the correct instance which existed before the update routine started execution and from the correct instance which would have resulted from completion of the update routine.

A system state will be called valid if it satisfies the following two conditions: (A) For each identifier field in each kind of node in each data structure instance, there is a unique identifier field value which is stored in that field; this value is not stored in any other location which could mistakenly be interpreted as a correct identifier field. (B) The only pointers to a node are found in other nodes of the data structure instance containing that node.

In practice, there will likely be a restricted area of main or secondary storage in which nodes of a data structure instance may occur. When considering that instance, we need only require the above conditions to be satisfied by that area of storage.

Requiring all memory states to be valid simplifies analysis, but is unreasonably restrictive in many cases. An important reason for relaxing the second restriction is to allow linking between different data structure instances or to allow a single node to be part of two instances simultaneously. Thus, we will allow limited violation of the above assumptions.

We wish to define a measure of the violation of the valid state conditions with respect to a particular data structure instance. To do this, we consider the fields of each type of node in the instance to be classified as: identifier field, pointer, or other. For each type of node in the instance we examine all areas of storage which are not nodes of that type (belonging to this instance) but could potentially be mistaken for a node of that type. (Call these areas "potential nodes.") For each such potential node, count the number of "identifier fields" which have the correct identifier value. The maximum count, for all potential nodes and all node types is defined to be the _identifier field invalidity_. Similarly, we count the number of pointer fields, in such potential nodes, which point to the nodes of the instance in question. The maximum of these counts is defined to be the _pointer invalidity_. The _invalidity_ is simply an ordered pair:

(identifier field invalidity, pointer invalidity).

Any valid state has an invalidity of (Ø,Ø), but because the definition of "valid" is so extremely restrictive, not all states with invalidity (Ø,Ø) are valid. (Invalid states with invalidity (Ø,Ø) are precisely those in which pointers to nodes of an instance occur outside the instance, but occur in locations whose alignment prevents them from being mistaken for legitimate pointers.) "Valid" could be redefined to mean "invalidity = (Ø,Ø)"; the definition above is used only for compatibility with previous papers.

(Note that if boundary alignment restrictions do not prevent nodes from overlapping, pointers in the instance itself may have to be counted in determining the invalidity. For example, if nodes are eight words long and must begin on a "quad-word" boundary, then we must consider as potential nodes those areas which overlap the first or last half of a node. Thus, pointers in genuine nodes may contribute to pointer invalidity. Figure 2.2 shows an example of such a storage structure, in which the pointer invalidity must be one or greater.)

The concept of invalidity was not used in [9]. Its use here is the major difference between these results and those previous ones. Thus when [9] is cited for further details, it should be noted that the discussion there will not consider invalidity.

We can now develop a mathematical model of change detection and correction in terms of metric spaces. The

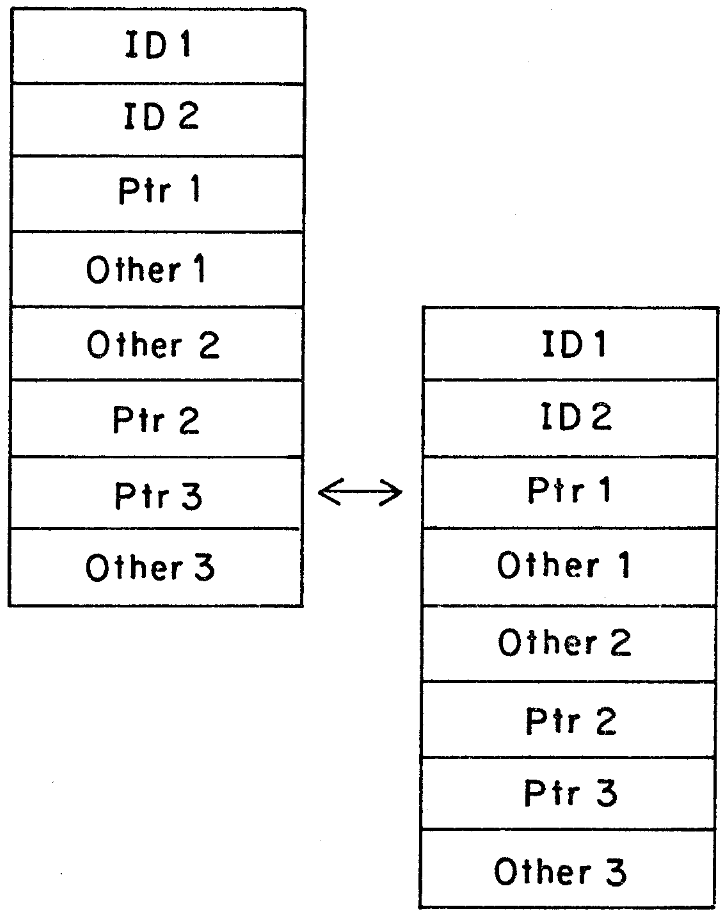

Figure 2.2 Storage structure with pointer invalidity greater than or equal to one

metric used is analogous to the one used by Hamming in the study of binary codes [5].

Let S be a set of memory states. Define a metric d on S by: if x and y are states in S, then d(x, y) is the minimum number of changes needed to transform x into y. To verify that d is a metric, one can easily show:

$$d(x, y) = 0 \text{ iff } x = y$$

$$d(x, y) = d(y, x) \text{ for all } x, y \text{ in } S$$

$$d(x, y) + d(y, z) \geq d(x, z) \text{ for all } x, y, z \text{ in } S.$$

Although d is defined on memory states, it is often convenient to refer to the distance between two instances of a data structure, rather than the distance between the memory states containing the instances.

Let <proc> be a detection, and possibly correction, procedure for a storage structure stored in the memory whose states are represented by S. Then <proc> induces an equivalence relation ind(<proc>) on S, which we will call *indistinguishability*: x ind(<proc>) y <=> <proc> cannot distinguish between x and y (that is, each memory location examined by <proc> has the same value in x as in y). It is trivial to verify that ind(<proc>) is an equivalence relation. (Intuitively, two states are indistinguishable if the differences between the two states do not affect the instace checked by <proc>.)

Let E(<proc>)(x) denote the equivalence class of x under ind(<proc>), that is, {y | x ind(<proc>) y}. When confusion will not arise, E(<proc>) will simply be denoted E.

For each memory state in S, <proc> either *accepts*, that is, concludes that the data structure instance is correct, or *rejects*, that is, concludes that the data structure instance is incorrect and possibly also corrects the structure. Define the subset C(<proc>) of S to be the set of memory states for which <proc> accepts. When the interpretation is obvious from context, C(<proc>) will simply be denoted C. Since two equivalent states are indistinguishable by <proc>, we observe that:

x ind(<proc>) y => (x in C <=> y in C).

Thus, for all x in S, E(x) is a subset of either C or the complement of C.

Define V(<proc>)(i,p) to be the set of all memory states whose identifier field invalidity does not exceed i and whose pointer invalidity does not exceed p. Let C'(<proc>)(i,p) be the intersection of C(<proc>) and V(<proc>)(i,p). When they are understood, the specification of the procedure and/or the specification of the invalidity will be omitted.

Call a detection and/or correction procedure *reasonable* iff given the address of the header of a data structure instance, it locates all other nodes or potential nodes of the instance by following pointers from nodes it has already located. (The main objective of this definition is to exclude the use of exhaustive memory searches.) Subsequently, unless otherwise specified, all detection/correction procedures will be assumed to be reasonable.

We now wish to define the detectability of a storage structure in terms of the mathematical model. The essential idea is to state that if a minimum of n changes separate any two distinct correct instances of a storage structure, then any set of n-1 or fewer changes can be detected. We may restrict one or both of the memory states involved by specifying a maximum invalidity. We thus define three forms of <u>detectability</u>:

(For each minimum the additional condition ~(x ind(<proc>) y) is to be understood.)

$$\text{det}(\langle proc \rangle, (i,p)) = \min_{x \text{ in } C'(i,p),\ y \text{ in } C} d(x,y) - 1$$

$$\text{weak-det}(\langle proc \rangle) = \min_{x,\ y \text{ in } C} d(x,y) - 1$$

$$\text{abs-det}(\langle proc \rangle, (i,p)) = \min_{x,\ y \text{ in } C'(i,p)} d(x,y) - 1$$

We will refer to these as detectability, weak detectability, and absolute detectability, respectively. Intuitively, detectability says that, starting from a well-behaved correct state (invalidity at most the specified value), a certain number of changes must be made to reach any distinct correct state. Absolute detectability requires that the correct state reached after applying changes also be well-behaved. Weak detectability allows arbitrary violations of the valid state conditions. (Note that if i and p are sufficiently large, all three detectabilities will specify the same value.)

If n ≤ det(<proc>, (i,p)) and the values of i and p are understood from context, we will simply say that the storage structure is n-detectable. Similarly, n-weak-detectable and n-abs-detectable can be defined.

We will be taking the intuitive meaning of n-detectable to be that if a correct instance in a well-behaved memory state has n changes applied to it, then we can detect that the changes have been made, by observing that the resulting instance is incorrect. This is justified by the following theorem:

Theorem 2.1: If n changes are made to a memory state containing a data structure instance which is correct, the invalidity of the state is at most (i,p), and 1 ≤ n ≤ det(<proc>, (i,p)), then either <proc> rejects the changed data structure instance or the two instances are indistinguishable.

Proof: Let S1 be in C'(i,p). Let S1 with n changes applied be S2. Then d(S1,S2) ≤ n. Since

$$n \leq \min_{x \text{ in } C'(i,p),\ y \text{ in } C} d(x,y) - 1$$

then

$$n < \min_{x \text{ in } C'(i,p),\ y \text{ in } C} d(x,y)$$

and hence, if <proc> does not reject (i.e., S2 in C) we must have S2 ind(<proc>) S1. □

We can easily show a relationship among the three kinds of detectability.

Theorem 2.2: For all i, p

weak-det(<proc>) $\leq$ det(<proc>, (i,p))

and

det(<proc>, (i,p)) $\leq$ abs-det(<proc>, (i,p)).

Proof: If we substitute the definitions of these detectabilities, the results follow immediately from the fact that C'(i,p) is a subset of C, for all i and p. □

We can also define <u>correctability</u>: corr(<proc>, (i,p)) is the maximum number of changes which can be made to a correct instance in a memory state with invalidity (i,p) such that a procedure exists which, given the state containing the changes, can create a state indistinguishable from the state without the changes. Similarly, we define weak-corr(<proc>) by allowing the initial state to have arbitrary invalidity. If n $\leq$ corr(<proc>, (i,p)) and the invalidity is understood, we will say that a storage structure is n-correctable. We define n-weak-correctable similarly.

Using these definitions it is possible to prove a basic relationship between detectability and correctability.

Theorem 2.3: If a storage structure is n-correctable then it is 2n-abs-detectable (using any arbitrary invalidity throughout).

Proof: Suppose to the contrary that there is a storage structure which is n-correctable but has absolute detectability equal to m with $n < m < 2n$. (If $m \leq n$, then we can actually transform one correct instance into another with n changes, so n-correction in that case is patently absurd.) Then there exist two memory states, S1 and S2 in C', which are distinguishable such that d(S1, S2) = m + 1. There is thus a set of m + 1 changes which transforms S1 to S2. Select n of these and apply them to S1, yielding X. Then d(S1, X) $\leq$ n and d(S2, X) $\leq$ n. Thus an n-correction procedure does not exist which works for state X, since X might have arisen from either S1 or S2 by n or fewer changes. □

Theorem 2.4: If a storage structure is n-weak-correctable then it is 2n-weak-detectable.

The proof is completely analogous to the proof of the preceding theorem.

The converses of these results are not true. In particular, it is a simple consequence of Theorem 4.5 that the converse of Theorem 2.3 is not true. Section 4 of the paper develops a partial converse of Theorem 2.3.

3. DETECTABILITY RESULTS

This section presents a collection of results on the detectability of storage structures. The results in some cases allow detectability to be determined exactly. In other cases, the results will provide upper and lower bounds on detectability. In addition, the methods used in establishing the results may prove useful as models for special proofs about particular storage structures. It is known that the results in this section are (numerically) maximal and that the hypotheses in the theorems are essential. Examples can be constructed to demonstrate these facts, but are not included here for reasons of space. Several such examples can be found in [9].

The following terminology is needed for some of the results. We will say that a subset of the pointers in an instance determines the complete structure, if given the subset, the values in all other structural data fields can be deduced, that is, all counts and identifier fields and all other pointers. We will say that a storage structure is k-determined if it satisfies the following conditions:

(1) each instance contains k disjoint sets of pointers, each of which determines the complete structure;

(2) there is an algorithm select(j) for j = 1, ..., k which locates all the pointers in the j'th set, given the header address for a structure instance, using only pointers in the j'th set. Select(j) must use only the relative

location of pointer fields within a node in determining which pointers belong to set j.

The final part of condition (2) excludes some storage structures in which the use of a pointer is indicated by a tag field. Changes to the tag field would invalidate the arguments in some of the following proofs.

The storage structure shown in Figure 2.1 is 2-determined. It is easy to see that the forward pointers and the back pointers both determine the complete structure.

Theorem 3.1: A k-determined storage structure is (k-1)-detectable.

Proof: Consider the following detection procedure: For each j = 1, ..., k use select(j) to locate the j'th set of pointers. Use these pointers to determine values for all other structural data and compare with the values which are actually in storage. If there is a mismatch, report an error. If there are no mismatches for any j, report that the instance is correct.

To prove (k-1)-detectability we must show that any set of k-1 or fewer changes is detected. Since there are fewer than k changes, at least one of the sets, say set q, contains no changes. Then select(q) finds the same data it would find in the unchanged instance and thus all the other structural data is determined to be as in the unchanged instance. Since some changes to the stored data structure have been made, a mismatch will occur and thus an error will be reported. If there have been no changes, the procedure will accept the structure instance, so the storage structure is (k-1)-detectable. □

Theorem 3.2: If a k-determined storage structure whose detection procedure is <proc> contains m identifier fields per node and has a stored count, and if one or more of the k sets of pointers has only one pointer to each node, then for all p, for all i < m, det(<proc>, (i,p)) $\geq$ k.

Proof: Use the detection procedure defined in the proof of Theorem 3.1.

If k or fewer changes are made to a correct instance and one of the k sets has no changes, the argument of Theorem 3.1 applies. The only other possibility is that exactly k changes are made, one in each set of pointers. By hypothesis, one set contains only one pointer to each node. The one pointer change made in this set causes the node pointed to by the unchanged pointer value to disappear from the structure determined by this set of pointers, so the stored count cannot agree, unless a "foreign" node has been added to the structure.

If a foreign node has been added, to create a correct instance there must be a pointer to it in each of the k sets, but the foreign node initially contains at most i correct identifier fields, so at least m-i $\geq$ 1 changes must be made to the foreign node. Thus, at least k+1 changes are required to insert a foreign node into a correct instance. □

Theorems 3.1 and 3.2 can be applied to the storage structure of Figure 2.1. Theorem 3.1 proves it is (at least) 1-detectable; Theorem 3.2 proves that it is (at least) 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 3.2:

Corollary 3.1: If a storage structure whose detection procedure is <proc> has m identifier fields per node and a stored count, and there is only one pointer to each node, then for all p, for all $i < m$, det(<proc>, (i,p)) $\geq$ 1.

We next define three properties of a storage structure which may be useful as intermediate steps in calculating detectability. For convenience, the specification of <proc>, the detection procedure, is omitted. We define <u>ch</u>-<u>same</u>(i,p) to be the minimum number of changes that transforms an instance in C'(i,p) into a distinct correct instance containing the same set of nodes. <u>Ch</u>-<u>repl</u>(i,p) is the maximum number of changes required to replace one or more nodes in an instance in C'(i,p) with the same number of foreign nodes, from outside the instance, yielding a correct instance with the same number of nodes. Similarly, let <u>ch</u>-<u>diff</u>(i,p) be the minimum number of changes that transforms an instance in C'(i,p) into a correct instance with a different number of nodes.

First, we state an obvious result which indicates how to calculate detectability if ch-same, ch-repl, and ch-diff are known.

Theorem 3.3: For any storage structure,

det(<proc>, (i,p)) =

min(ch-same(i,p), ch-repl(i,p), ch-diff(i,p)) - 1.

The following result requires quite strong conditions on a k-determined storage structure, but does provide a means of evaluating detectability in terms of several parameters for storage structures which do satisfy the conditions.

Theorem 3.4: If each of the k sets of pointers in a k-determined storage structure contains only one pointer to each node and there are a minimum of n identifier fields per node, then

$$\text{ch-same}(i,p) \geq 2k$$

and for $i \leq n$

$$\text{ch-repl}(i,p) \geq k + n - i.$$

(For $i > n$, ch-repl(i,p) = ch-repl(n,p). This could be written directly into the expression for ch-same, but it seems an unnecessary complication. The proof will consider explicitly only the case $i \leq n$.)

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 at least n-i identifier fields in the foreign node. So, the minimum number of changes to insert one or more foreign nodes is k + n - i, thus proving the result for ch-repl.

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, then if the unchanged value is non-null, the node formerly pointed to now does not have any pointer pointing to it; if the changed value is non-null, a node now has two pointers pointing to it. (If both values are null, the pointer has not been changed.) Thus 2k changes are required, proving the result for ch-same. []

For the storage structure in Figure 2.1, k=2, n=1 so ch-same(Ø,Ø) $\geq$ 4 and ch-repl(Ø,Ø) $\geq$ 3.

We can obtain a better bound for ch-repl in the case p = Ø. If m of the k sets of pointers have exactly one pointer in each node, then

$$\text{ch-repl}(i, \emptyset) \geq k + n + m - i.$$

(For a proof see [9, Theorem 4.4.5].)

We next prove a result which complements the preceding one, by giving a bound on ch-diff. First, we define a storage structure to be <u>k-count-determined</u> if there exist k disjoint sets of pointers, each one of which can be used to determine the number of nodes in a structure instance.

Theorem 3.5: If a k-count-determined storage structure has j stored counts, ch-diff $\geq$ k + j.

Proof: Suppose an undetectable change sequence alters the number of nodes in a structure instance. To yield a correct instance, each of the j counts must be changed and since there are k disjoint sets of pointers which determine the count, we must change at least one pointer in each of the k sets, for a total of k + j changes. Note that the result is true even if j = 0 since the counts derived from the different sets can be compared with each other. It is impossible to have k = 0, since the count must be determined by the complete set of pointers. □

The storage structure in Figure 2.1 is 2-count-determined (in general if a storage structure is k-count-determined and q-determined, k $\geq$ q; here they are equal) so we conclude that ch-diff $\geq$ 3. If we apply Theorem 3.3 to this result and the result of Theorem 3.4, we conclude that the storage structure is 2-detectable. This was already shown by applying Theorem 3.2, but there are cases in which one technique would be better than the other.

Finally, we prove three results which provide upper bounds on detectability. The first gives an upper bound on ch-repl, which provides a bound on detectability, since the detectability must be less than ch-repl. The second provides a bound on ch-diff, which also provides a bound on detectability. The third is phrased to provide a lower bound on the update cost, given the detectability, but also provides an upper bound on detectability, given the update cost.

Theorem 3.6: If a storage structure has a maximum of m pointers in a node, k pointers entering a node, and n identifier fields in a node, then for $i \leq n$, $p \leq m$

$$\text{ch-repl}(i,p) \leq m + k + n - i - p.$$

(For $i > n$, ch-repl(i,p) = ch-repl(n,p) and for $p > m$, ch-repl(i,p) = ch-repl(i,m). Again, this could be written directly into the expression for ch-repl, but it seems an unnecessary complication. The proof will consider explicitly only the case $i \leq n$, $p \leq m$.)

Proof: Given a correct instance which contains one or more nodes (other than a list head node), we can substitute an arbitrary region of storage for a non-list-head node, as follows. Insert appropriate identifier field values (maximum of n-i changes). Insert pointers equal to the pointers in some selected node, except that pointers from the node to itself are set to point to the new node (maximum of m-p changes). Change all pointers to the selected node to point to the new node (maximum of k changes). The result is a correct instance, so ch-same $\leq m + k + n - i - p$. □

Theorem 3.7: If a storage structure 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 q stored counts, then ch-diff $\leq$ n + q - 1.

(A simple example of a fixed list head pointer can be seen in the threaded tree storage structure of [6, Section 2.3.1].)

Proof: The list head of an empty instance differs by at most n pointer changes from any other instance, so n pointer changes and q count changes can transform any instance to the empty instance. Thus ch-diff $\leq$ n + q - 1. □

The storage structure of Figure 2.1 has n=2, q=1, so Theorem 3.7 shows that the detectability is at most 2. We earlier showed it was at least 2, so we have determined the exact value of the detectability.

Theorem 3.8: If a storage structure 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 in a state of specified invalidity into another such instance. In a k-detectable storage structure, a correct instance in a state of specified invalidity is at least k + 1 changes distant from any other correct instance, and hence from any other correct instance in a state of specified invalidity. Thus, a correct update must make at least k + 1 changes. □

This result is significant because it illustrates an important relationship between detectability and cost. The theorem shows that if a storage structure is k-detectable for large k, the cost of updating the storage structure must also be large.

4. 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 write acc(x, <proc>) to denote the accessible set in memory state x. When the procedure, and hence the structure, are clear from the context, we simply write acc(x). (Note that in this definition and the following one, <proc> specifies the storage structure involved; the procedure itself is not relevant.)

We define the correctability radius of a storage structure, denoted cr(<proc>), to be the maximum r such that for any correct state x and any state y with d(x, y) $\leq$ r, acc(x, <proc>) is a subset of acc(y, <proc>). Thus, intuitively, the correctability radius is the largest number of changes which can be made such that it is possible to guarantee no nodes in a correct instance become inaccessible to a reasonable procedure.

We now wish to prove that a storage structure is r-correctable for

r = min(cr(<proc>), det(<proc>)/2).

We will assume throughout that the storage structure employs a sufficient number of identifier fields (as specified precisely below). First we prove that there is a reasonable procedure (as defined in Section 2) for locating all the nodes of the structure instance, and then that if a superset of nodes in the structure instance can be found, the instance can be corrected.

Lemma 4.1: If k $\leq$ cr(<proc>) changes have been made to a correct instance in a memory state with identifier invalidity i, and each node of the instance has at least i+1 identifier fields, then all the nodes in the unmodified instance can be located by a reasonable procedure which locates only finitely many nodes not part of the unmodified instance.

Proof: The complete proof of this lemma is rather lengthy; an abbreviated version is given here. (The details may be found in [9, Lemma 4.3.1].)

First, we claim that all nodes of the unmodified instance may be reached by following a sequence of pointers from the list head and that there is such a path with no more than k nodes having bad identifier fields. Let the nodes of the path be a(Ø), a(1), ..., a(n), with a(Ø) = list head, a(n) = desired node, and a(j-1) pointing to a(j) for j = 1, ..., n. This claim is not proven here: the first part is a direct consequence of the definition of cr(<proc>) and the second part should be intuitively clear, since only k changes have been made.

We now prove that the reasonable procedure in Figure 4.1, given the changed instance, locates all nodes in the unchanged instance, and that only a finite number of other nodes are located.

The parameters of the procedure are: a pointer to the list head of the structure instance (LIST.HEAD) and a bound on the number of changes which may have been made to the instance (R). The technique used is essentially a depth-first search [1, p176 and following] of the instance, with the restriction that no path have more than R incorrect identifier fields. Pointers are initially placed on NODE.STACK; as they are removed from NODE.STACK they are inserted in NODE.TABLE and the pointers from the corresponding node are placed on NODE.STACK. When a pointer is removed from NODE.STACK which is already in NODE.TABLE the pointer is normally ignored.

```
procedure NODE.LOC(LIST.HEAD, R)
begin
    pointer LIST.HEAD, integer R, pointer NODE.PTR,
    table of (pointer PTR key, integer LEVEL) NODE.TABLE,
    stack of (pointer, integer) NODE.STACK,
    pointer Q, integer I;
    create empty table NODE.TABLE;
    put (null, Ø) in NODE.TABLE;
    push (LIST.HEAD, Ø) onto NODE.STACK;
    while (NODE.STACK is not empty) do
    begin
        (NODE.PTR, I) <- top of NODE.STACK;
        pop NODE.STACK;
        if(NODE.PTR is not in NODE.TABLE or
            LEVEL(NODE.PTR) > I) then
        begin
            if(NODE.PTR is not in NODE.TABLE) then
                put (NODE.PTR, I) in NODE.TABLE
            else
                LEVEL(NODE.PTR) <- I;
            for each pointer Q in NODE(NODE.PTR) do
                if(ID(Q) is correct) then
                    push (Q, I) onto NODE.STACK
                else
                    if(I < R) then
                        push (Q, I+1) onto NODE.STACK;
        end
    end
end
```

Figure 4.1
Node locator procedure

Associated with each pointer on NODE.STACK or in NODE.TABLE is a "level" number. Intuitively, this number is the number of incorrect identifier fields encountered on a path to the node. If the level number increases beyond R, the pointer is discarded. When a pointer is removed from NODE.STACK which is already in NODE.TABLE but with a higher level number, the pointer is treated essentially as if it were not in NODE.TABLE.

The need for level numbers in NODE.STACK is obvious: if they were not present, the procedure could wander through an unbounded number of "foreign" nodes. It may not be clear that the level numbers in NODE.TABLE are essential for the correct operation of NODE.LOC. In most cases, omitting those level numbers would not affect the operation of the algorithm. For a pathological case in which they are essential, see [9].

We now proceed to prove that NODE.LOC behaves as claimed. Specifically, we prove that if NODE.LOC is given the list head of an instance which differs from a correct instance by r or fewer changes, NODE.LOC terminates and at termination, NODE.TABLE contains: a null pointer, a pointer to each node in the correct instance, and a finite number of other pointers.

Termination follows from the other properties to be proved because they establish a bound on the size of NODE.TABLE. Each iteration of the main loop either adds a new entry to NODE.TABLE, decreases the level number of an entry in NODE.TABLE, or decreases the size of NODE.STACK. (Each iteration pops an entry from the stack. Thus the size of the stack decreases unless the first IF is successful. When the first IF is successful either a new entry is added to NODE.TABLE or the level number of an existing entry is decreased.) Since the size of NODE.TABLE is bounded and level numbers are never decreased below zero, NODE.STACK must eventually become empty, terminating the procedure.

We now claim that all of the nodes a(j), j = Ø, ..., n are placed in NODE.TABLE and that eventually each has a level number less than or equal to the number of nodes in {a(Ø), ..., a(j)} with bad identifier fields. We can prove this by induction. It is clearly true for a(Ø). If it is true for a(j-1) then, when a(j-1) is placed in NODE.TABLE or when its level number is reduced to the appropriate value, a(j) will be stacked with an appropriate level number, and hence eventually placed in NODE.TABLE.

Thus, the arbitrarily selected node a(n) will be placed in NODE.TABLE, showing that all nodes of the correct instance are placed in NODE.TABLE.

Let m be the maximum number of pointers in any node. We show that the number of pointers to nodes not in the correct instance is bounded by

R*(m**(2*R-1) - 1)/(m - 1).

For each pointer which is changed to point to a foreign node, a path through foreign nodes of length at most 2*R-1 may be followed before the level number exceeds R. (The maximum occurs if the path includes R-1 nodes whose identifier fields have been changed to the correct value.) Thus, a pointer change can add at most an m-ary tree of height 2*R-1 to the set of pointers in NODE.TABLE. The number of nodes in such a tree is (m**(2*R-1) - 1)/(m - 1). Multiplying by R for R possible pointer changes yields the indicated bound. This bound can obviously be improved, but in this context the existence of a bound is sufficient. □

Theorem 4.1: Define r by

r = min(cr(<proc>), det(<proc>, (i,p))/2)

and let k $\leq$ r. If k changes have been made to a correct instance in a memory state with identifier invalidity i and pointer invalidity p, and each node in the instance has at least i+1 identifier fields, then a reasonable procedure exists which can restore the modified instance to the unchanged form. That is,

corr(<proc>, (i,p)) $\geq$ min(cr(<proc>), det(<proc>, (i,p))/2).

For simplicity it is assumed in the following proof that each node has exactly m pointers to other nodes. The theorem is true if the nodes are of different kinds having different numbers of pointers, but this generality introduces additional inessential complexity to the correction procedure and the proof. Similarly, to simplify the algorithm and the proof, we assume that each node has only one identifier field. The extension to multiple identifier fields is straightforward.

Proof: By Lemma 4.1 we can find a superset of the nodes in the correct instance. We claim that procedure GEN.CORR in Figure 4.2, given such a superset, performs the required function. The parameters are: a pointer to the list head of the instance to be corrected, the number of nodes supposed to be in the instance (which may be in error), the order of correction to be performed (r), the name of a 2r-detection procedure for the storage structure (CHECK.2R), and a table containing a superset of the nodes in the correct instance. Since at most r changes are made, some of which may reverse some of the initial r changes, at most $r+k \leq 2r$ changes to the correct instance exist during the execution of GEN.CORR. Since the storage structure is 2r-detectable, if CHECK.2R accepts an instance, it must be the unchanged, correct instance.

Now we demonstrate that each set of k changes will be reversed during execution of GEN.CORR.

First, suppose that the count was not changed. Then change j for j = 1, ..., k is either a change to the identifier field of some node a(j) or a change to the b(j)'th pointer in some node a(j). Denote the first case by (Ø, a(j), null) and the second by (b(j), a(j), c(j)) where c(j) is the original (unchanged) value of the pointer. Then a set consisting of these k changes is an element of the set

```
procedure GEN.CORR(LIST.HEAD, COUNT, R, CHECK.2R, NODE.TABLE)
begin
    pointer LIST.HEAD, integer COUNT, integer R,
    procedure CHECK.2R,
    table of (pointer PTR key, integer LEVEL) NODE.TABLE,
    integer I, integer J;
    define A to be the power set of
        {Ø, 1, ..., M} X (NODE.TABLE - {null}) X NODE.TABLE;
    for I <- Ø to R do
        for each I-sequence ALPHA in A do
        /*  by this, we mean to select a set of cardinality I
            and arbitrarily order the elements
            to form a sequence of length I  */
        begin
L1:         for J <- 1 to I do
                if(ALPHA(J, 1) = Ø) then
                begin
                    T(J) <- ID(ALPHA(J, 2))
                    ID(ALPHA(J, 2)) <- correct i.d.;
                end
                else
                begin
                    T(J) <- pointer ALPHA(J, 1)
                              in node ALPHA(J, 2);
                    pointer ALPHA(J, 1) in
                        node ALPHA(J, 2) <- ALPHA(J, 3);
                end
            if(CHECK.2R(LIST.HEAD)) then return
            else
                if(I < R) then
                begin
                    SAVE.COUNT <- COUNT;
L2:                 for J <- Ø to cardinality of
                                    NODE.TABLE do
                    begin
                        COUNT <- J;
                        if(CHECK.2R(LIST.HEAD)) then return;
                    end
                    COUNT <- SAVE.COUNT;
                end
                for J <- 1 to I do
                    if(ALPHA(J, 1) = Ø) then
                        ID(ALPHA(J, 2)) <- T(J)
                    else
                        pointer ALPHA(J, 1) in
                            node ALPHA(J, 2) <- T(J);
            end
    "correction unsuccessful";
end
```

Figure 4.2
General correction procedure

A in the procedure. When this element of A is selected by the *for* loop at L1, the changes will be reversed.

Secondly, if the count is changed, then the k-1 other changes will be reversed in the manner just described and since $k-1 < r$ the *for* loop at L2 which varies the count will be executed. Since NODE.TABLE contains at least as many nodes as the correct instance, at some point the correct count will be generated.

We have implicitly assumed that after a set of changes is tried, they are removed, leaving the instance as originally passed to GEN.CORR, before the next set of changes is tried. This can be easily verified: the vector T is used to hold the values in fields which are changed and after an unsuccessful try, is used to restore the values of the fields. □

It is probably unnecessary to consider the execution time of GEN.CORR in any detail. It is clear that its execution time will make it impractical for use as a correction procedure. GEN.CORR simply serves the purpose of showing the correctability of a broad range of storage structures. Once the correctability is known, an efficient correction procedure can be sought.

The preceding theorem provides the basic result on correctability but it is not easily applicable because "correctability radius" is not an obvious property of a storage structure. The following lemma and theorem provide more directly applicable results.

Lemma 4.2: If a storage structure provides r+1 edge-disjoint paths from the list head to each node of the structure, the correctability radius of the storage structure is at least r.

Proof: Suppose the correctability radius is less than r. Then there is a correct instance and a set of r or fewer changes which makes a node inaccessible to all reasonable procedures. But there are r + 1 edge (pointer)-disjoint paths to each node, so this is impossible. □

Theorem 4.2 (General Correction Theorem): Suppose a storage structure has at least i+1 identifier fields in each node. Using an identifier field invalidity of i and any constant pointer invalidity throughout, if the storage structure is 2r-detectable and there are at least r+1 edge-disjoint paths to each node of the instance, then the storage structure is r-correctable.

Proof: By Lemma 4.2 the correctability radius of such a storage structure is at least r. Thus, by Theorem 4.1, the storage structure is at least min(r, 2r/2) = r-correctable. □

Only Lemma 4.1 made direct use of the presence of identifier fields. There, the presence of i+1 identifier fields (with an identifier field invalidity of i) was needed to prove termination of NODE.LOC. If one allows an alternative termination argument, appealing to the finiteness of storage, the requirement for identifier fields can be eliminated.

The results in Theorems 4.1 and 4.2 are the most generally useful correctability results, but they are not maximal. The following theorems, which use absolute detectability, provide maximal results for correctability and prove their maximality.

Theorem 4.3: Define r by

$$r = \min(cr(\langle proc \rangle), \text{abs-det}(\langle proc \rangle, (i,p))/2)$$

and let $k \leq r$. If k changes have been made to an instance in C'(i,p) and each node in the instance has at least i+1 identifier fields, then a reasonable procedure exists which can restore the modified instance to the unchanged form.

Proof: Use GEN.CORR of Figure 4.2 modified so that if CHECK.2R accepts, then NODE.LOC is executed on the current memory state and CHECK.VALID of Figure 4.3 is invoked with parameters: NODE.TABLE.1 the original NODE.TABLE, NODE.TABLE.2 the new NODE.TABLE, ID.INV = i, PTR.INV = p, and I and R from GEN.CORR. CHECK.VALID determines the invalidity of the nodes in NODE.TABLE.1 with respect to the structure represented in NODE.TABLE.2. It succeeds if this invalidity is less than (ID.INV, PTR.INV). The modified GEN.CORR accepts a state (and thus terminates) if and only if both CHECK.2R and CHECK.VALID succeed.

For each node in NODE.TABLE.2 which is not in NODE.TABLE.1 (not part of the correct instance), CHECK.VALID

```
procedure CHECK.VALID(NODE.TABLE.1, NODE.TABLE.2,
        ID.INV, PTR.INV, I, R)
begin
    pointer X, pointer Y;
    integer ID.COUNT, PTR.COUNT, INV.COUNT;
    INV.COUNT <- Ø;
    for each entry X in NODE.TABLE.1 do
        if (X is not in NODE.TABLE.2) then
        begin
            ID.COUNT <- the number of correct identifier
                             fields in the node at X;
            PTR.COUNT <- the number of pointers in the node
                             at X which are in NODE.TABLE.2;
            if (ID.COUNT > ID.INV) then
               INV.COUNT <- INV.COUNT + ID.COUNT - ID.INV;
            if (PTR.COUNT > PTR.INV) then
               INV.COUNT <- INV.COUNT + PTR.COUNT - PTR.INV;
        end
    if (INV.COUNT + I < R) then return(true)
        else return(false);
end
```

Figure 4.3
Validity checking procedure

counts the number of excess correct identifier fields and excess pointers into the instance. If the total count for all nodes, plus the number of changes made by GEN.CORR (I) exceeds the total number of changes (R), then CHECK.VALID rejects the state, since it cannot be transformed to a state in C'(i,p) without exceeding the allowed total number of changes.

Thus, any state accepted by the modified GEN.CORR must be indistinguishable from a state in C'(i,p) and since the total number of changes (from the original, unchanged state) is at most $r + k \leq 2r$, GEN.CORR must create a state which is indistinguishable from the original state, and thus restores the unmodified instance. GEN.CORR does not attempt to create a state in C'(i,p) since that would involve changing nodes outside the instance it is correcting. Presumably, in an actual system, other correction routines would eventually correct those nodes.

By the same argument as in Theorem 4.1, any set of changes to nodes part of the instance will be reversed. Any changes to nodes not in NODE.TABLE.1 will be completely ignored and any changes to nodes in NODE.TABLE.1 but not in the instance will be counted as reversed by CHECK.VALID. So, under the conditions of the theorem, the unchanged state will be implicitly recreated, and the instance itself will be recreated in its unmodified form.

Thus, the modified GEN.CORR will recreate the unmodified instance and will accept no instance distinct from the unmodified instance, as required. □

Using Lemma 4.2, we easily can prove an analogue of Theorem 4.2.

Theorem 4.4: Suppose a storage structure has at least i+1 identifier fields in each node. Using an identifier field invalidity of i and any constant pointer invalidity throughout, if the storage structure is 2r-abs-detectable and there are at least r+1 edge-disjoint paths to each node of the instance, then the storage structure is r-correctable.

To prove these results maximal we need the converse of Lemma 4.2.

Lemma 4.3: If a storage structure has correctability radius r, then there are at least r + 1 edge-disjoint paths to each node of any instance.

Proof: Suppose the result is false. Then there is a storage structure with correctability radius r such that some instance contains a node X which has r or fewer edge-disjoint paths from the header of the instance.

By Theorem 11.4 in [3] the maximum number of edge-disjoint paths is equal to the minimum number of edges whose deletion destroys all paths. Thus there is a set of r or fewer pointers which are essential in accessing X from the header. Change all of these to nulls (r or fewer changes). Then X is not in the accessible set, so the correctability radius is less than r, contradiction. □

The following two theorems show that the results of Theorems 4.3 and 4.4 are maximal and thus that they are all that is needed in determining the correctability of storage structures which use a sufficient number of identifier fields.

Theorem 4.5: If a storage structure is r-correctable then abs-det(<proc>) $\geq$ 2r and cr(<proc>) $\geq$ r. (Using any constant invalidities throughout.)

Proof: The first part was proven in Theorem 2.3. To prove the second part, we note that if cr(<proc>) $<$ r, then part of the structure could be made inaccessible to all reasonable procedures by r changes, preventing any reasonable procedure from performing correction. □

Theorem 4.6: If a storage structure is r-correctable, then abs-det(<proc>) $\geq$ 2r and there are r+1 edge-disjoint paths to each node of each instance. (Using any constant invalidities throughout.)

Proof: Again, the first part has already been proven. By Theorem 4.5 we have cr(<proc>) $\geq$ r and by Lemma 4.3 there are then at least r+1 edge-disjoint paths to each node of each instance. □

5. CONCLUSIONS AND FURTHER WORK

The preceding sections introduce the study of data structure robustness and provide a number of basic theorems on the detectability and correctability of storage structures. The correctability results are, in a sense, complete, subject to a simple assumption about identifier fields. The detectability results are incomplete but should be useful, both as a collection of results which can be applied directly and as models for special-purpose proofs about individual storage structures.

The results in Sections 3 and 4 are useful in determining the resistance to damage of various storage structures. Examples of applying those results to particular storage structures are not included here, but can be found in [2, 8, 9]. Some of these storage structures have also been subjected to empirical testing, not to verify the theoretical results, but to determine behaviour under conditions not described by the theory. For example, if a storage structure with detection procedure <proc> has det(<proc>) = 2, then we know any set of one or two changes will be detected and that at least one set of three changes cannot be detected. However, the theory does not predict what fraction of the set of all possible triples of changes produces undetectable errors. In the storage structures tested which have det(<proc>) > 1, no "randomly" generated set of changes ever produced an undetectable error (3000 sets were tried for each number of changes tested). Thus, in some cases, storage structures may be even more robust in practice than the theoretical values developed here would indicate.

One line of research which should be pursued is to develop theoretical methods for determining such "probabilistic" detection and correction properties of storage structures. This seems to be much more difficult to do than the "absolute" analysis developed here. One problem is that more parameters must be considered. An example of such a parameter is the number of nodes in an instance. This did not have to be considered in the analysis performed in Sections 3 and 4, but some of the empirical results strongly suggest that this is a relevant parameter for probabilistic analysis.

A difficulty with the results presented here is that they treat data structure instances as single units for detection and correction purposes. This is undesirable if instances are very large (as in a data base, for example). Two approaches should be considered. One is to determine "local" detectability and correctability. For example, we could define a storage structure to be locally 1-detectable if an arbitrary number of changes can be detected provided they are "sufficiently far apart." (Defining the distance between changes is one of the problems which must be solved in order to develop the concepts of local detectability and correctability.) The other approach is to determine ways of partitioning large instances so they can be checked and corrected without reference to the entire instance. Of course, the objective must be to accomplish this partitioning without unduly complicating update and access routines for the storage structure.

Acknowledgements

Professors D. E. Morgan and F. W. Tompa provided helpful comments on the development of the results in this paper. Mr. J. P. Black, as well as Professors Morgan and Tompa, read earlier drafts of the paper and suggested clarifications to the presentation.

This research was supported by the Natural Sciences and Engineering Research Council of Canada under grant number A3Ø78.

BIBLIOGRAPHY

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Massachusetts, 1974.

[2] J. P. Black, D. J. Taylor, and D. E. Morgan, A robust B-tree implementation, submitted to the 5th International Conference on Software Engineering. Also available as Computer Science Research Report CS-8Ø-15, University of Waterloo, Ontario, Canada, 198Ø.

[3] J. A. Bondy and U. S. R. Murty, Graph Theory with Applications, Macmillan Press, London, 1976.

[4] C. C. Gotlieb and F. W. Tompa, Choosing a storage schema, Acta Inform., vol. 3 (1974), pp297-319.

[5] R. W. Hamming, Error detecting and error correcting codes, Bell System Tech. J., vol. 26, no. 2 (April 195Ø), pp147-16Ø.

[6] D. E. Knuth, The Art of Computer Programming, vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Massachusetts, 1968.

[7] P. M. Melliar-Smith 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 28-3Ø, 1977. (Published as SIGPLAN Notices, vol. 12, no. 3, March 1977.) pp95-1ØØ.

[8] D. J. Taylor, D. E. Morgan, and J. P. Black, Redundancy in data structures: Improving software fault tolerance, to appear in IEEE Trans. Software Engrg., vol. 6, no. 6 (November 198Ø). Also available as Computer Science Research Report CS-79-34, University of Waterloo, Ontario, Canada, 1979.

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

[1Ø] F. W. Tompa, Data structure design, in Data Structures, Computer Graphics, and Pattern Recognition, edited by A. Klinger, et al., Academic Press, New York, 1977, pp3-3Ø.