REDUNDANCY IN DATA STRUCTURES: SOME THEORETICAL RESULTS

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

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

CS-79-35

Abstract

A companion paper, "Redundancy in Data Structures: Improving Software Fault Tolerance", provides an informal introduction to robust data structures. Here, we present the underlying theory for them, and use it to discuss the synthesis and cost effectiveness of robust data structures.

Key Words and Phrases: Software reliability, software fault tolerance, robust data structures, redundancy, compound data structures, error detection, error correction.

1. INTRODUCTION

In Part I of this two-part paper, we gave an informal introduction to data structure robustness. This included basic definitions, terminology, assumptions, examples of robust data structures and their practical implementation, as well as some empirical results. The purpose of this second part is to give a more precise treatment of the subject of robust storage structures, including more general results which were very loosely argued in Part I.

We recall some of our assumptions from Part I. A storage structure consists of a header and a (possibly empty) set of nodes. We are concerned with changes to structural data in the form of pointers, counts, and identifier fields. The valid state hypothesis assumes that there are no external pointers into an instance, and that there are no identifier fields with values appropriate to a particular instance outside of the instance. Finally, we exclude all detection or correction procedures which make use of exhaustive memory searches: we restrict them to following sequences of pointers from the header of an instance.

Section 2 provides some upper and lower bounds on detectability, including additional definitions. Section 3 states and proves the General Correction Theorem by exhibiting a general correction procedure. Section 4

defines and discusses one class of compound data structures. Section 5 provides a general discussion of robust data structure synthesis which is complemented in Section 6 by a discussion of the costs and effectiveness of robust data structures. Finally, Section 7 presents a summary, conclusions, and further work.

2. DETECTABILITY

The purpose of this section is to provide means of determining upper and lower bounds on the detectability of a storage structure. 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, ch-repl, and ch-diff (defined below). Section 6 contains another useful upper bound result.

The results will be illustrated by an example, the double-linked implementation of a simple list. 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 (Upper bound on detectability): If a data structure allows an "empty" instance (i.e., an instance containing only the header), and n is the number of pointers in the header of its storage structure which do not permanently point to a fixed location in the header,[1] and there are j stored counts, then the detectability of the storage structure is at most n + j - 1.

Proof: The header 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. □

For double-linked lists, n = 2 and j = 1, and so Theorem 1 proves they are at most 2-detectable.

Several detectability results are related to the concepts k-determined and j-count-determined. These may be defined informally as follows. A storage structure is k-determined if the pointers in each instance of the storage structure can be partitioned 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.) A storage

---

[1] A pointer in the header which points to a fixed location in the header may seem unlikely, but does occur in practice. For example, see the threaded tree implementation in [2, p322].

structure is j-count-determined if the pointers in each instance of the structure can be partitioned into j disjoint sets, such that each set can be used to calculate the number of nodes in the instance. Clearly, the values of j and k in "k-determined" and "j-count-determined" are two measures of the amount of redundancy in a storage structure.

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

Theorem 2: A k-determined storage structure is (at least) (k-1)-detectable.

Proof: We may detect errors in such a storage structure 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 at least 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. []

In the case of double-linked linear lists, either the forward or backward links may be used to reconstruct an instance. This storage structure is thus 2-determined, a fact which was used implicitly in the correction routine of Part I to repair single pointer errors. It follows from

Theorem 2 that double-linked lists are (at least) 1-detectable.

Theorem 3: If a k-determined storage structure 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 storage structure is (at least) 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 from this set. Thus either the stored count cannot agree with the actual number of nodes in the instance or a "foreign" node has been added to the instance. (That is, an area of storage which is not a node of this instance 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 storage structure is k-detectable. []

We conclude from Theorems 1 and 3 that double-linked lists are exactly 2-detectable, which formalises our argument of Part I.

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 a storage structure uses identifier fields and a stored count, and there is only one pointer to each node, then the storage structure is (at least) 1-detectable.

Three properties of a storage structure are now defined which provide an alternative method of determining lower bounds on detectability. The minimum number of changes (over all correct instances) that transforms a correct structure instance into another correct instance containing the same set of nodes is defined to be ch-same. Ch-repl is the minimum number of changes required to replace one or more nodes in a data structure instance with the same number of foreign nodes from outside the instance, leaving the total number of nodes unchanged. 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 a storage structure is exactly min(ch-same, ch-repl, 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-repl, ch-diff), and the detectability is defined to be one less than this value. □

This alternative approach to calculating detectability was used informally in Part I to argue the 2-detectability of chained and threaded binary trees. However, one cannot apply Theorems 1 through 3 to CT-trees, as they are not 2-determined. One reason they are not is that although it is possible to construct a CT-tree given only the chains and threads, the construction is not unique. (See [3] for a more detailed discussion.)

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

$$\text{ch-same} \geq 2k$$

and

$$\text{ch-repl} \geq k + n + m.$$

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.

If the same set of nodes has simply been restructured,

then there must be at least two changes in each of the k sets of pointers. If only one change is made in a set, there are two cases: (1) The former value of the changed pointer was null and the new 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 former 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, and thus

$$\text{ch-same} \geq 2k.$$

To place a foreign node in an instance, 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 place one or more foreign nodes in an instance is k + n + m, and thus

$$\text{ch-repl} \geq k + n + m. \ \square$$

It might seem that we could take m as the minimum number of non-null pointers in a node, thus increasing ch-repl in some cases. The following is a counterexample.

Referring to Figure 2.1, suppose a linear list

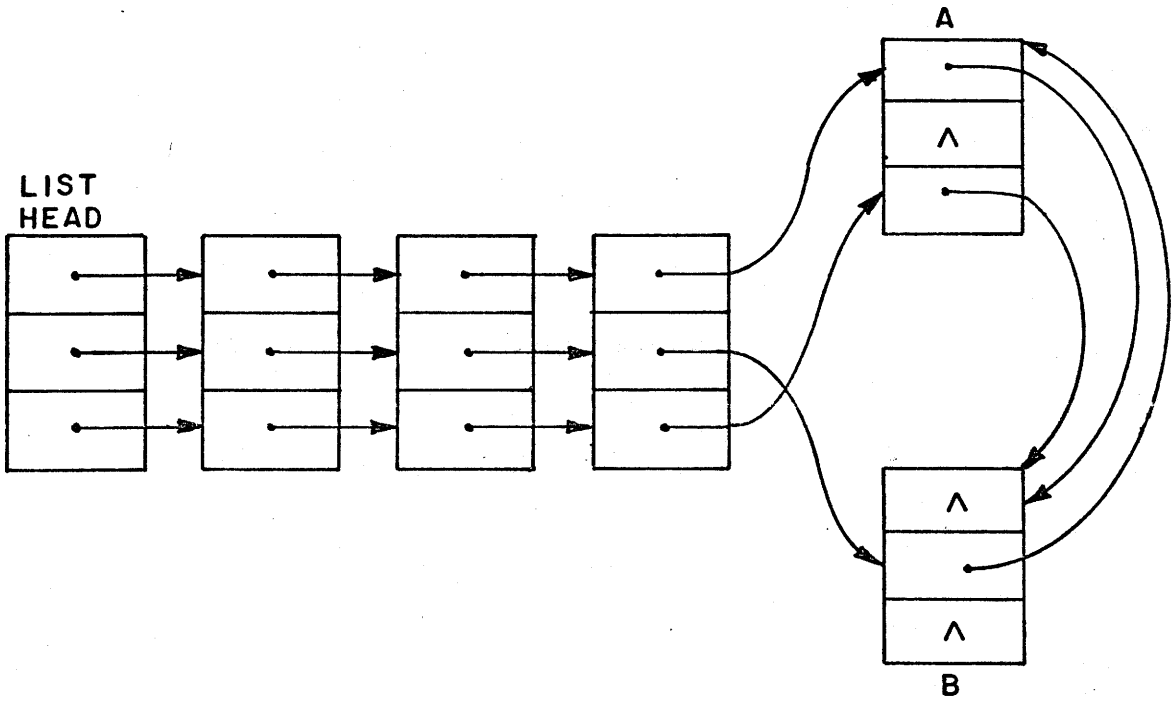


Figure 2.1 Counter Example for Revised m

implementation has three pointers from each node to the following node. The list has two special "list end" nodes, A and B, which have the property that, for odd-numbered sets of pointers, B is the last node on the list (i.e., contains a null pointer), and A is the second last. For even-numbered pointers, the roles of A and B are reversed. Such an implementation is 3-determined, and has m = 0 under the original definition. If there is a stored count but no identifier fields, then n = 0, giving ch-repl $\geq$ 3. In fact, ch-repl = 3: we can replace the pair (A, B) with two similarly structured foreign nodes (X, Y) by changing only the three pointers from C to (A, B). The revised definition of m would give m = 1 and ch-repl $\geq$ 4, which is false.

For double-linked lists, k = 2, m = 2, n = 1; thus ch-same $\geq$ 4 and ch-repl $\geq$ 5. In fact, ch-same = 6 for double-linked lists (for example, to exchange a pair of adjacent nodes requires that three forward and three back pointers be changed), but a lengthy argument, from first principles, would be required to show this.

Theorem 6: If a k-count-determined storage structure 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 ≥ 3. (In fact, ch-diff = 3.) Applying Theorem 4 to the values obtained for ch-same, ch-repl, 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 a storage structure may most easily be computed by first computing ch-same, ch-repl, and ch-diff, then applying Theorem 4.

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 [3].

3. CORRECTABILITY

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

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

Sketch of proof (the details may be found in [3]): 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.

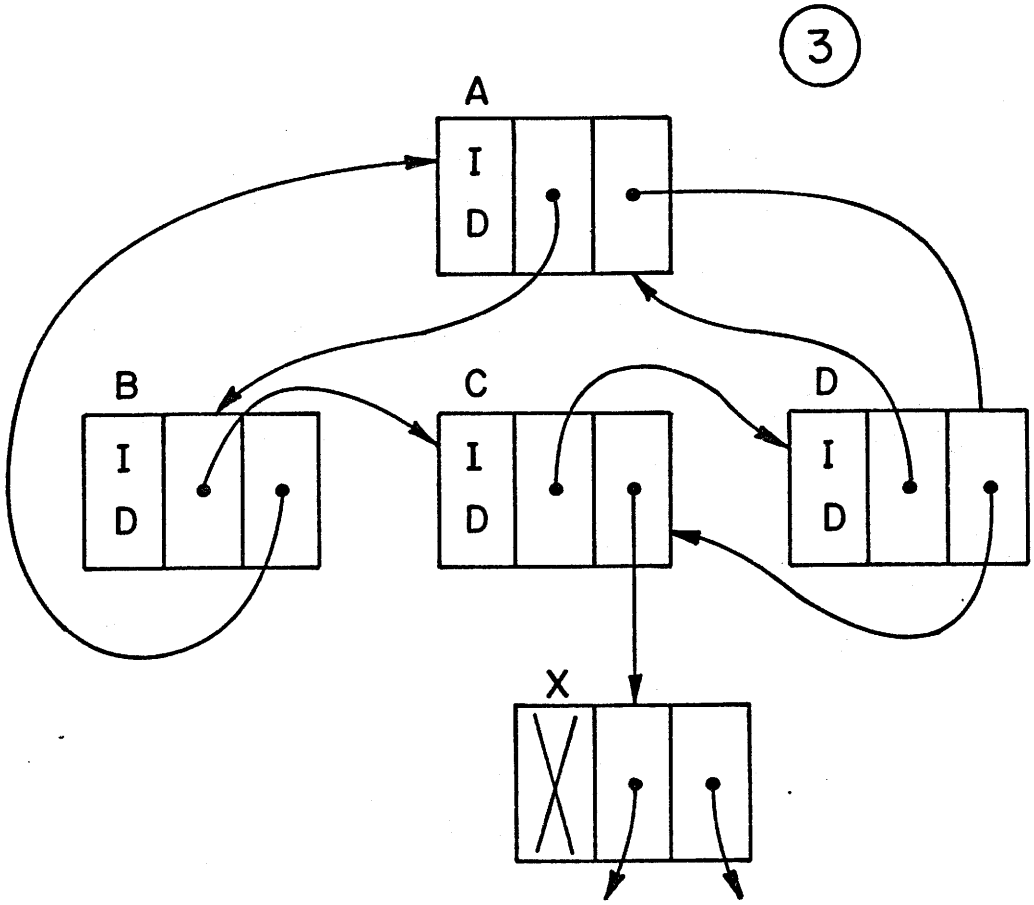
Because there were originally r + 1 paths to each node, this procedure must be able to find all nodes which were in the instance before it was changed. It may also "find" other nodes, but there is a bound on the number of such nodes it will locate.

Once a superset of the nodes has been found, correction can be performed on a trial and error basis. Since the

storage structure is 2r-detectable, any two correct instances must be at least 2r + 1 changes "distant" from each other. Sets of r or fewer changes are applied, so, we have at most 2r changes at any time. Therefore, the only correct instance which can be created is the desired instance. 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 = 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 3.1 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



A	O
B	O
C	O
D	O
X	I

Figure 3.1 Operation of General Correction Procedure

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 double-linked list of n nodes, this correction procedure will take time $O(n^3)$; however, we showed in Part I that it is possible to write a special-purpose correction procedure for double-linked lists which takes time O(n).

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

4. APPLICATION TO COMPOUND STRUCTURES

Our intent in this section is to extend the basic results presented above to suitably defined compound structures. The result is quite restrictive, but indicates at least one direction for further research.

An instance of a compound storage structure, compounded

from storage structures S(1), S(2), ..., S(t), is defined as follows:

a) The structural information in every instance of the compound structure (i.e., pointer and identifier fields) may be partitioned into t disjoint subsets.

b) The i-th set, with its list head and possible stored counts, forms a data structure instance of S(i), for i = 1, 2, ..., t.

c) Excluding the list heads of the t instances, each storage structure instance contains the same set of nodes.

Some comments regarding the definition are in order. It is important to notice that the component sub-structures are logically unrelated. Thus, while one could consider the double-linked list as a compound structure of two singly-linked lists (one of which had no identifier fields or count), this approach gives a lower value for detectability since one can no longer claim to be able to reconstruct the forward pointers from the back pointers. Note also that the third condition implies that ch-diff and ch-repl types of changes are reflected in each sub-structure.

We now compute ch-same, ch-repl, and ch-diff for a compound structure. For simplicity, we consider a compound data structure composed of two sub-structures, and we assume ch-same, ch-repl, and ch-diff are known for each of the sub-structures. We denote ch-same for the first sub-structure by ch-same(1), for the second by ch-same(2), and similarly for ch-repl and ch-diff.

a) Ch-same. As the two sub-structures are assumed to

be logically independent, we need only re-arrange the nodes with respect to one of the sub-structures in order to obtain another correct compound instance over the same nodes:

$$\text{ch-same} = \min (\text{ch-same}(1), \text{ch-same}(2)).$$

b) Ch-repl. In order to replace some number of nodes in the compound instance with the same number of foreign nodes, we must perform the same replacement on each sub-structure. (If we do not do this, then we can detect the change by simple comparison of the sets of nodes in each sub-structure. The two sets will no longer be identical.) Thus

$$\text{ch-repl} = \text{ch-repl}(1) + \text{ch-repl}(2).$$

c) Ch-diff. As with ch-repl, we must add or delete the same set of nodes for both sub-structures, and thus

$$\text{ch-diff} = \text{ch-diff}(1) + \text{ch-diff}(2).$$

Using the values just calculated and Theorem 1, we obtain the following result.

Theorem 8. The detectability of a compound structure is

$$\min (\text{ch-same}(1), \text{ch-same}(2), \text{ch-repl}(1) + \text{ch-repl}(2), \text{ch-diff}(1) + \text{ch-diff}(2)) - 1.$$

The obvious generalization to a compound data structure with

n sub-structures is:

$$\min(\text{ch-same}(1), \text{ch-same}(2), \ldots, \text{ch-same}(n),$$
$$\text{ch-repl}(1) + \text{ch-repl}(2) + \ldots + \text{ch-repl}(n),$$
$$\text{ch-diff}(1) + \text{ch-diff}(2) + \ldots + \text{ch-repl}(n)) - 1.$$

We consider two examples: a compound structure composed of two double-linked lists, and a composition of a chained and threaded binary tree with a double-linked list. See Figures 4.1 and 4.2.

We can show that for double-linked lists, ch-same = 6, ch-repl = 5, and ch-diff = 3. Combining two such structures, we find that the detectability of the compound structure is min(6,6,5+5,3+3) - 1 = 5. Thus any set of five or fewer changes to pointer, count, or identifier fields may be detected. By the addition of a "redundant" linear list, the detectability has been increased from 2 to 5. Then Theorem 7 gives 2-correctability for the compound structure.

For our other example, we compound a double-linked linear list with a chained and threaded binary tree. For the latter, we have argued that ch-same = 3, ch-repl = 5, and ch-diff = 3. For the detectability of the compound structure, we thus have min(3,6,10,6) - 1 = 2. In this case, the detectability does not increase, since ch-same of the binary tree was the term limiting its detectability, and the compound ch-same terms are not added.

Examining the theoretical results in the light of these examples allows us to make some more general remarks about

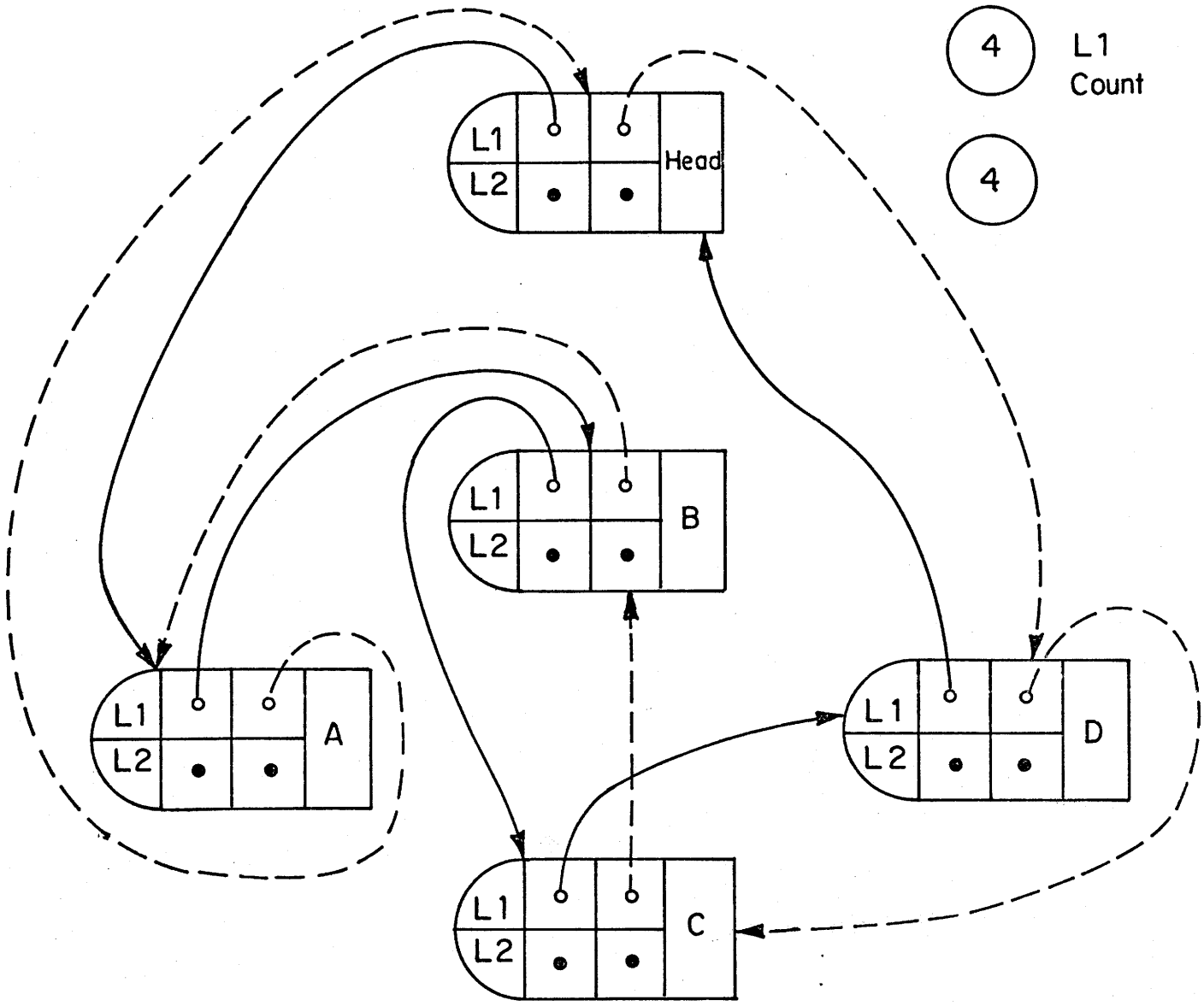


Fig. 4.1 (a) Compound structure showing first list (L1) pointers. Logical order (A, B, C, D)

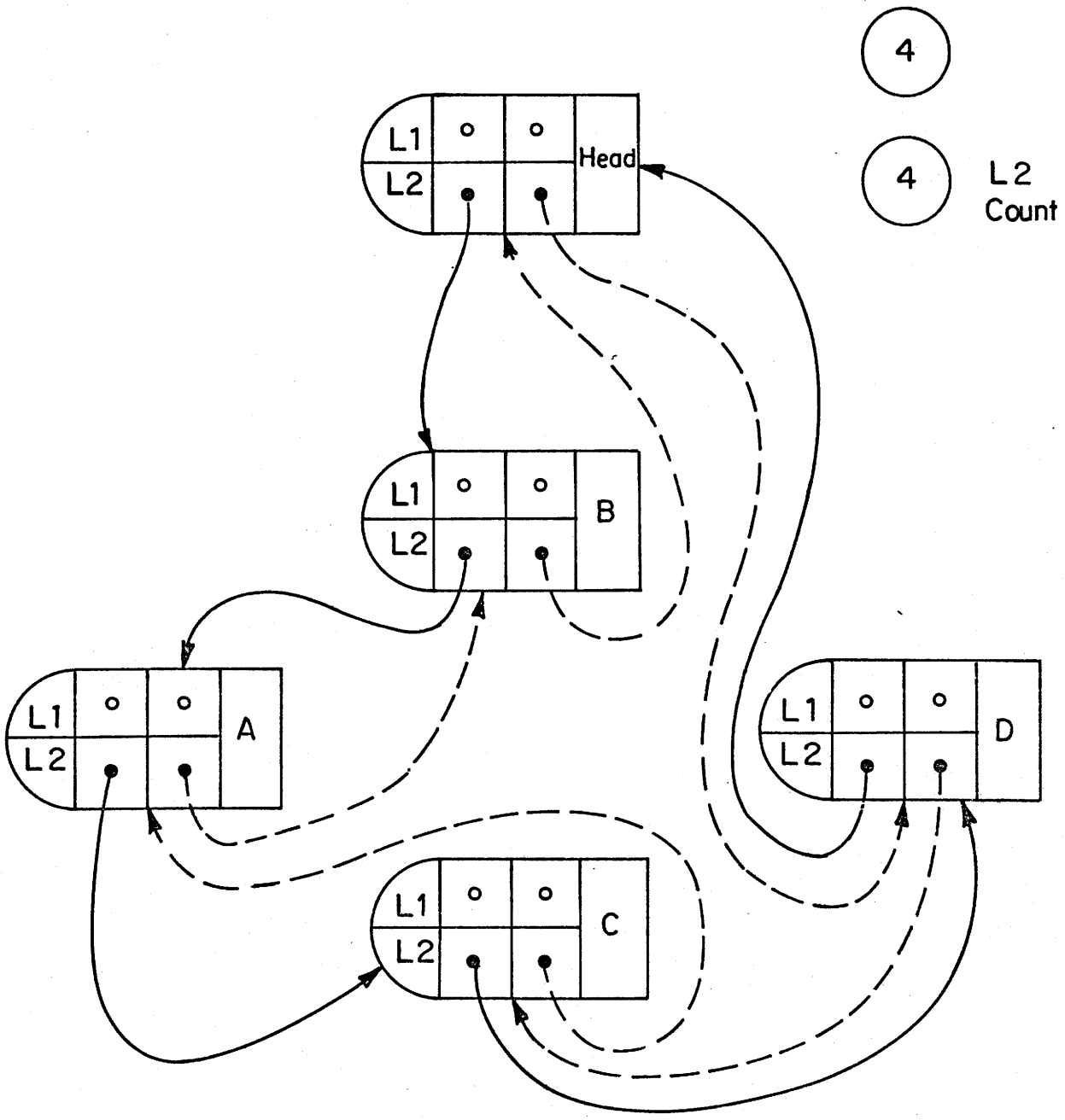


Fig. 4.1 (b) Compound structure showing second list (L2) pointers. Logical order (B,A,C,D)

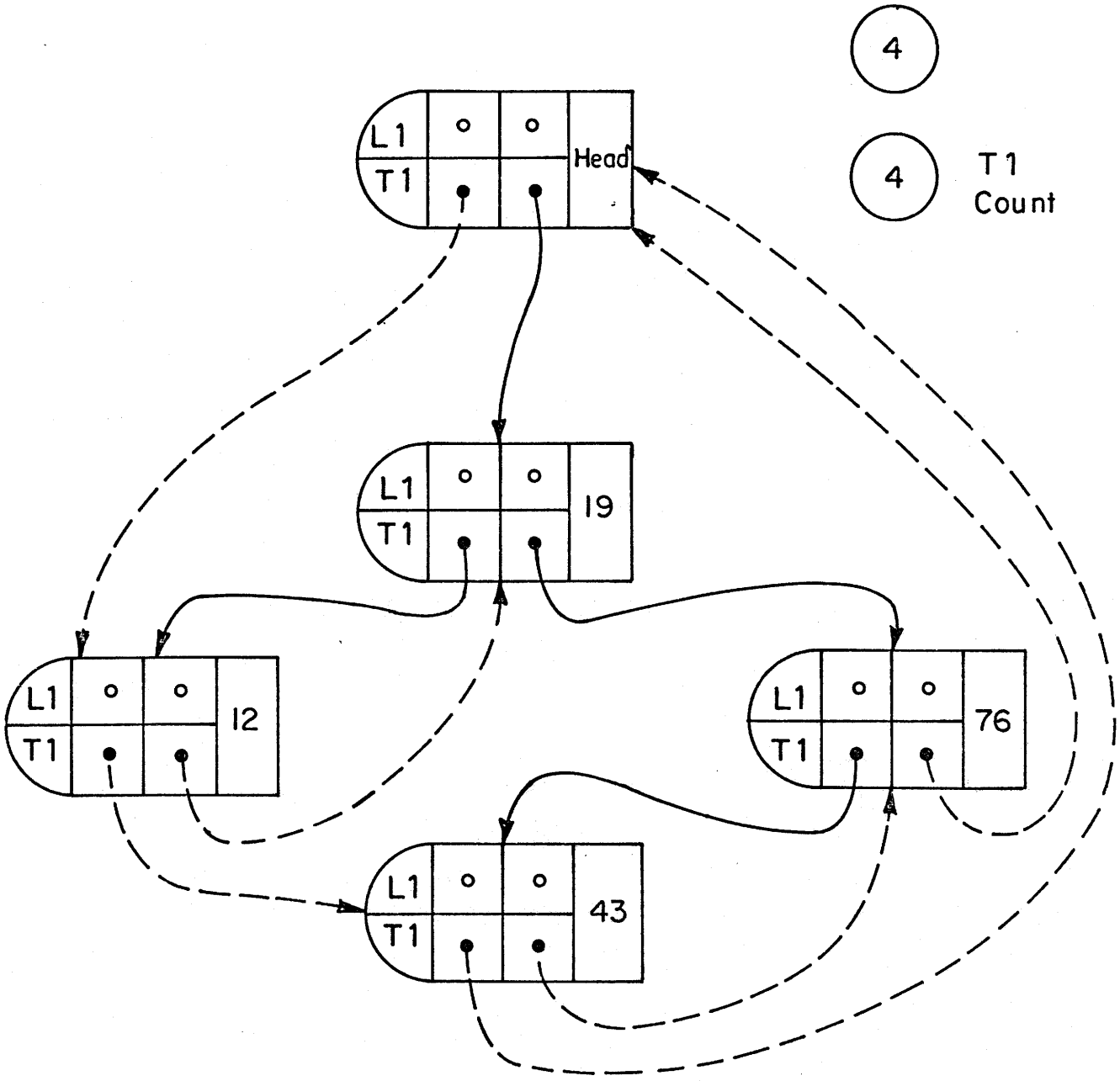


Fig. 4.2(a) Compound binary tree-linked list, showing tree pointers, threads and chains

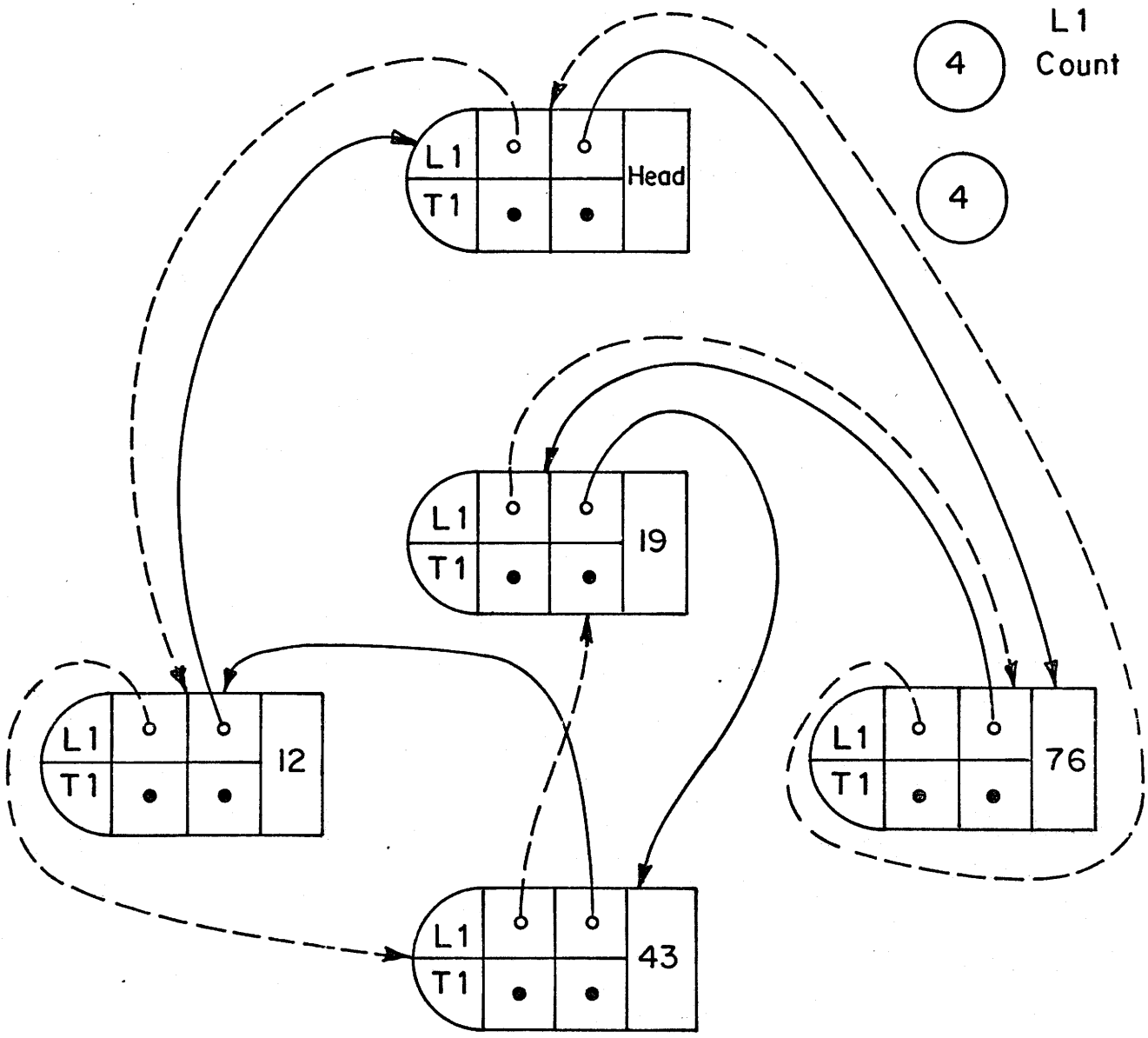


Fig. 4.2 (b) Compound binary tree - linked list, showing list pointers. Logical order (76, 19, 43, 12)

the construction of robust data structures.

5. SYNTHESIS OF ROBUST DATA STRUCTURES

We have demonstrated two methods of adding redundancy to improve detectability and correctability: an ad hoc method which adds identifier, pointer, and count fields to a given data structure in an attempt to improve robustness, and compounding data structures whose robustness is known. The first was exemplified by adding chain and thread links to "ordinary" binary trees, thus achieving 2-detectability and 1-correctability. The second was illustrated by compounding two double-linked lists, which increased the detectability from 2 to 5.

Given an arbitrary data structure, we also wish to consider increasing its robustness by compounding it with a simple structure such as a double-linked list. The list could be thought of as "roping" the overall structure together for greater resistance to error. Under what conditions is this approach reasonable, and can we gain more robustness by adding more strands of rope (i.e., by adding a second or even third double-linked list)?

The answers to these questions follow immediately from the calculation of the compound detectability. Because the ch-same's of the component structures do not add in the expression, the compound detectability must remain less than c, the original value of ch-same. Thus, if we consider

adding one or more double-linked lists (where ch-same = 6), the best we can do is increase the detectability to min(c, 6) - 1. In fact, by compounding two double-linked linear lists with the original structure, we are able to guarantee a detectability of min(c, 6) - 1, because of the addition of terms due to ch-repl and ch-diff of the linear lists. This implies that we do not increase the detectability by compounding more than two double-linked lists onto the original structure. Finally, if ch-same was the limiting term in the original structure, compounding it with any structure cannot increase the detectability (e.g., see the binary tree example in Section 4).

We have been considering ways of increasing robustness by the addition of structural redundancy to storage structures. We may also consider alternative ways of organizing a fixed quantity of structural redundancy. We loosely define the amount of structural redundancy in a storage structure to be the number, p, of edge-disjoint paths to each node. For fixed p, how can we vary the detectability and correctability as we choose various implementations of the same structure? While much work remains to be done in this area, we present two examples. For p = 2, we saw in the companion paper that modified(2) double-linked lists are 3-detectable, whereas double-linked lists are 2-detectable.

As a second example, we consider the compound structure

composed of two double-linked linear lists, for which p = 4. As we have shown, such an implementation is 2-correctable, and 5-detectable. However, by choosing a different pointer structure, we can find a linear list implementation with four edge-disjoint paths to each node which is 3-correctable and 8-detectable. Rather than choosing pointers from a node A to adjacent nodes in the structure, we choose four pointers to nodes at specified distances from A. ([3, Section 5.3] gives a general indication of how this may be done.) In both examples, the increase in detectability and correctability is matched by an increase in cost: execution time for insertions and deletions increases, and the insert and delete routines are more complex. We conjecture that an important difference between these two structures is the rate of error propagation. In the first structure, errors seem to propagate at the same rate as in double-linked lists; in the second, they propagate approximately four times as fast.

We conclude that the most critical quantity related to the detectability of a data structure implementation is ch-same, the number of changes required to rearrange the nodes in an instance of the data structure. If this value is small, no compound data structure formed from the given one can have a larger detectability. In this case, detectability might possibly be increased by an ad hoc addition of redundant structural information in the form of

extra pointers. We have also shown that compounding an arbitrary data structure with one or two double-linked lists may yield some improvement in detectability.

6. 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 storage structure 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.

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

Various costs are associated with the robust storage structures considered here. We identify three: additional storage requirements, increased processing time for insertions and deletions, and processing time to perform change detection. The processing time used in checking can be adjusted by varying the interval between executions of the detection procedure. However, making the interval too long may give an unacceptably high probability of multiple errors or undetectable error propagation.

The storage cost of an encoding (of a storage structure) can be considered to be made up of three parts: data content, structural information, and redundancy. The redundant data used in encodings discussed here is 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 9: 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 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 storage structure 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 storage structures, the number of changes performed by an update routine will always be much greater than the bound, but the variation between storage structures 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 10: If a storage structure 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 a storage structure, 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 unable to find the node in question by following a sequence of pointers from the header. []

This result means that, minimally, there must be r + 1 pointers to each node in an r-correctable storage structure. 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.

Finally, we may note an intuitive relationship between the three forms of redundancy being considered and the three properties ch-same, ch-repl, and ch-diff. We observe that adding identifier fields will increase ch-repl but will not affect ch-same or ch-diff. Similarly, adding one or more counts will increase ch-diff but will not affect ch-same or ch-repl. Adding redundant pointers to a storage structure, however, may increase all three of ch-same, ch-repl, and ch-diff.

We have shown that achieving very high detectability or correctability has serious performance implications. However, the empirical results of Part I indicate that the effective detectability of a storage structure can be higher than that which is analytically shown to be possible. This is true because the specific combinations of changes that are required to produce an undetectable error rarely occur in practice, but must be considered in a proof. Thus, the

double-linked list storage structure, which is 2-detectable in theory appears to be better than 12-detectable in practice. However, the single-linked list storage structure, which is 1-detectable in theory also appears to be 1-detectable in practice [3]. A "rule of thumb" is that for typical applications requiring fairly robust storage structures, a 2-detectable, 1-correctable storage structure should initially be assumed to be sufficient. Only if such a storage structure proves to be insufficiently robust in practice should a more robust storage structure be sought.

7. SUMMARY, CONCLUSIONS, AND FURTHER WORK

In these two papers, we have attempted to provide both an intuitive and a formal description of our approach to robust data structures. In Part II, we first developed a theory for evaluating the robustness of storage structures. The basic approach relied on the valid state hypothesis, although similar, but more complicated results can be obtained without it. The results presented here were extended to compound storage structures in Section 4. Sections 5 and 6 used the theoretical results and practical experience to motivate a general discussion of robust data structure synthesis, including the associated cost and effectiveness trade-offs.

Many areas for further work can be discerned. We used the General Correction Theorem (Theorem 7) in order to

determine the correctability of a compound implementation. However, given correction procedures for sub-structures, is it possible to deduce a correction procedure for the compound structure? What can be said if we remove the restriction that all nodes in the compound structure belong to each constituent structure? Does there exist a unified approach to robustness involving both content and structural data? What, in a practical sense, is a "reasonable" amount of redundancy, and how is the effective robustness under practical conditions related to the theoretical values of correctability and detectability?

One message is clear: a little redundancy, thoughtfully deployed and exploited, can yield significant benefits for fault tolerance; however, excessive or inappropriately applied redundancy is pointless. We have shown experimentally in the companion paper and [3] that the effective detectability of an implementation can be considerably higher than that which is guaranteed by the theory. The major goal of our research is to find where and how to apply redundancy to yield cost-effective fault tolerant systems.

BIBLIOGRAPHY

1. Bondy, J. A. and U. S. R. Murty. Graph Theory with Applications. London, Macmillan Press Ltd., 1976.

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

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

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO
TECHNICAL REPORTS 1979

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

* Out of print - contact author

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

* Out of print - contact author

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

* Out of print - contact author

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO

RESEARCH REPORTS 1980

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

* In preparation

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

* In preparation