ROBUST IMPLEMENTATIONS OF COMPOUND DATA STRUCTURES

David J. Taylor
James P. Black
David E. Morgan

Research Report CS-79-10

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

February 1979

ABSTRACT

One method for enhancing software fault-tolerance is to introduce redundancy in data structure implementations, then use this redundancy to detect and correct erroneous changes made to an instance of the data structure. Three commonly used forms of redundancy are considered here: storing a count of the number of nodes in a structure instance, using node type identifier fields, and using additional pointers.

In previous work, we have derived values for the detectability and correctability of simple data structures such as binary trees and lists with multiple links. We have also shown experimentally that the effective detectability of an implementation can be considerably higher than that which is guaranteed by the theory.

The purpose of this paper is to show how the detectability of a compound data structure implementation may be calculated. Our thesis is that a little redundancy, thoughtfully deployed and exploited, can yield significant benefits for fault-tolerance; however, excessive or inappropriately applied redundancy is pointless.

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

I. INTRODUCTION

One method for enhancing software fault tolerance is to introduce redundancy in data structure implementations, then use this redundancy to detect and correct erroneous changes made to an instance of the data structure. Three commonly used forms of structural redundancy are considered here: storing a count of the number of nodes in a structure instance, using node type identifier fields, and using additional pointers.

Previous results [7, 8, 9] have been concerned with the detectability and correctability of implementations of binary trees and linked lists, as well as ways of deriving these values for certain simple data structure implementations. In this paper, we extend the detectability results to certain combinations of simple data structures, which we call compound structures.

Section II presents basic terminology, and states fundamental results with outlines of the proofs. These results are applied to simple data structures in Section III. Section IV contains the definitions and results relating to compound data structures. Section V shows how these results may be applied to the synthesis of robust data structure implementations, and Section VI contains a summary, as well as conclusions and ideas for further work.

II. TERMINOLOGY AND BASIC THEORY

A. Terminology.

The following definitions will be used in discussing data structures. A data structure is defined to be a logical organisation of data. We define a data structure implementation to be a representation, on some storage medium, of a data structure. A data structure instance is a particular occurrence of a data structure implementation. Thus, "binary tree" is a data structure; a representation in which there are pointers from each node to the left and right sons of the node is an implementation of a binary tree; and if a particular set of data is stored according to this implementation, that is a data structure instance.

A change is defined to be an elementary modification to a data structure instance. The change may be erroneous, or part of a correct update. To illustrate this definition, consider the following implementation of a linear list. Suppose the list contains four items, each of the first three has a pointer to the next, and the last contains a null pointer:

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

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

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

This single change effectively replaces D by X.

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

If all sets of N or fewer changes can be detected, we say the implementation is N-detectable. Similarly, if all sets of N or fewer changes can be corrected, we say the implementation is N-correctable. These definitions of detectability and correctability are closely related to Hamming's definitions for binary codes [5]. (Note that N-detectability implies K-detectability for K < N, and similarly for correctability.)

A robust data structure implementation is an implementation containing redundant structural data which allows erroneous changes to be detected, and possibly corrected as well. Thus, the robustness of a data structure implementation is defined in terms of its detectability and correctability.

In order to exploit the redundancy in data structure implementations, it is necessary to have procedures which perform error detection and possibly correction. Such

procedures, sometimes called audits, have been successfully used for some time in applications requiring high degrees of fault tolerance. The classical examples are concerned with automated telephone switching control [2, 3, 4]. In these cases, audit programs capable of varying degrees of detection and/or correction are run periodically, when failures occur, or when trouble is suspected. While we are not immediately concerned with the details of such an audit system in this paper, a practical application of our results would almost certainly use this type of technique.

B. Basic Theory.

We define three properties of a data structure implementation which are useful as intermediate steps in calculating its detectability: ch-same, ch-repl, and ch-diff. Ch-same is the minimum number of changes required to transform one correct data structure instance into another correct instance consisting of the same set of nodes, i.e., the nodes have been restructured. In the example given above, ch-same is three, as three changes are required to change the list

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

into the list

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

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. In the example of Section I, a single change replaced D by X. Similarly, ch-diff is the minimum number of changes required to change a correct instance into a correct instance with a different number of nodes. In the example, one change is required to replace an arbitrary part of the list with NULL:

A -> B -> NULL.

With these definitions, the following result is trivial.

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

Several detectability results are related to the concepts k-determined and k-count-determined. These may be defined informally as follows. An implementation is k-determined if the pointers in each instance of the implementation can be 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.) An implementation is k-count-determined if the pointers in each instance of the implementation can be partitioned into k disjoint sets, such that each set can be used to calculate the number of nodes in the instance.

The double-linked list implementation shown in Figure 2.1 contains forward and back pointers in each node and thus is 2-determined. The modified(2) double-linked list (see Section III) of Figure 2.2, in which the back pointer points to the second preceding node, is 2-determined but 3-count-determined. Either set of back pointers, as well as the forward pointers, may be used to determine the number of nodes in the instance.

Theorem 2. If each of the k sets of pointers in a k-determined implementation contains only one pointer to each node, if m of those sets have exactly one pointer in each node, and if there are a minimum of n identifier fields per node, then

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

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. To insert a foreign node, we must change at least one pointer in each of the k sets and we must insert n identifier fields in the foreign node. For m of the k sets there must be an equal number of pointers entering and leaving any set of nodes, so there must be at least m pointers from foreign nodes to nodes in the unchanged structure. So, the minimum number of changes to

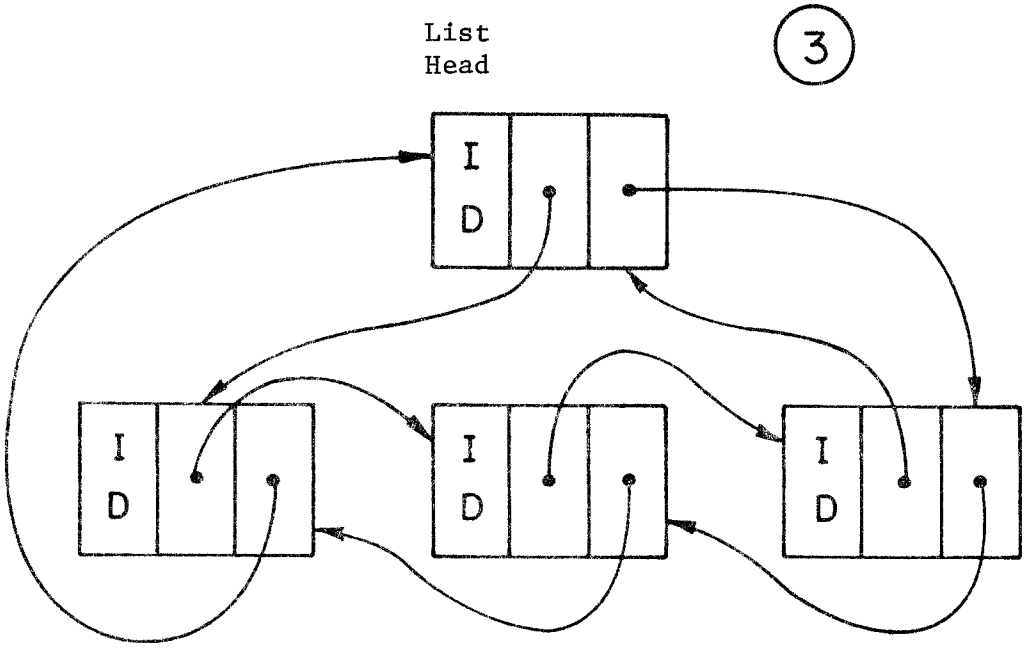


Figure 2.1(a)

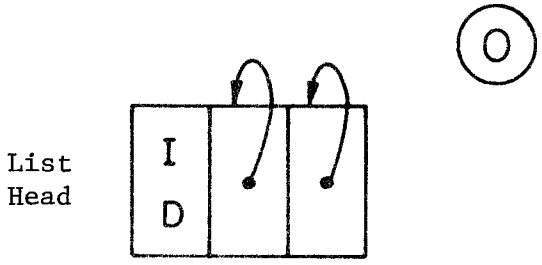


Figure 2.1(b) Double-Linked List Implementation

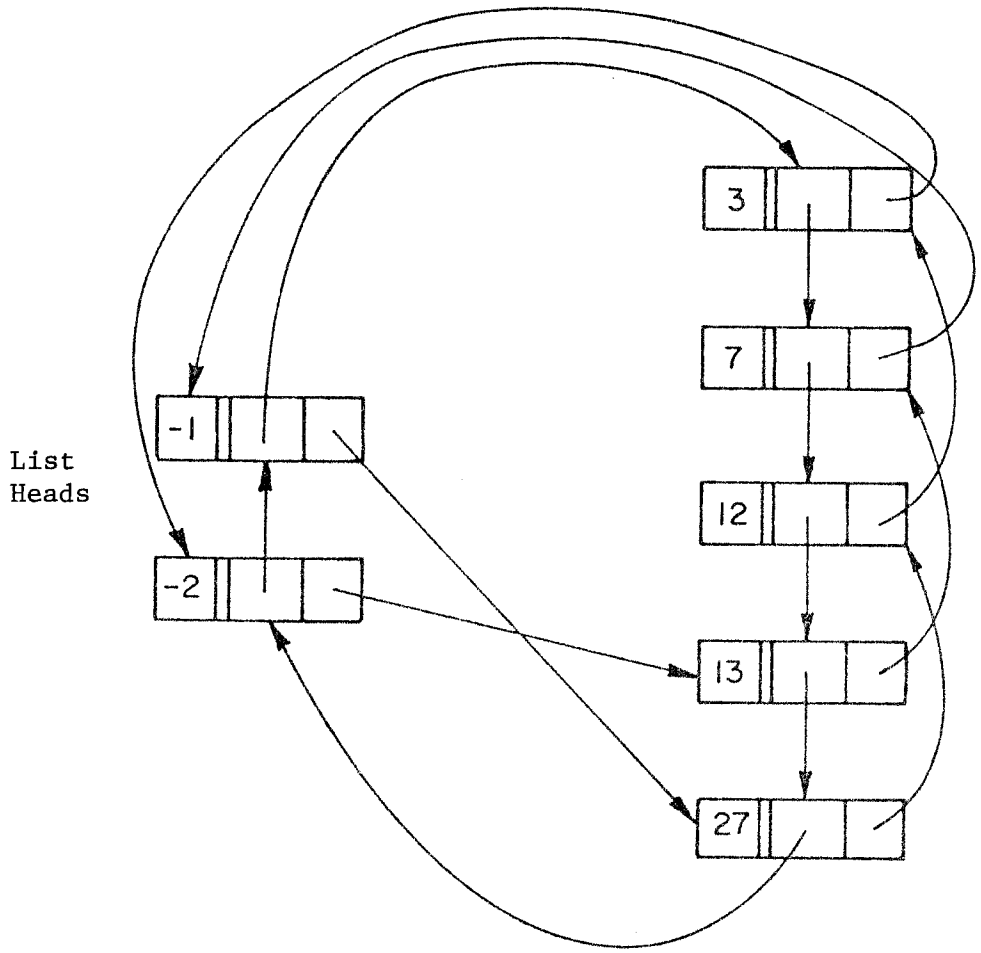


Figure 2.2 Modified (2) Double Linked List

insert one or more foreign nodes is k + n + m, which gives

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

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

We thus have

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

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

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. A counterexample is the following.

Referring to Figure 2.3, suppose a linear list 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

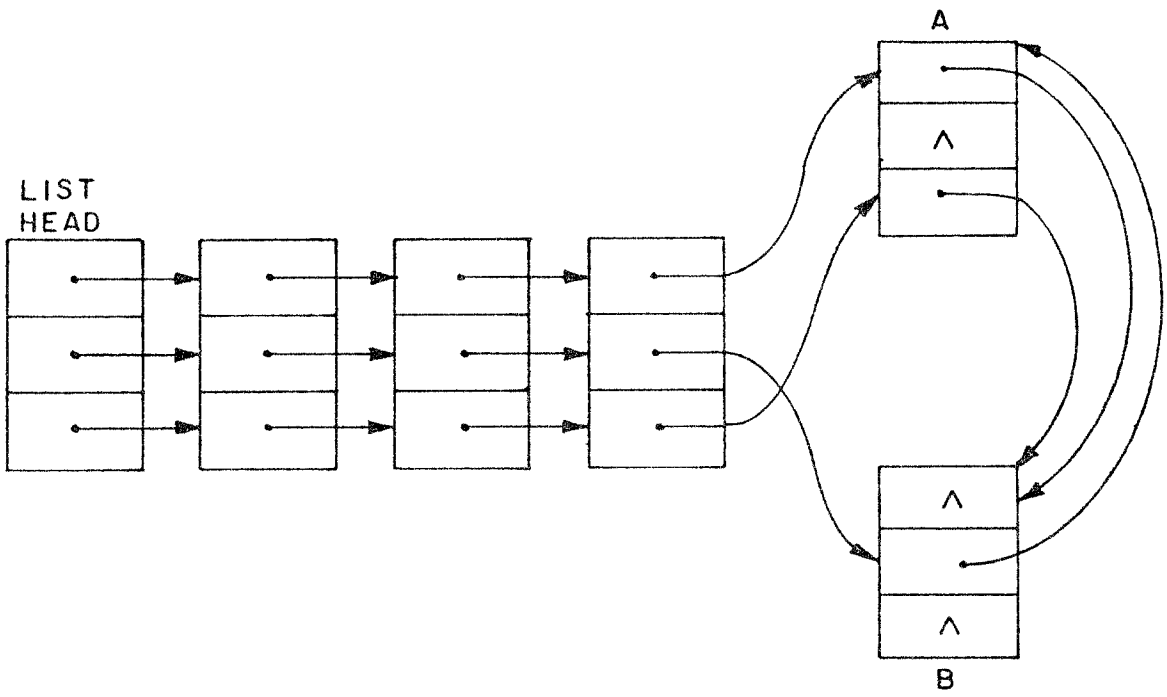


Figure 2.3 Counter Example for Revised ch-repl

an implementation is 3-determined, and has m = Ø under the original definition. If there is a stored count but no identifier fields, then n = Ø, 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.

As an example of the theorem, consider double-linked lists. For this implementation, k = 2, m = 2, n = 1; thus ch-same $\geq$ 4, and ch-repl $\geq$ 5. In fact, ch-same = 6, and ch-repl = 5. While we do not prove this, Figures 2.4 and 2.5 demonstrate this result informally.

Theorem 2 allows ch-same and ch-repl to be calculated. We now prove a theorem which will allow us to calculate ch-diff so we can use Theorem 1.

Theorem 3. If a k-count-determined implementation has j stored counts, ch-diff $\geq$ k + j.

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

For double-linked lists, k = 2 and j = 1, so ch-diff $\geq$ 3. (In fact, ch-diff = 3. For example, changing any list into an empty list requires changing the two head pointers and the count.) Applying Theorem 1 to the values obtained

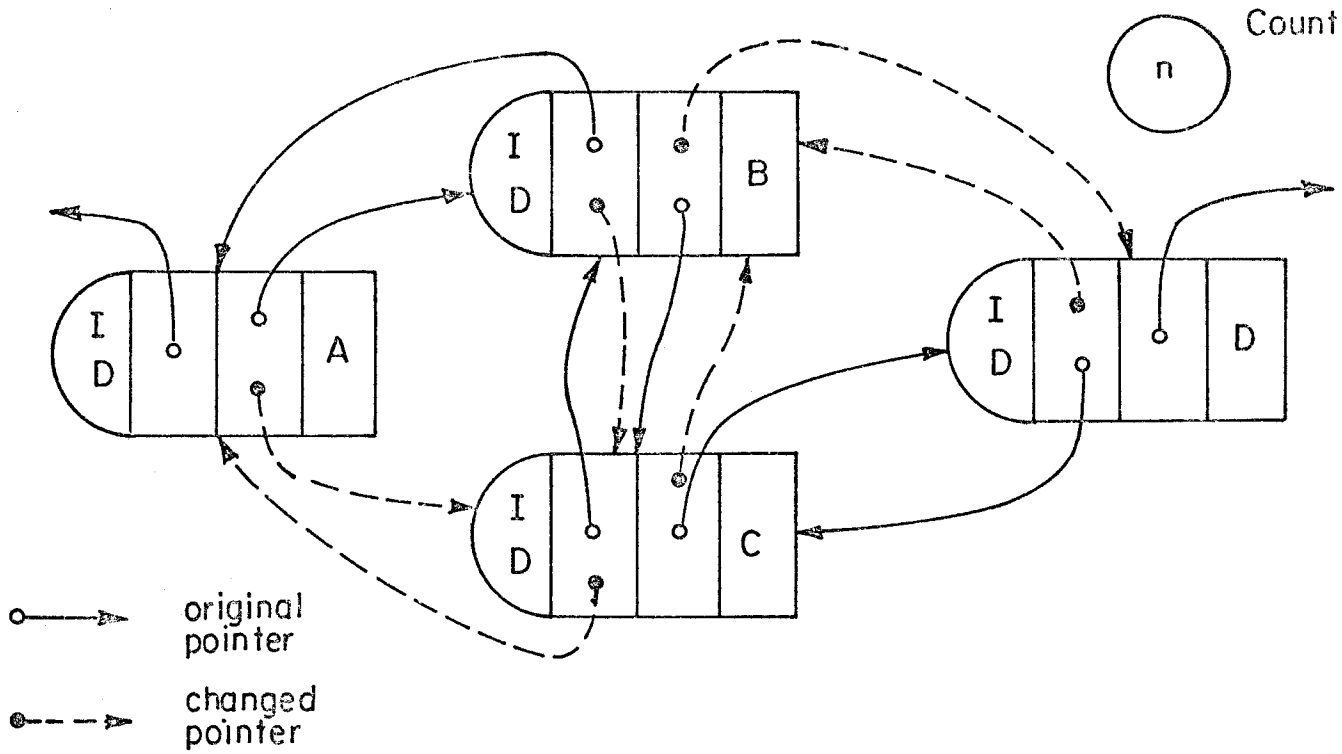


Fig. 2.4 ch-same = 6 for double linked lists

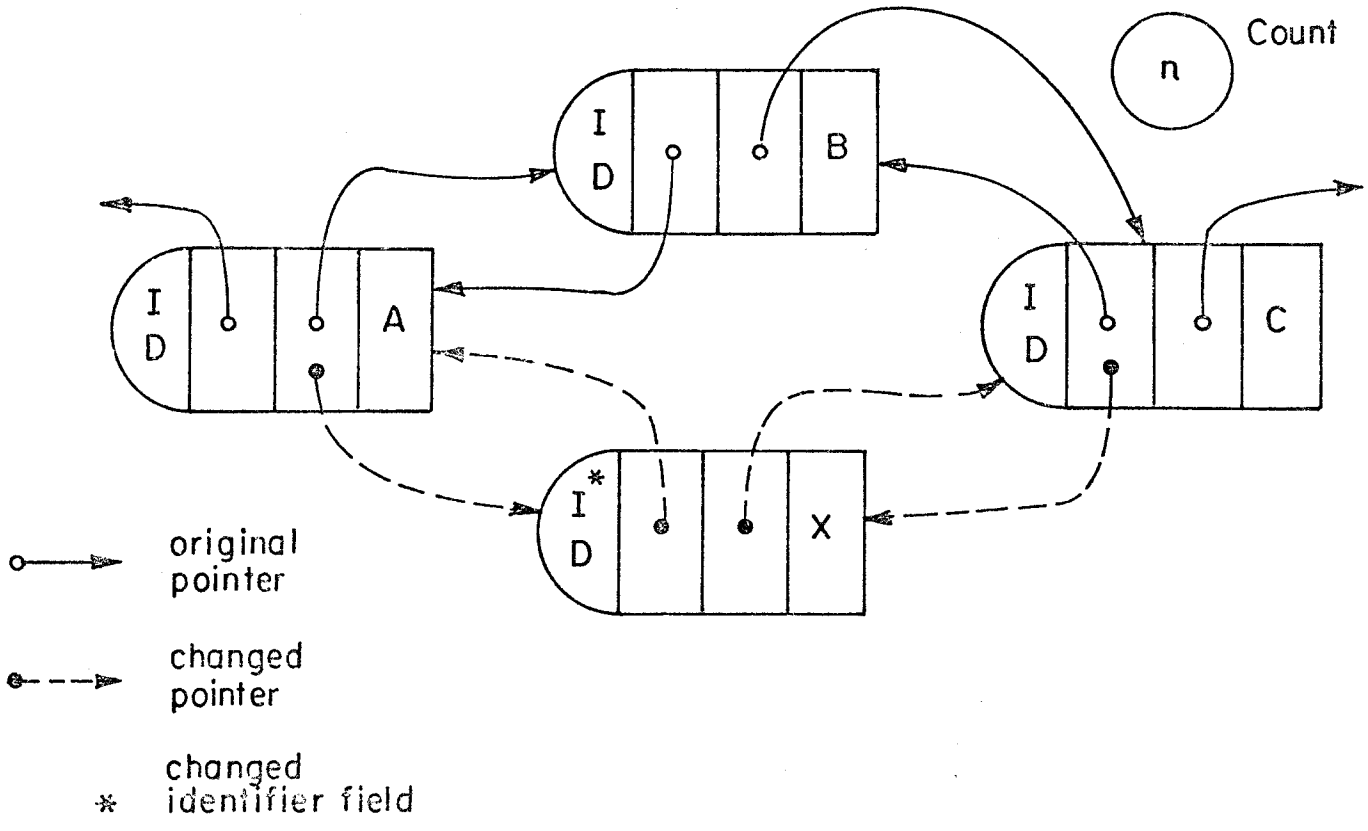


Fig. 2.5 ch-repl = 5 for double linked lists

for ch-same, ch-repl, and ch-diff, we conclude that double-linked lists are (at least) 2-detectable.

We present the following result without proof, as the proof is quite lengthy. (See [9], Section 4 for the proof.)

Theorem 4 (General Correction Theorem). If a data structure implementation employing identifier fields is 2r-detectable and there are at least r+1 edge-disjoint paths to each node of the structure, then the implementation is r-correctable.

Applying this result to the double-linked list implementation discussed above, we have that it is 1-correctable.

III. APPLICATIONS TO SIMPLE DATA STRUCTURES.

In this section, we apply the theory developed above to novel implementations of a linear list and a binary tree.

We first consider an implementation of a linear list which is similar to the double-linked one, but in which each "backward" pointer points to the second preceding node rather than the immediately preceding node. The storage required per node is clearly the same as for a double-linked list, but one more change is required when inserting a node. (Three backward pointers must be changed, rather than two.) This implementation, which is referred to as a "modified(2) double-linked list", is 3-detectable and 1-correctable.

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

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

For a second example, we consider binary trees. A common implementation is the "right-threaded" binary tree, in which each node with a null right sub-tree contains a pointer to the node which is its in-order successor. If such an implementation contains identifier fields and a stored count, then it is Ø-correctable and 1-detectable.

We would like to obtain a 1-correctable implementation of a binary tree. To satisfy the hypothesis of Theorem 4, there must be two edge-disjoint paths to each node of a

structure. This clearly implies that there be at least two pointers to each node, a condition which does not hold for simple binary trees. In a right-threaded tree, we note that each node has exactly one non-thread link pointing to it and either zero or one thread links pointing to it. In fact, a node has a thread link pointing to it iff it has a non-null left sub-tree. (If the left sub-tree is non-null, the final in-order node in the sub-tree contains a thread to the node in question.) Thus, nodes with null left links have only one incoming edge, so an obvious possibility is to link these nodes together, using the left link field. A tag must be added to each node indicating the use of the left link, and the list head must now contain a pointer to the "first" node with a null left link. The nodes could be linked in any order, but for our purposes, in-order will be most convenient.

This structure will be called a chained and threaded binary tree. The nodes with logically null left links, joined in in-order, will be called the chain, and the links joining them will be called chain links. The 2-detectability is proven by exhibiting a detection procedure and demonstrating its correctness. The 1-correctability then follows, as there are two edge-disjoint paths to each node: one using the normal left and right pointers, and one using only chains and threads. (Figure 3.1 gives a procedure to visit all nodes in the tree

```
procedure SCAN-CT(T) begin
    pointer T, pointer CH, pointer P;
    CH <- LEFT(T);  /*  first chain node  */
    while (CH ≠ T) do
    begin
        P <- CH;
        visit P;
        while (RTAG(P) = "THREAD") do
        begin
            P <- RIGHT(P);
            visit P;
        end
        CH <- LEFT(CH);
    end end
```

Figure 3.1 Inorder traversal using chains and threads

in in-order, using only chains and threads.) While we do not prove these results, Figure 3.2 shows two examples of undetectable triple changes, implying that such an implementation is at most 2-detectable.

IV. APPLICATION TO COMPOUND STRUCTURES.

An instance of a compound data structure implementation, compounded from implementations M(1), M(2), ..., M(s), is defined as follows:

1. The structural information in every instance of the compound structure (i.e., pointer and identifier fields) may be partitioned into s disjoint subsets.
2. The i-th set, with its list head and possible stored counts, forms a data structure instance of M(i), for i = 1, 2, ..., s.
3. Excluding the list heads of the s instances, each data structure instance contains the same set of nodes.

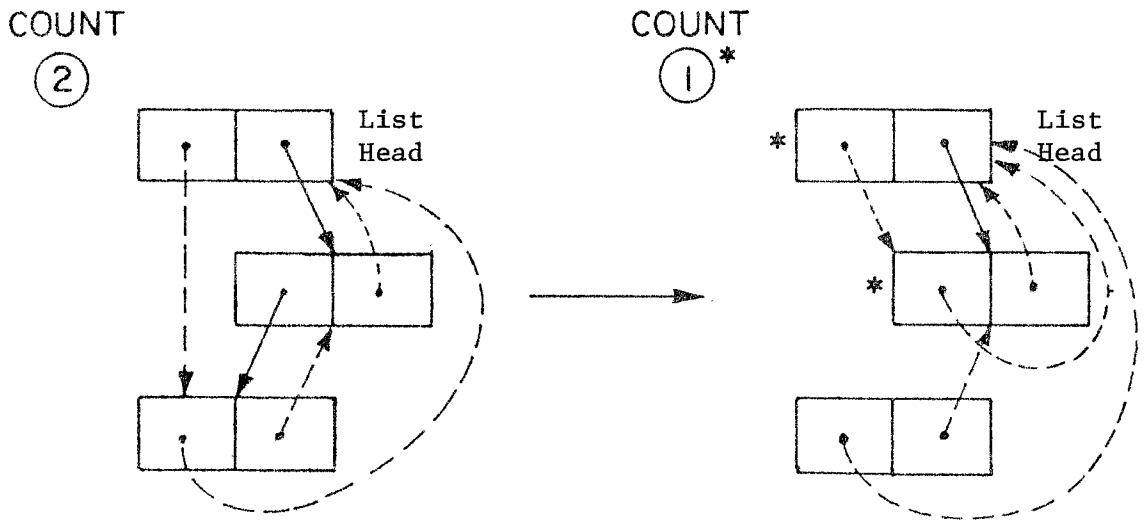
We now compute ch-same, ch-repl, and ch-diff for a compound structure. For simplicity, we consider a compound data structure composed of two sub-structures, and we assume ch-same, ch-repl, and ch-diff are known for each of the sub-structures.

1. 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:

ch-same = min (ch-same(1), ch-same(2)).

2. Ch-repl. In order to replace some number of nodes

(a) Ch-diff type of change



(b) Ch-same type of change

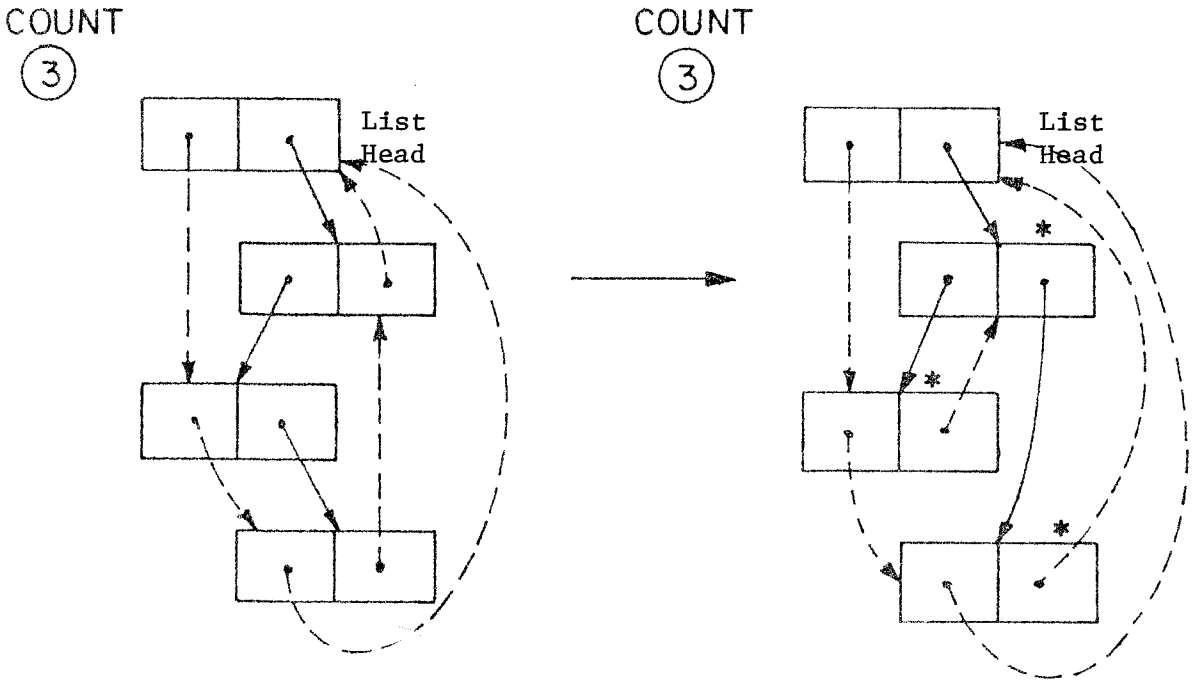


Figure 3.2 Undetectable Triple Changes

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

ch-repl = ch-repl(1) + ch-repl(2).

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

ch-diff = ch-diff(1) + ch-diff(2).

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

Theorem 5. The detectability of a compound structure is

min (ch-same(1), ch-same(2),
ch-repl(1) + ch-repl(2),
ch-diff(1) + ch-diff(2)) - 1.

The obvious generalization to a compound data structure with n sub-structures is:

min (ch-same(1), ch-same(2), ..., ch-same(n),
ch-repl(1) + ch-repl(2) + ... + ch-repl(n),
ch-diff(1) + ch-diff(2) + ... + 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 4 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 can show that ch-same = 3, ch-repl = 5, and ch-diff = 3. Figure 3.2 gives examples of the limiting values for ch-same and ch-diff. In order to replace one or more node(s) with foreign nodes (ch-repl), we must change two incoming pointers, two outgoing pointers, and an identifier field. 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 the construction of robust data structures.

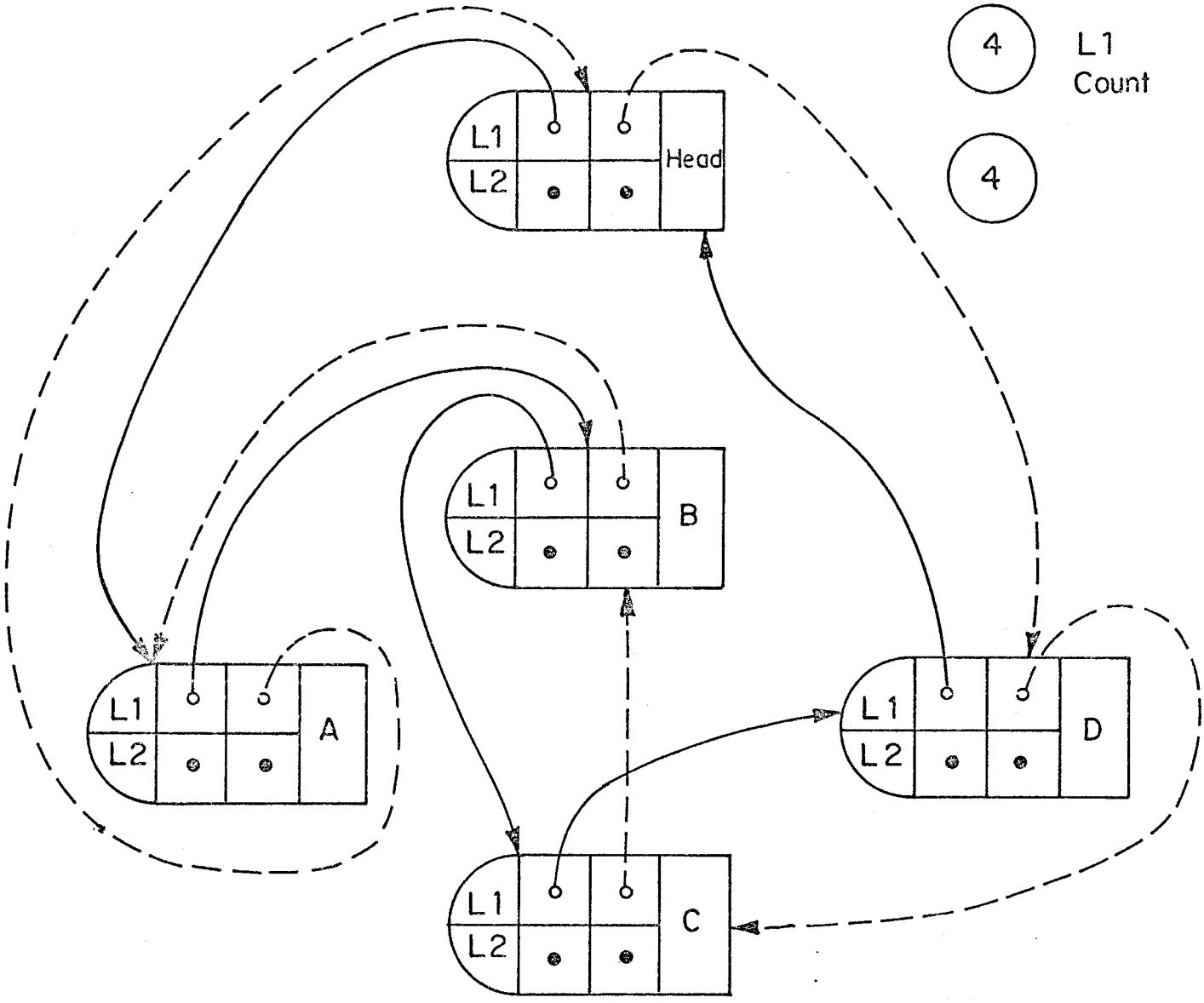


Fig. 4.1 (a) Compound structure showing first list (L1) pointers. Logical order (A, B, C, D)

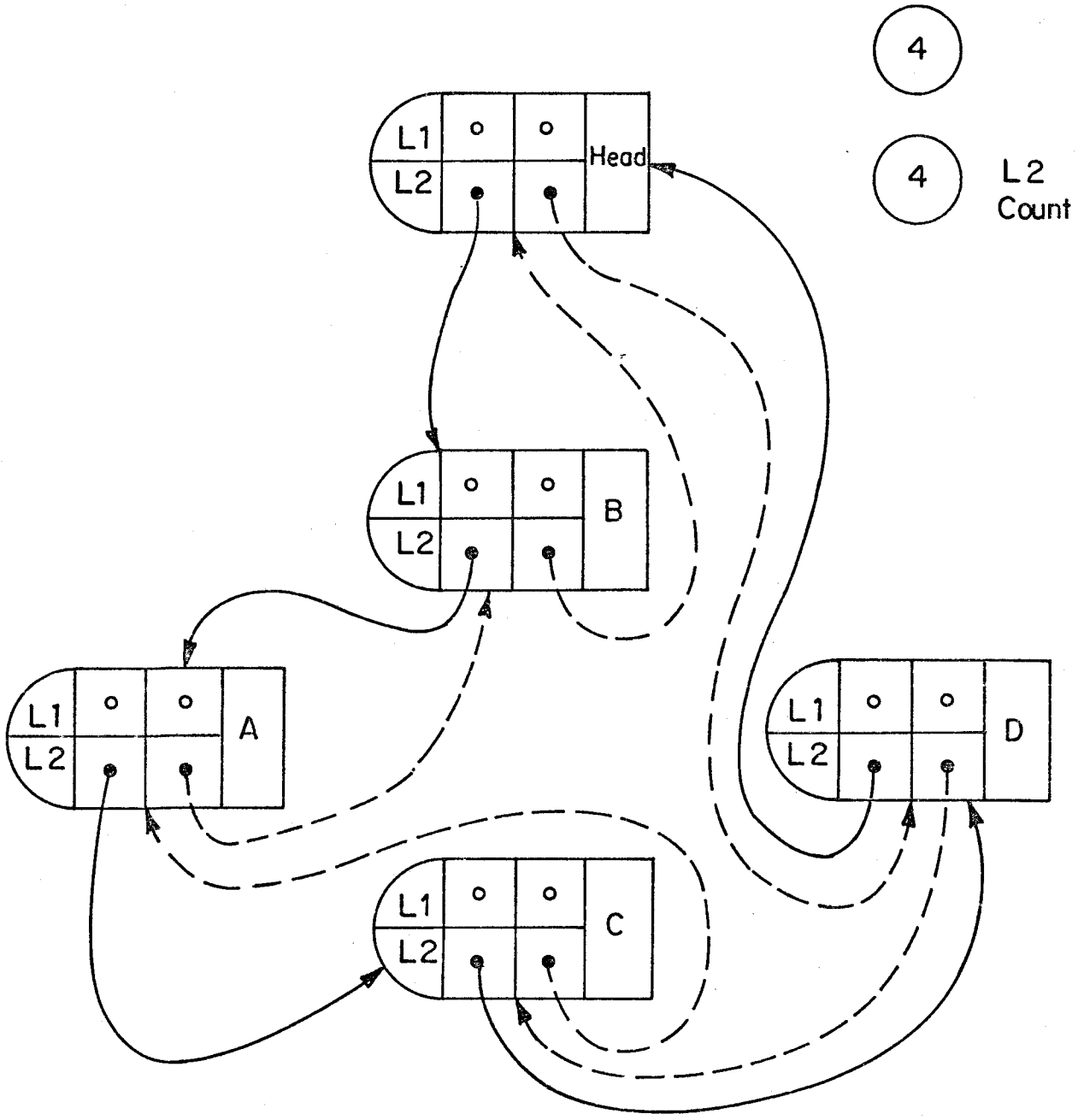


Fig. 4.1 (b) Compound structure showing second list (L2) pointers. Logical order (B,A,C,D)

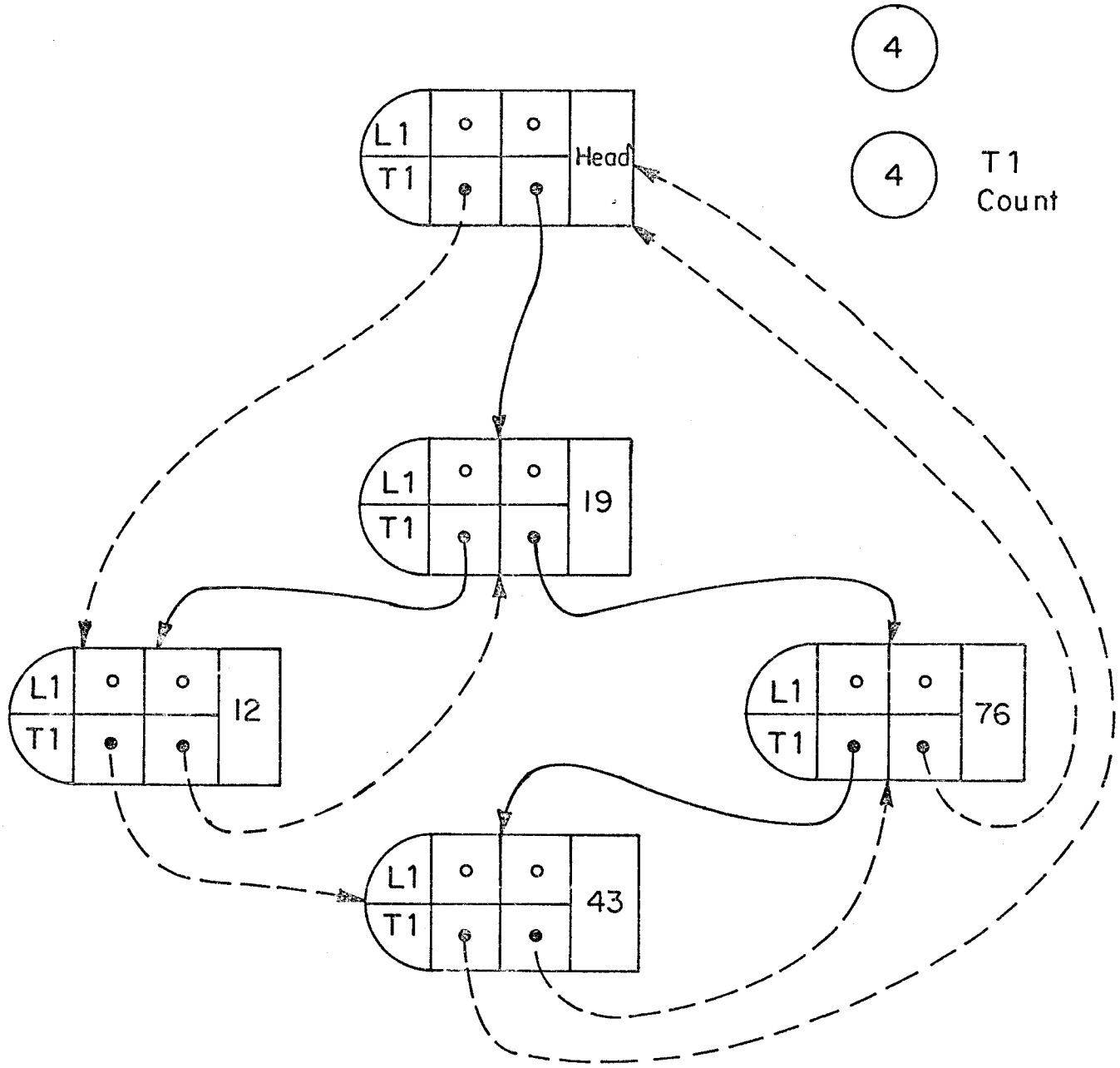


Fig. 4.2(a) Compound binary tree-linked list, showing tree pointers, threads and chains

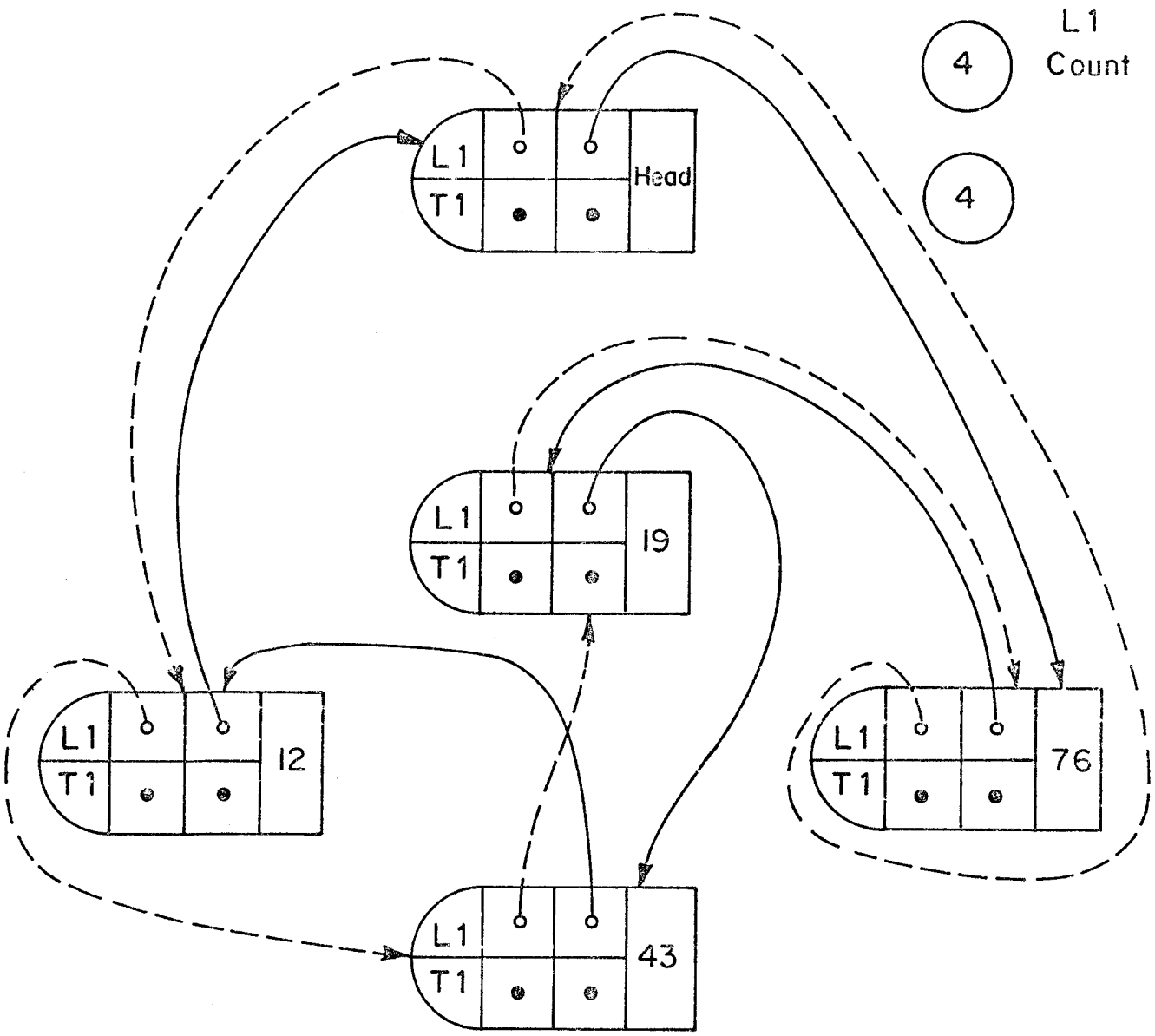


Fig. 4.2 (b) Compound binary tree - linked list, showing list pointers. Logical order (76,19,43,12)

V. 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. 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 IV).

We have been considering ways of increasing robustness by the addition of structural redundancy to implementations. We may also consider alternative ways of organizing a fixed quantity of structural redundancy. We loosely define the amount of structural redundancy in a data structure implementation 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 Section III 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. ([8, 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.

VI. SUMMARY, CONCLUSIONS, AND FURTHER WORK.

After a review of basic definitions and results, we presented our definition of compound data structures, and used the definition to calculate the detectability of a compound data structure implementation. The detectability was defined in terms of ch-same, ch-repl, and ch-diff for the component sub-structures. After illustrating the theoretical results with examples of linear list and binary tree implementations, we made some general remarks on compound data structure synthesis, and gave some indication of the associated cost-effectiveness tradeoffs.

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.

Many areas for further work can be discerned. We used the General Correction Theorem (Theorem 4) 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 [7, 8] 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.

ACKNOWLEDGEMENTS

Professor F. W. Tompa provided helpful comments on the development of the results in this paper. The research was supported by the Natural Sciences and Engineering Research Council of Canada under grants A3078 and A8116.

BIBLIOGRAPHY

1. Avizienis, A. Fault-tolerance and fault-intolerance: Proceedings, International Conference on Reliable Software, April 21-23, 1975, Los Angeles, California. (Published as SIGPLAN Notices, Vol. 10, No. 6, June 1975.) pp458-464.

2. Beuscher, H. J., et al. Administration and Maintenance Plan, Bell System Technical Journal, Vol. 48, Oct. 1969. pp2765-2815.

3. Bowman, P. W., et al. 1A Processor: Maintenance Software, Bell System Technical Journal, Vol. 5, No. 2, Feb. 1977. pp255-287.

4. Downing, R. W., et al. No. 1 ESS Maintenance Plan, Bell System Technical Journal, Vol. 43, Sept. 1964. pp1961-2019.

5. Hamming, R. W., Error Detecting and Correcting Codes, Bell System Technical Journal, Vol. 26, No. 2, April 1950. pp147-160.

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

7. Taylor, D. J., and Morgan, D. E. Detectability and Correctability of Data Structure Implementations, submitted to Transactions on Programming Languages and Systems. Also available as Computer Science Research Report, CS-78-51, University of Waterloo, Waterloo, Ontario, Canada; November 1978.

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

9. Taylor, D. J. Theoretical Foundations for Robust Data Structure Implementations, submitted to the Journal of ACM. Also available as Computer Science Research Report, CS-78-52, University of Waterloo, Waterloo, Ontario, Canada; December 1978.