PRINCIPLES OF DATA STRUCTURE ERROR CORRECTION

David J. Taylor
James P. Black

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

Research Report CS-81-30

September 1981

Abstract

Error correction in robust data structures is a difficult problem. Several correction algorithms have been examined to determine common design features which may prove useful in the design of correction algorithms for other structures. This paper presents a summary of the algorithms studied and the design principles which were derived. The paper is not a "cookbook" for constructing error correction algorithms, but should prove useful to those designing such algorithms.

1. INTRODUCTION

In previous papers [1, 2, 6, 7], we have described methods for analysing the robustness of representations of data structures. The two principle robustness properties of data structure representations are the number of errors which can always be detected (detectability) and the number of errors which can be corrected (correctability). In a sense, each of these properties implies a procedure: an error detection procedure for detectability, a correction procedure for correctability. In general, writing a detection procedure for a structure is not tremendously difficult. The conceptual difficulties are generally no worse than, for example, the problem of inserting a node into the structure. This is not the situation with correction procedures: they are usually very hard to design.

Because correction procedures present such great difficulties, we have less experience with them than with detection procedures. Indeed, until recently the experience was extremely limited: the brief sections on error correction in [6, 7] included virtually everything then known. Recently we have obtained new insights into the design of error correction procedures; this paper attempts to describe those insights.

In the remaining sections of the paper, we first present several examples of error correction routines. Then, in Section 3, we abstract from these examples a number of useful general principles. In Section 4, we describe an example where the principles don't apply. Section 5 contains a discussion of storage structure design principles and Section 6 contains a summary and areas for further work.

Before proceeding to the examples, one cautionary remark must be made. Although any system which is to be fault tolerant must contain some form of error detection, it need not necessarily contain correction procedures of the kind discussed here. When an error is detected there are at least four fundamental possibilities: halt, use backward error recovery, use forward error recovery, or ignore the problem. The last possibility is occasionally (rarely) a good idea. However, halting to allow manual intervention is appropriate on many systems and backward error recovery should be the first alternative considered if halting is unacceptable. Backward error recovery can consist of a checkpoint and rollback mechanism or take a more sophisticated form, such as recovery blocks [4, 5]. Because of its complexity, forward error recovery, such as described here, should be used only when the other three alternatives are unacceptable. The usual example of a system requiring forward error recovery is a real-time system in which backward error recovery cannot be done because rollback past an interaction with the external environment is unacceptable.

2. CORRECTION EXAMPLES

This section describes six correction algorithms for different storage structures or classes of storage structures. The first two have been described in previous papers, so their descriptions here will be quite brief. Descriptions of the storage structures referred to in this section are contained in the appendix. We are confident that the algorithms described here do correct errors as specified. Each algorithm has been validated either by a proof of its error correction properties or by extensive testing of an implementation of the algorithm. Testing was carried out both by introducing errors automatically at random and manually introducing errors to exercise all parts of the algorithm.

Example 1: General Correction Algorithm

One of the basic theorems of data structure robustness states (essentially) that 2r-detectability and r+1 edge-disjoint paths to each node imply r-correctability. The proof of this theorem [7] uses the General Correction Algorithm, which can perform the required r-correction. The algorithm works by first performing a depth-first search of the data structure instance, using the presence of identifier fields to prevent the examination of an unbounded number of nodes which are not part of the correct storage structure instance. Then the algorithm makes guesses at possible corrections and uses a detection procedure to check each guess. Because the guesses are essentially random, the execution time of the General Correction Algorithm is very slow. If the detection procedure is linear time, then correcting an instance of n nodes may take time O(n**(2r+1)).

Because the General Correction Algorithm is only useful as a theoretical device for proving correctability, more efficient special-purpose correction algorithms have been designed for various storage structures. Each of the following algorithms has a worst-case execution time which is linear in the size of the instance being corrected.

Example 2: Linear Lists, Single Error Correction

The first linear-time correction algorithm was devised for double-linked lists. A pseudo-code version of this algorithm is in [6]. A minor modification of this algorithm yields a 1-correction algorithm for modified(k) double-linked lists. Because it has been published elsewhere, we limit the discussion here to remarking that this is a particularly simple correction algorithm. A forward and a backward scan to the point of error makes possible a small constant number of tests to determine which field is in error and what its value should be.

Example 3: Linear Lists, Local Error Correction

While it is only possible to guarantee the correction of a single error in a modified(k) double-linked list, many cases of multiple errors can also be corrected. In fact, it appears that for $k > 1$, any set of "well separated" errors can be corrected. This is referred to as "local error correctability." For $k = 1$ (a standard double-linked list), local error correction is impossible because changing a forward pointer in node X and a backward pointer in node Y, which follows node X, results in all paths to nodes between X and Y being lost, preventing correction. This same phenomenon can occur for $k > 1$ only if both pointers to a node are changed or a set of k consecutive back pointers is changed: neither of these represent well separated changes.

Before describing the local error correction algorithm for modified(2) double-linked lists, it must be pointed out that precise general definitions for "well separated" and "local error correction" have not yet been developed. The algorithm given here should, at present, be considered simply a particular correction algorithm which can correct certain well-defined sets of errors. It is hoped that the specified sets of errors correspond to the reader's intuitive understanding of "well separated."

To understand the operation of the local correction algorithm for modified(2) double-linked lists, it is useful to draw the list structure in an unusual way. In Figure 1, we show a list with the two header nodes at the bottom of the figure and the nodes reachable by following back pointers in a column above each header. The forward pointers then form a zig-zag pattern running between the two columns. For convenience, the nodes are numbered in reverse order, since the correction algorithm traverses the list backwards.

The basic idea of the local correction routine is to have two parallel traversals of the list, one using the column of nodes above H1, the other using the column of nodes above H0. Let us refer to these as the left and right traversals, respectively. Ignoring details of initialisation, assume each traversal has reached node 1, the right traversal by H0 -> 1, the left by H1 -> 2 -> 1. Now to reach node 2, the left traversal "retreats" to 2: no pointer need be followed since 2 was already visited on the way to 1. The right traversal must follow pointers from 1 to 3 and then 3 to 2. Then, to reach node 3, the right

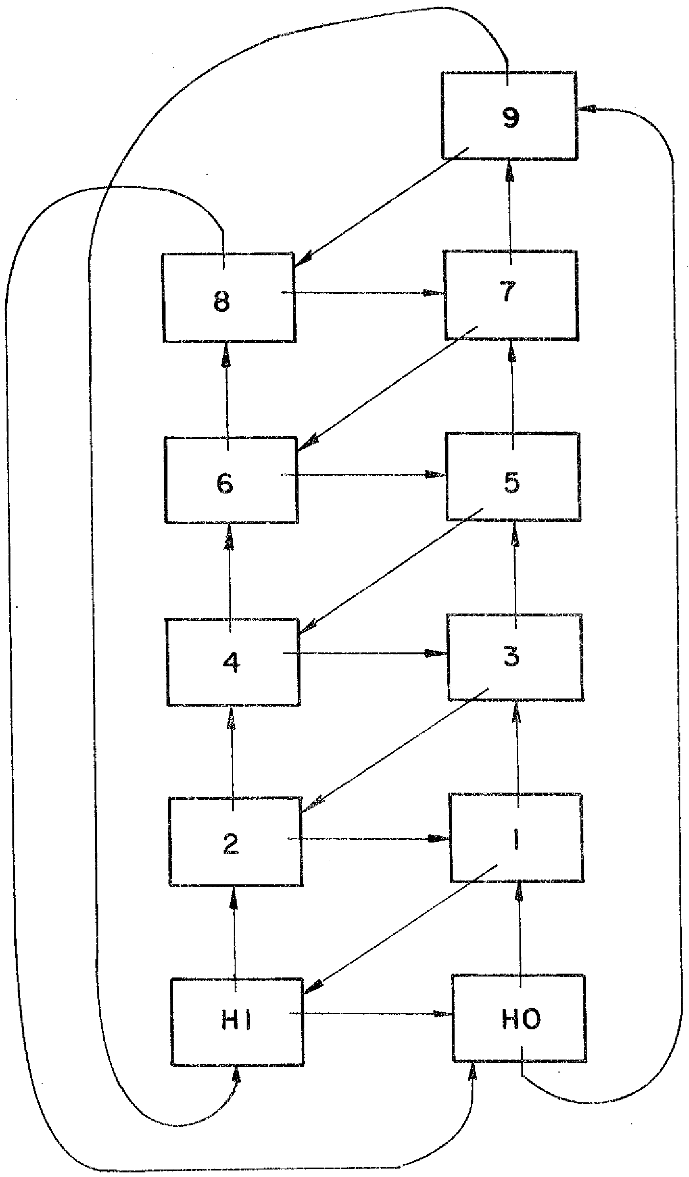

Figure 1

Modified(2) Double-linked list

traversal retreats and the left traversal follows 2 -> 4 -> 3. It should now be clear how the two traversal routines can individually enumerate all the nodes in reverse order, if the structure is undamaged.

Now consider what happens when an error is encountered. As an example, suppose the backward ponter in node 2 is changed from 4 to 8. (In the following discussion we simply assume various pointers are correct and at the conclusion enumerate the exact set which must be correct to allow the correction to take place.) In this example, everything will be fine as nodes 1 and 2 are enumerated, but when we advance from node 2, the left routine will reach node 7 while the right routine will reach node 3. Because the right routine did not follow any pointers, the error must be in a pointer followed by the left routine, either 2 -> 8 or 8 -> 7. To distinguish between these cases, we note that a forward pointer error affects only one node in the traversal, but a backward pointer error affects all following nodes. So, we advance once more: the left routine retreats to 8, the right routine follows 3 -> 5 and 5 -> 4. We still have a disagreement, so the error was in 2 -> 8, and we now know that the backward pointer in node 2 should point to node 4, the node just reached by the right traversal.

The question is: which pointers must be correct to allow this to take place? Of course, errors preceding this one must already have been corrected, and this depends on separation of this error from those previous ones, but that possibility will already have been examined with respect to the preceding errors. Thus, we require the backward pointer in node 3, the forward pointer in node 5, and the forward pointer in node 8 to be correct. (Actually, the pointer in node 8 can be incorrect, but if it is incorrect and points to node 3, that causes a severe problem.) Depending on how the algorithm reacts to incorrect identifier fields, we might also have to require some correct identifier fields, but this is not essential.

The analysis for forward pointer errors is very similar to the above, and identifier field errors and count errors can also be handled in the context of this algorithm. Thus, we have an algorithm which corrects an arbitrarily large number of errors, subject to a simple separation constraint, and accomplishes all this in linear time.

To make this example more concrete, assume that each traversal "balks" when it encounters a bad identifier field, simply returning the address of the node containing the bad identifier field, and that each traversal indicates not only the next node but a status. Possible status values are: O.K.--everything seems correct; TERM--end of list reached, everything seems correct; IDERR--bad identifier field encountered; PTRERR--illegal pointer encountered. The correction routine can then use a "fault dictionary" as shown in Figure 2 to select a diagnosis routine. The

If returned addresses are equal:

	O.K.	TERM	IDERR	PTRERR
O.K.	next	error	error	error
TERM	error	trm_ck	error	error
IDERR	error	error	error	error
PTRERR	error	error	error	error

If returned addresses are unequal:

	O.K.	TERM	IDERR	PTRERR
O.K.	adv_ck	tr_adv	id_ck	ch_ptr
TERM	tr_adv	error	error	error
IDERR	id_ck	error	error	error
PTRERR	ch_ptr	error	error	error

Figure 2

Fault dictionary for modified(2) local correction routine diagnosis routines are: next--everything is correct, proceed; term--correction completed; adv_ck--advance again and compare (as described above); tr_adv--similar to adv_ck but after correction check for termination of traversal; id_ck--use other traversal routine to determine whether error caused by bad pointer or bad identifier field; ch_ptr--change pointer using information from other traversal routine; error--cannot occur for well separated errors, so abort correction.

Example 4: CTB-tree Error Correction

The next algorithm to be discussed corrects single errors in CTB-trees [2]. Because of its complexity, only a general indication can be given of the design of the algorithm. Like the previous algorithm, this one uses two parallel traversals. Here, one traversal uses the normal "tree pointers" of the B-tree, and the other uses the chain and thread pointers. The traversal order used places a node immediately after the nodes in its leftmost subtree. As before, an error is detected by observing a disagreement between the two traversals. Unfortunately, it is not easy to determine which traversal is in error, so the correction algorithm guesses: it first assumes the tree pointer traversal is correct and makes the appropriate correction to a chain or thread pointer. The resulting tree is passed to a detection routine to be checked. If the check fails, the "correction" is undone and a tree pointer is changed, assuming the chain and thread pointers to be correct. The resulting tree is again passed to a detection routine. Because the detection routine checks the whole tree, this algorithm can only correct single errors. In principle, a detection procedure could be executed over a subtree, allowing many cases of multiple errors to be corrected, but the details of such a scheme have not been worked out.

Example 5: RCL Error Correction

As nodes of a CTB-tree are examined in the above algorithm, the robust contiguous list (RCL) structure of each node must be checked and corrected. This correction routine begins by attempting to verify the count, the dummy key (K0), and the fill values. If an error is found at this stage, a guess may have to be made as to which is in error, and the resulting guess checked using a detection procedure. (In this case, the correction algorithm itself is used as a detection procedure, with a flag parameter indicating not to attempt correction.) Once these items have been verified, it is straightforward to scan the RCL, checking keys and differences, and when a discrepancy is discovered, to determine whether a key or a difference is in error.

Example 6: K-linked Lists Error Correction

All of the preceding algorithms, except the General Correction Algorithm, can only be guaranteed to correct single errors. The following algorithm is a multiple error correction algorithm for k-linked lists [1]. In [8] it was shown that a k-linked list is (2k-1)-correctable, but no linear time correction algorithm was then known.

The idea used in this algorithm is to partition the pointers into k sets, such that each set is 1-correctable. (Each set contains two pointers per node. The correction is performed by a generalisation of the algorithm described in Example 2 of this section.) The algorithm begins by performing 1-correction on each of these sets. Then it uses a set which is now correct to correct any sets which were not previously corrected. The only major difficulty is that if a set contains several errors it may appear (to the 1-correction routine) to have no errors or one correctable error. Thus, after the 1-correction routines have been executed, it is necessary to guess which one is to be believed, then use the corresponding set to fix all the other sets, even those which apparently don't need to be fixed. If this results in problems (a large number of "errors" being fixed), another guess is tried. In the worst case, it may be the k'th guess which is correct, and since a guess can result in O(n*k) node accesses for an n-node instance, the algorithm might take O(n*k**2) time. Except for pathological cases, the time will be only O(n*k), but in any event, the time is linear in n, for any given value of k.

3. PRINCIPLES

Using the six correction procedures of the preceding section as examples, we would now like to determine some principles for the design of correction procedures. Implications for the design of storage structures will be discussed in Section 5.

Principle 1: Data should be partitioned.

In some simple cases this may not be necessary, but for a complex structure, partitioning seems essential. In the modified(2) local correction algorithm, an essential feature is the partitioning of nodes between the two traversals. Similarly, the CTB-tree correction uses two parallel traversals operating on disjoint sets of pointers. A more vivid example occurs in the k-linked correction algorithm, which uses a k-way partition to "divide and conquer" the correction problem.

To be useful, a partition generally must possess two properties. The first is that the partition be "stable" in some sense: as much as possible, errors should not shift data between blocks of a partition. One feature of a storage structure which may cause this problem is a tagged pointer, such as is often used to distinguish thread pointers in binary tree storage structures. If different tag values can cause the pointer to belong to different blocks of the partition, considerable confusion is possible. The second property is that each block of the partition should ideally be a "determining set:" all other structural data should be reconstructable from each block of the partition. This allows the structural data from different blocks of the partition to be "compared" in order to detect and correct errors. The three examples in the previous paragraph are all reasonably stable partitions and all partition blocks are determining sets.

Principle 2: Looping and unbounded foreign traversal must be prevented.

If a loop has been introduced into a structure which should be loop-free, the correction routine must avoid repeated traversals of the loop. Similarly, when a pointer is changed to point to an area of storage which is not a node of the instance being checked, it must be possible to bound the number of "nodes" which will be visited outside the instance. Usually, the latter is easy to accomplish if there are identifier fields in the instance. It is more difficult to make a general statement about the prevention of looping. The General Correction Procedure keeps a list of all nodes previously visited, which is effective but very expensive. The other examples given in Section 2 generally make use of the fact that an invalid loop must contain a detectable error somewhere, so when an error is detected, no further traversal is attempted from the point of error until the error has been corrected. The single error correction algorithm for double-linked lists does this by using a traversal which runs from the end of the list backward, when an error is detected. The other algorithms examine a bounded set of data in the vicinity of the error in order to perform correction. In the CTB-tree error correction algorithm, the size of this set is not bounded by a constant, but has a bound which depends on the height of the tree in some cases. (During initialisation, determination of the correct tree height is an important activity.)

Principle 3: A coroutine organisation is frequently useful.

In the modified(2) local correction procedure and the CTB-tree correction procedure, there are (conceptually) two coroutines which can individually generate a traversal order using one block of the partition. This simplifies the correction algorithm since any single error can affect only one coroutine. Thus, the only errors which are hard to handle are those which cause both coroutines to detect no errors but produce a disagreement on the identity of the next node in the traversal order.

Principle 4: Use a fault dictionary.

A classical hardware diagnosis technique makes use of a "maintenance dictionary" [3]. The observed symptoms of a problem are looked up in the maintenance dictionary to determine which replaceable unit(s) is (are) implicated. If the collection of symptoms is properly organised, an analogous technique can be used in correcting data structure errors. An example of such a "fault dictionary" was given in the description of the modified(2) local correction routine. That example is sufficiently simple that an explicit fault dictionary is useful but not essential. The CTB-tree correction routine uses a fault dictionary which is sufficiently complex that there seems no reasonable way of not making the dictionary explicit.

Principle 5: If you can't decide, guess.

In many cases it is too difficult to analyse all available data in order to decide precisely which field is in error. It is often useful therefore to perform a more limited analysis, which yields several candidates for the error, rather than just one. To make this technique successful, two criteria must be satisfied: (1) The number of guesses must be limited, usually bounded by a constant independent of the data structure instance. (2) It must be possible to check the guess with an appropriate algorithm.

In Section 2, only the two double-linked list correction algorithms do not make any guesses. The General Correction Algorithm is almost entirely based on guessing. Because it does not satisfy criterion (1), it is not a practical correction algorithm. The CTB-tree and RCL correction procedures each make at most two guesses and then use a detection procedure to check the correctness of the complete resulting instance. In the k-linked correction procedure, the guessing concerns which single error correction result to believe; the guess is checked by observing the apparent number of errors which must be corrected. If a local error correction procedure were to make guesses, a guess would have to be checked using only a fixed set of data "near" the current position in the instance.

4. A HARD PROBLEM: THE CT-TREE

In addition to the examples of correction algorithms in Section 2, we have an example of a correction algorithm we have not been able to develop. The chained and threaded representation of a binary tree (CT-tree) is known to be 1-correctable, but no correction algorithm for it, other than the General Correction Algorithm, has been developed.

Attempts have been made to produce a correction algorithm for CT-trees and have led to the following conclusions: (1) An algorithm for performing single error correction on CT-trees in linear time probably exists. (2) Any such algorithm is too big and complex to be worth writing. Neither of these has been proven; the second is too subjective even to attempt a proof.

It is possible to examine the CT-tree in terms of the principles of Section 3 to determine why our attempts to obtain an efficient error correction algorithm have been unsuccessful. The first principle requires a stable partitioning of structural information. Because of the tagged pointers, a partitioning into (1) links and (2) chains and threads is very unstable. This partition also fails to have the "determining" property, since chains and threads cannot be used to reconstruct links. Because only a relatively few link configurations are consistent with a given set of chains and threads, it might be possible to work around this second difficulty, but there seems no easy way of avoiding unstable partition problems. There also seems to be no other useful partition of the structural data.

Principle 2 requires loops to be prevented. Because any standard tree traversal will require an indeterminate number of links to be followed, upon occasion, to access the next node, preventing loops is very difficult.

The coroutine idea of Principle 3 is applicable, except for the partition stability problem. Principles 4 and 5 might also apply, if the problems previously outlined could be solved.

The preceding say, in effect, that our present principles do not allow us to construct a reasonably simple, efficient correction routine for CT-trees. The only hope of success would seem to be the discovery of a new, and quite different, principle for the design of correction routines, or the development of a better binary tree storage structure.

5. IMPLICATIONS FOR STORAGE STRUCTURE DESIGN

In a sense, the principles of Section 3 simply imply that storage structures should be designed so that the principles can be applied. A few, more specific, implications for storage structure design can be extracted.

One implication is that a k-determined storage structure should be easy to correct if k is sufficiently large. (Informally, a structure is k-determined if the structural data can be partitioned into k sets, such that any one set can be used to reconstruct all other structural data.) Storage structures which are only 1-determined may be difficult to deal with.

Another implication is that structures with "simple" traversals, where the number of operations to reach the next node is bounded by a constant, are easier to deal with. Unfortunately, this property tends to be inherited from the data structure being represented. For example, no standard linked representation of a binary tree has this property. A final implication, drawn directly from the examples of Section 2, is that "global" properties such as a count of the number of nodes in an instance, may not be very useful for correction purposes. (Although they are often convenient for error detection.) Counts were assumed to be present in all storage structures, but usually the correction routines simply reset the count to match the pointer structure, once pointers were corrected. There are two exceptions: the count is quite important in the RCL correction algorithm and the count is used as a "tie breaker" in the k-linked list correction algorithm when two apparently correct sets disagree on the number of nodes in the list. The RCL is the only contiguous storage structure included here: counts may have a greater importance for contiguous structures than linked structures.

6. SUMMARY AND FURTHER WORK

We have presented a number of examples of correction algorithms, extracted some general principles from the algorithms, and suggested some implications from these for the design of storage structures. We believe that the principles of Section 3 are useful in the design of storage structure error correction algorithms. Although the principles are presented here as observations of common properties of correction algorithms, the principles were in several cases used in the design of the algorithms. For example, Principles 1, 3, and 5 (partitioning, coroutine traversal, and guessing) were all used explicitly to guide the overall design of the CTB-tree correction algorithm.

Although the work presented here should provide help in the design of error correction algorithms, much work remains to be done. For example, the local correction routine presented in Section 2 is only applicable to modified(2) double-linked lists. It should be extended to modified(k) double-linked lists for k > 2. Another obvious unsolved problem is efficient error correction in storage structures like the CT-tree, which do not yield to our present principles.

A significant implementation problem is the inter-module coupling in correction algorithms. The problem is not obvious in the descriptions of Section 2, but in the implementation of correction algorithms, it often seems to be necessary for various modules to know about details of the implementation of other modules. For example, the local error correction algorithm for modified(2) double-linked lists uses two traversal procedures. One would expect an initialisation routine and a "next node" routine to be sufficient for the traversal and that no other modules would have to know the implementation of these routines. Unfortunately, it is necessary to know whether the "next" routine has just followed zero or two pointers, and there must be a mechanism for resetting the traversal when an error is corrected in a pointer previously followed. Finding good modular design techniques for correction algorithms is a problem we have not yet been able to solve.

Acknowledgements

The work described in this paper was supported by the Natural Sciences and Engineering Research Council of Canada under grant A3078 and a Postgraduate Scholarship. Support was also provided by a research contract from the Digital Equipment Corporation. The implementation of the software described in Section 2 was performed with the assistance of several programmers: Ralph Hill, Mark Ingram, and Adrian Pepper. In addition, Chidi Nwulu did much of the design for the k-linked list correction algorithm, as well as its implementation.

BIBLIOGRAPHY

1. Black, J. P., D. J. Taylor, and D. E. Morgan. A compendium of robust data structures. Digest of Papers, The Eleventh Annual International Symposium on Fault-Tolerant Computing, Portland, Maine, June 24-26, 1981. pp129-131.

2. Black, J.P., D. J. Taylor, and D. E. Morgan. A robust B-tree implementation. Proceedings of the 5th International Conference on Software Engineering, March 9-12, 1981, San Diego, California. pp63-70.

3. Downing, R. W., J. S. Nowak, and L. S. Tuomenoksa. No. 1 ESS Maintenance Plan. Bell System Technical Journal, vol. 43 (September 1964). p1961-2019.

4. Horning, J. J. Programming Languages. In Computing Systems Reliability, edited by T. Anderson and B. Randell. Cambridge University Press, 1979. pp109-152.

5. Randell, Brian. System Structure for Software Fault Tolerance. IEEE Transactions on Software Engineering, vol 1, no. 2 (June 1975). pp220-232.

6. Taylor, David J., James P. Black, and David E. Morgan. Redundancy in data structures: Improving software fault tolerance. IEEE Transactions on Software Engineering, vol. 6, no. 6 (November 1980). pp585-594.

7. Taylor, David J., James P. Black, and David E. Morgan. Redundancy in data structures: Some theoretical results. IEEE Transactions on Software Engineering, vol. 6, no. 6 (November 1980). pp595-602.

8. Taylor, David J. Robust Data Structure Implementations for Software Reliability. Ph.D. thesis, University of Waterloo, Waterloo, Ontario, 1977.

APPENDIX: STORAGE STRUCTURES

Several linked linear list structures are referred to in Section 2. The double-linked list is essentially just the standard two-pointer representation with pointers to the following and preceding nodes. We also assume that each node contains an identifier field, whose value uniquely identifies a node as belonging to a particular data structure instance, and that there is a header node containing a count of the number of nodes on the list.

A modified(k) double-linked list is like a standard double-linked list except that the "backward" pointer points to the k'th preceding node rather than the immediately preceding node. Thus, a modified(1) double-linked list is a standard double-linked list. It is necessary to have k header nodes for a modified(k) list and to store the headers contiguously.

A k-linked list has 2k pointers in each node, one pointer to each of the k following nodes and to each of the k preceding nodes. Identifier fields and a count are also used, and like modified(k) lists, k header nodes are required.

A chained and threaded binary tree (CT-tree) is a binary tree using identifier fields and a node count stored in the header, which uses logically null pointers to provide a second access path to each node. Unused right pointers contain a thread to the symmetric order successor of the node (this is a standard binary tree implementation trick). Unused left pointers contain a "chain" to the next (in symmetric order) node which has an unused left pointer. In each case, a tag is used to indicate whether a pointer is a "normal" link or a chain or thread. An example of a CT-tree is shown in Figure 3.

A chained and threaded B-tree (CTB-tree) is a B-tree which uses identifier fields, has a node count and the height of the tree stored in the header, and the height (above the leaves) stored in each branch node. A CTB-tree also has chain pointers connecting all the leaf nodes in order and has thread pointers from certain leaf nodes to branch nodes. The rule for thread pointers is that the rightmost leaf in the leftmost sub-tree of a branch node contains a thread to that branch node. Height fields are not required for error detection or correction: they were added to simplify the correction algorithm. An example of a CTB-tree is shown in Figure 4.

Each node of a CTB-tree is represented as a Robust Contiguous List (RCL). An RCL contains stored differences between consecutive keys, a stored count of the number of keys, a special "dummy" key at the beginning of the list, and has unused key and difference positions filled with a special value computed from the dummy key. An example of an RCL is shown in Figure 5.

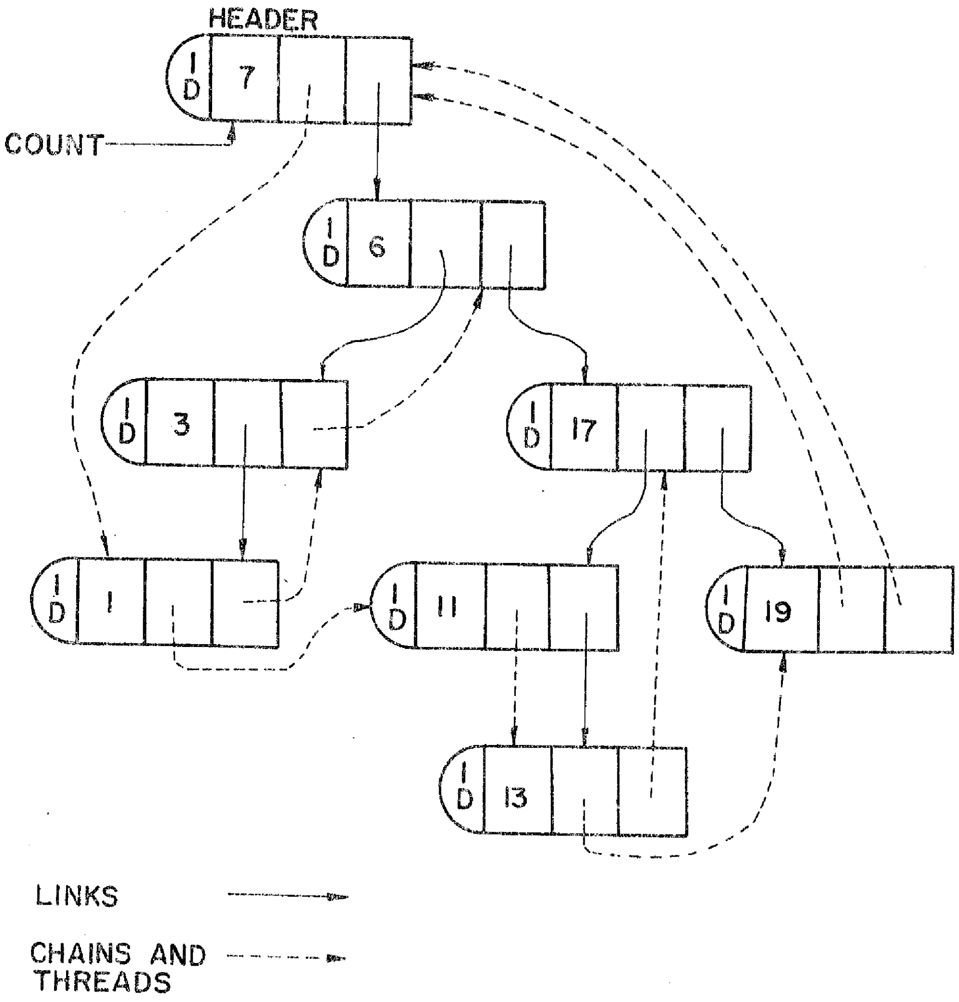

Figure 3

Chained and Threaded Binary Tree

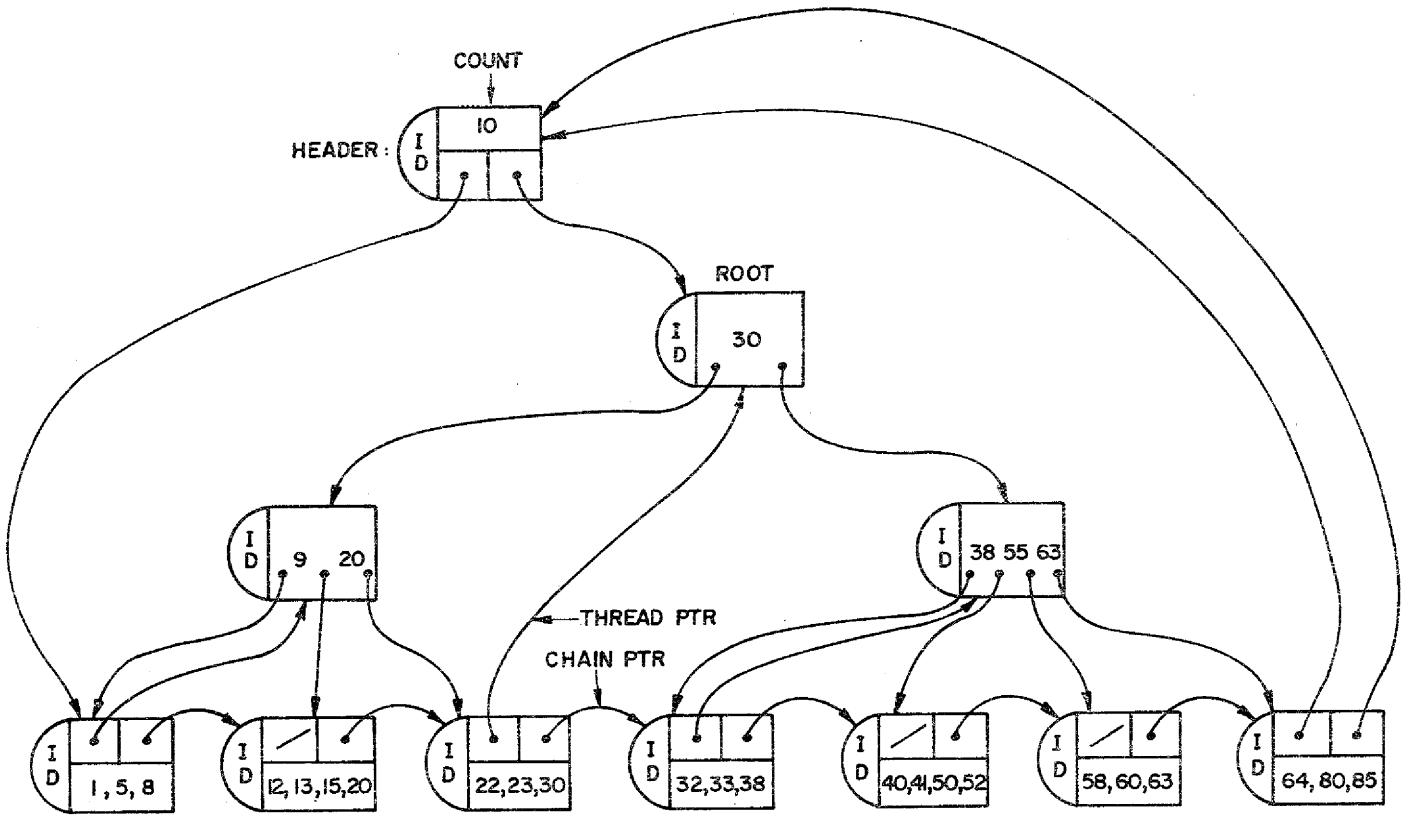

Figure 4

Chained and Threaded B-tree

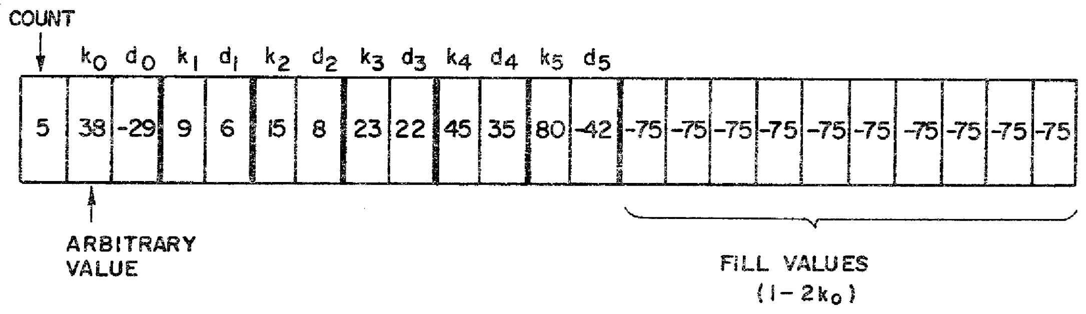

Figure 5

Robust Contiguous List