A ROBUST B-TREE IMPLEMENTATION

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

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

March, 1980

CS-80-15

(c) COPYRIGHT: Computer Science Department
University of Waterloo
Waterloo, Ontario, Canada

Abstract

A storage structure for B-trees is presented which is robust in that any pair of changes to structural fields of an instance of the structure can be detected, as well as many sets of larger numbers of changes. Included in the paper are a motivation for robustness as a design criterion, cost and performance implications of the B-tree implementation, and a solution to the subproblem of a robust implementation for each node's contiguous list of pointers and keys.

Key Words and Phrases: Software fault tolerance, robust data structures, computer system reliability, data base reliability, data base management system, redundancy, error detection, error correction, B-trees, B*-trees, CTB-trees.

1. INTRODUCTION

As the size and complexity of data base systems, and society's reliance on them, increase, their reliability becomes an increasingly important consideration. One way of improving reliability is by incorporating redundant information in a system's data structures, and using this information to detect and correct errors before they cause system failure. Such data structures, which incorporate redundant information allowing error detection and possibly correction are termed *robust*. This paper presents a robust storage structure for B-trees, and demonstrates that a significant degree of robustness can be achieved at reasonable cost. Files structured as B-trees are often one of the basic building blocks of data bases.

Our approach to increasing software system reliability is *fault tolerance*, using Avizienis' terminology [2], and complements the more well known *fault intolerant* [2] techniques applied during system development to increase the reliability of the final product. Examples of fault intolerant techniques are proofs of program correctness, structured design and programming methodologies, and development aids for systematic testing and debugging. Fault tolerant techniques, on the other hand, attempt to cope with the unexpected events which occur during normal system use: the sudden appearance of residual bugs or design flaws, hardware malfunctions, user mistakes, and

events unrelated to a given program, such as an operating system crash. The fault tolerant approach attempts to prevent such events and the errors they produce from causing system failure. This involves error detection and correction, and when system failure does occur, recovery.

The key to error detection, correction, and recovery is redundancy, which may be of four types: redundant hardware, redundant software, redundant data, and redundant information about the system's behaviour. This paper is concerned with adding redundant data to storage structures, and exploiting and maintaining this redundancy through (redundant) software.

More detailed discussions of fault tolerance and intolerance, and computing systems reliability may be found in [1, 2, 4, 5]. For the general framework of our research, see [6, 7].

After this brief introduction, Section 2 introduces some of the general terminology and concepts of robust data structures. Section 3 develops a sequence of increasingly robust B-tree implementations, although they all suffer from a local "vulnerability" to errors at each node. This subproblem is solved in Section 4, and its solution leads to the final proposal of Section 5 for a robust implementation of B-trees, called the CTB-tree. In Section 6, the cost and performance implications of using CTB-trees are examined, and Section 7 presents our conclusions and some ideas for

further work.

2. ROBUST DATA STRUCTURES

Following Tompa [10], we define a <u>data structure</u> to be a logical organisation of data. The data structure is represented by a <u>storage structure</u>, which has a particular <u>encoding</u> for a given storage medium. For example, "binary tree" is a data structure; one possible storage structure for it contains pointers from each node to its left and right sons; if we further specify that pointers are stored as absolute addresses of 32 bits, that is an encoding of a binary tree. Furthermore, a <u>data structure instance</u> is a particular occurrence of a data structure. When the context makes the meaning clear, we will also use "data structure instance" to refer to the storage structure for the instance, or its encoded form.

In general, the synthesis of robust data structures involves adding redundancy to storage structures in order to allow the detection and correction of errors. On the one hand, we seek conditions under which the robustness of an arbitrary storage structure may be quantified; on the other hand, given a data structure, we wish to find an implementation (storage structure) for it which has at least some degree of robustness. The former approach is documented in [7, 8, 9]. Here, we wish to present an informal analysis for a particular data structure, the

B-tree [3], while at the same time introducing some more general concepts.

In order to quantify robustness, we use the terms correct, change, detectability, and correctability. Before defining them, however, we will give the assumptions underlying our approach.

An instance of a storage structure consists of a header and a (possibly empty) set of nodes connected by pointers. Because of the difficulty of making general comments regarding the data content or semantic integrity of a data structure, we are concerneed only with the correctness of structural information (structural integrity). In addition to pointers, the structural information in an instance consists of identifier fields, count fields containing the number of nodes in an instance or pointers in a node, and key fields. An identifier field in a correct instance contains a value which is unique to the type of node and particular instance in the system under consideration. In order to simplify our presentation, we make the Valid State Hypothesis (VSH): the only pointers to nodes of an instance appear inside the instance itself, and no identifier field values for any one instance appear in nodes external to the instance.

For the purposes of this paper, an instance of a storage structure is correct if a "detection procedure" applied to the instance returns the value "correct". In

some sense, then, the detection procedure defines the storage structure. Any of the procedures we discuss must be _reasonable_, that is, they may only locate or examine nodes by following pointers from the header of an instance: this precludes exhaustive memory searches.

Detection properties of storage structures are stated in terms of changes. A _change_ is an elementary modification to the encoded form of a data structure instance. (The meaning of "elementary modification" can be altered to suit the environment; here it will mean the change of of a single word.) We assume that a single change to the encoding is reflected as a single change in the storage structure instance.

If a single change can transform one correct instance into another, then the encoding has no detection capabilities. If at least N+1 changes are required to transform one correct instance into another, then the encoding is N-detectable. Equivalently, the detection procedure rejects any instance which differs from a correct instance by N or fewer (structural) changes. Similarly, a storage structure is _N-correctable_ if there exists a procedure which, for all sets of N or fewer changes, can take a correct instance modified by that number of changes and recreate the correct instance.

We will clarify and make use of these definitions in the next section, whose purpose is to analyse the robustness

of normal B-trees, and suggest appropriate forms of redundancy which will result in a 2-detectable, 1-correctable storage structure called a CTB-tree. A compromise between the cost of failures and the cost of the facilities necessary to cope with errors without failure determines the amount and type of redundancy which should be used to improve reliability. A 2-detectable and 1-correctable structure is often adequately robust; indeed, empirical results indicate the effective robustness of many such structures to be significantly higher [6].

3. THE ROAD TO THE CTB-TREE

We define a B-tree of order n as follows.

1. A B-tree of height h consists of a header which contains a null pointer, or a pointer to either a leaf node containing $1 \leq k \leq 2n$ keys, or to a root node with $2 \leq k \leq 2n + 1$ sons, each of which is an interior B-tree of height h-1.
2. A leaf node with $n \leq k \leq 2n$ keys is an interior B-tree of height 0.
3. An interior node with $n + 1 \leq k \leq 2n + 1$ pointers to interior B-trees of height $h - 1$ is an interior B-tree of height h.
4. For an interior node or root node of k sons, all keys in the i'th interior B-tree are less than or

equal to key(i) ($1 \leq i < k$), and the keys in the i'th interior B-tree are greater than key(i-1), for $1 < i \leq k$.

5. For any node having k keys, key(i) < key(i+1), $1 \leq i < k$.

We assume all data is kept in leaf nodes.

This standard storage structure is not at all robust; specifically, it is 0-detectable and 0-correctable. There are in general many pointers which can be changed to null, resulting in the deletion of an entire subtree, leaving a correct B-tree. In fact, changing to null the pointer from the header to the root deletes the entire tree. In this section, we are concerned only with changes to identifier, pointer, and (global) count fields; Section 4 discusses robustness in the face of changes to keys.

A well known way to improve the robustness of storage structures is to add to the header a count of the number of nodes in the instance, and to add identifier fields to each node of the structure. This introduces little overhead, and such redundancy is easily maintained. The addition of these fields makes the storage structure 1-detectable, although still 0-correctable. A single change to the count can be detected by traversing the tree to find the apparent number of nodes. A change to an identifier field can be detected in like manner, as can the change of some pointer to null. The change of a pointer to point to a foreign node can be

detected, as the foreign node cannot have a valid identifier field under the Valid State Hypothesis.

The remaining case is the change of a pointer to point to some other node already part of the instance. This may be detected by flagging nodes during traversal of the tree, or, if space for this is unavailable, by the more expensive method of storing node addresses in a table with O(log N) search and insertion times for an N-node tree.

Two independent reasons for the 0-correctability are worth mentioning. A single pointer change may sever one or more nodes from the tree, making them inaccessible to a reasonable correction procedure. Additionally, while it may be detected that the count does not match the number of nodes in the instance, it is undecidable whether this is due to a count change or a pointer change.

To demonstrate that the detectability of this structure is exactly one, we point out that deletion of certain nodes or subtrees requires only one pointer change and one count change to transform one correct instance into another.

One form of redundancy often added to B-trees is a set of pointers, each connecting a leaf to its successor. These pointers, which we will call chain pointers, improves the efficiency of sequential retrieval by obviating the need for indexing through the interior nodes. An additional benefit of these pointers is that they increase the detectability of the storage structure to two, although the correctability is

still zero: an interior node can still become inaccessible as a result of a single pointer change.

This time, we will show the 2-detectability by demonstrating that any two correct instances are at least three changes distant from each other. The three cases we consider involve changing the number of nodes, replacing one or more nodes by the same number of foreign nodes, and rearranging the same set of nodes.

1. Change in number. Deletion is clearly cheaper than insertion, as valid identifier fields must be supplied for the inserted nodes under VSH. Deleting a subtree now requires changing the count, a pointer coming into the subtree from above, and a change to one chain pointer to "jump over" the deleted subtree, for a minimum of three changes.
2. Replacement. Again, because of the need to place valid identifier fields in the foreign node(s), the minimum occurs for one node. If this node replaces a leaf, incoming and outgoing chain pointers must be changed, as well as an incoming tree pointer, for four changes. If the node is to replace an interior node, the minimum occurs when replacing a root node which has exactly two sons, in which case the incoming header pointer must be changed, and the proper son pointers placed in the foreign node. Counting the identifier field, the minimum is again four.

3. Rearrangement. Given the constraint on key ordering, this type of modification is quite expensive in terms of changes, and will not be considered further.

We have shown that the chained B-tree is 2-detectable, although still 0-correctable. We would like to increase the latter so that single errors can be corrected.

In order to ensure that no node can be disconnected from an instance with a single change (this is clearly necessary if a reasonable procedure is to be used), we require at least two edge-disjoint paths to any node. This condition is already satisfied for leaf nodes, which can be found by following either tree pointers or chain pointers from the header. What is required is a second path to interior nodes.

By analogy with binary search trees, we consider adding a thread pointer from each leaf node to its in-order successor. This indeed provides two edge-disjoint paths to each interior node: one by chain and thread pointers, and one by the normal tree pointers. In fact, each interior node containing k keys (k + 1 sons) has k threads pointing to it. As shown by the General Correction Theorem of [8], Section 4.3, we have now guaranteed the 1-correctability of the chained and threaded B-tree. The theorem, which we state without proof, says that a storage structure using identifier fields which is 2r-detectable and has r + 1 edge-disjoint paths to each node is r-correctable.

As a final refinement, we note that putting a thread in every leaf node provides more redundancy than needed to guarantee the 1-correctability, as well as having serious performance implications: when an interior node is split due to an insertion, n thread pointers must be updated, and these pointers are potentially quite distant from each other. In order to improve performance and increase the possibilities for concurrency, we will reduce the number of threads pointing to an interior node to one, which will be in the last in-order leaf of the leftmost subtree of each node.

An instance of this storage structure, which we call a chained and threaded B-tree (CTB-tree) is shown in Figure 1. Note that the order of the tree is two, and that the final chain and thread pointers both point back to the header.

There is, however, an important problem which we have left unsolved in the CTB-tree: each node of the tree is still 0-detectable and 0-correctable with respect to the key fields; entire data items in leaf nodes may be lost by a single change to the count field in the leaf. This problem is addressed in the next section.

4. ROBUST CONTIGUOUS LIST STORAGE: A SUBPROBLEM

In this section we discuss a slightly more general problem: that of finding a robust storage structure for a contiguous list of up to m elements. That is, the list

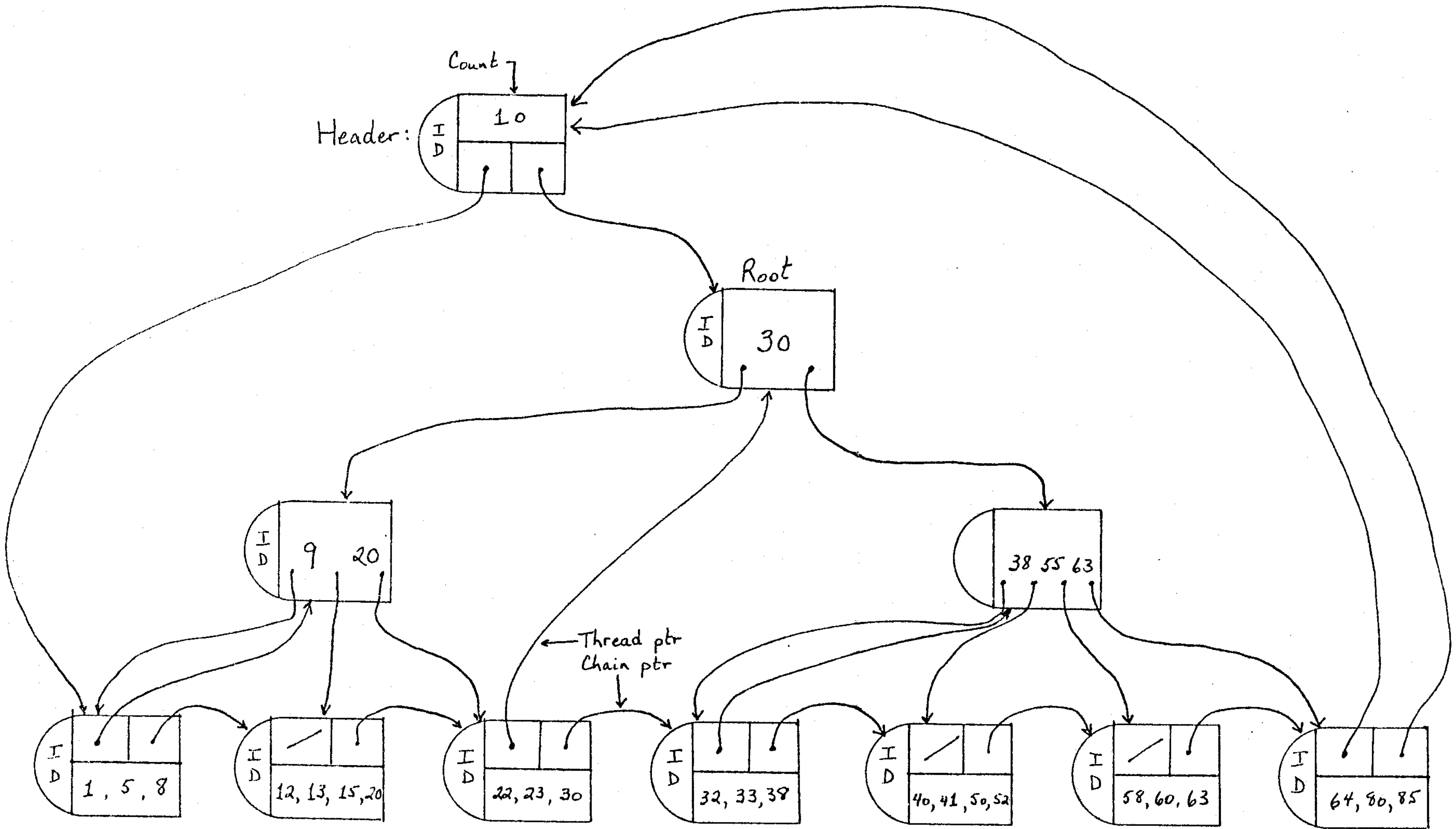

Figure 1. A CTB-tree of order 2.

consists of a fixed size contiguous area of storage capable of containing m or fewer list elements. We assume that each element contains a key field, and that the elements are to be stored left-justified in the storage area or node. We seek a storage structure which is 2-detectable and 1-correctable with respect to changes to key fields, and to the count of the number of elements in the list.

The storage structure for the nodes of the CTB-tree given in the last section is not at all robust. Even constraining the key values to be ordered is only "fuzzy" redundancy: if keys are not dense in a given instance, a small change may not cause the order constraint to be violated. Thus, we need to "protect" two types of information in a contiguous list: the key field values, and the number of nodes in the list.

The approach is quite straightforward. We associate with each key, k(i), a difference field, d(i), which contains the value k(i + 1) - k(i). For the count, we fill empty element slots with some fill value, permitting a redundant determination of the number of elements in the list. However, a certain degree of care is required to ensure that the desired robustness is maintained in boundary cases such as the beginning and end of the list, a one-element list, or a full list.

Considering first the key and difference fields, one needs to decide upon the value of the difference field of

the last list element. Possibilities which come to mind include zero, or the difference to the first key in the list. In both cases, however, the detectability remains zero, since a change to the key of a one-element list produces another correct list. A better idea is to create a dummy "header element" consisting of a key and a difference field, and to specify that the last difference field contains the difference between the last and header keys. We assume that the header key for a given list cannot be known *a priori*, i.e., that it is not a fixed value for all nodes. One possibility would be to place a random value in the field when the node is created. Finally, we assume that the header difference field of a correct but empty list contains the header key value. This protects the value of the header key of an empty list.

This raises the detectability to 1, but not further: one may change the header key and first key of a single element list by the same amount, and still obtain a correct list. This motivates our use of the random header key as fill value for empty list element slots. This solves the previous problem, as fill values and the header key must now agree (and also makes it unnecessary to treat the difference field of an empty list as a special case), but does not solve the following problem.

Consider a contiguous list where the fill value is equal to the header key, k(0), and which contains i

elements. Then $k(i+1)=k(0)$, and $d(i)=k(0)-k(i)=k(i+1)-k(i)$. Changing $d(i + 1)$ to zero, and adding 1 to the count produces anew a correct list of $i + 1$ elements. As a final adjustment, we make the fill value a simple function of $k(0)$, say $k(0) + 1$, which solves this problem of adding one element to the end of the list with two changes, as well as the previous one of changing the header and first keys of a single element list by the same amount.

Before giving a more formal argument of the 2-detectability, we will summarise our definition of a robust contiguous list. Figure 2 shows an instance of such a structure.

A robust contiguous list of m ($m > 1$) or fewer elements is described as follows. The key fields are denoted by $k(0)$, $k(1)$, . . . $k(m)$; the difference fields by $d(0)$, $d(1)$, . . . $d(m)$; and the count field by c.

1. $k(0)$ contains an arbitrary value, R.
2. For $0 \leq j \leq c$, $d(j) = k(j + 1) - k(j)$. (Addition modulo $c+1$)
3. Fill values: for $c < j \leq m$, $d(j) = k(j) = R + 1$.

In order to prove that this storage structure is 2-detectable, we show that any two correct instances are at least 3 changes distant from each other.

1. Element insertion. Even ignoring elements to the right, the count must be changed, and two difference fields calculated.

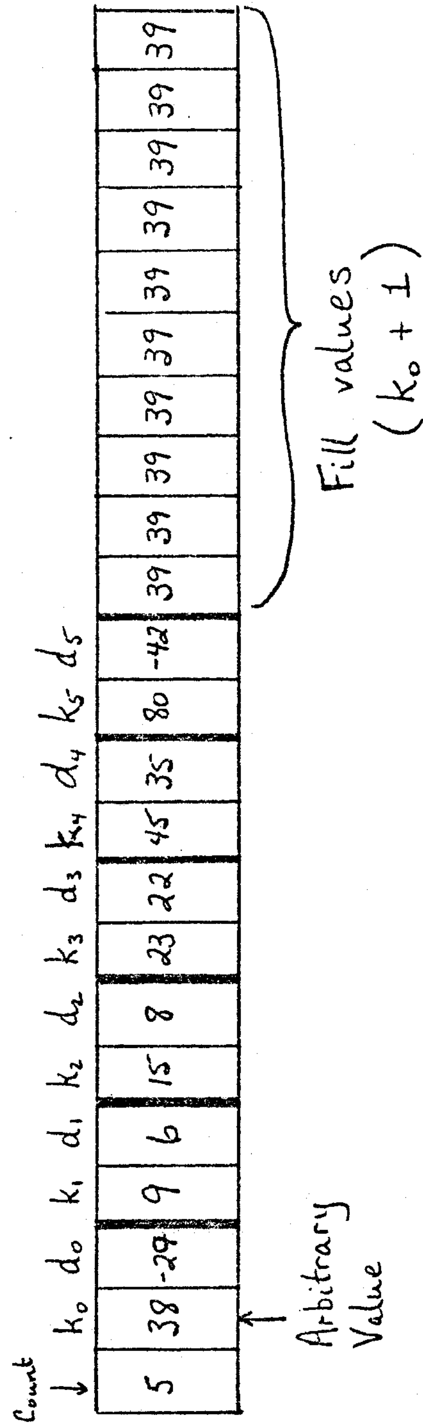

Figure 2. A Robust Contiguous List of 10 or fewer elements.

2. Element deletion. Similar.

3. Change of key. In addition to changing the key, the two neighbouring difference fields must be changed by the same amount.

Note that since the value of the key is not relevant for element insertion or deletion, the argument holds even for "fortuitous" values of the key such as R, R + 1, etc. As pointed out above, changing two keys by the same amount cannot produce a correct list either, as some difference field will be incorrect unless c = 1. But in this case, the fill values and the changed k(0) no longer agree.

Given that this storage structure is 2-detectable, it follows immediately that it is 1-correctable. A detection procedure is easily written to check whether a node is correct or not. Given a correct node modified by a single change, a correction procedure could, in the worst case, guess at possible corrections to single fields until the "corrected" node was accepted by the detection procedure. As at most two changes would be applied at any one time (the change we are trying to correct and the guess attempted by the correction procedure), the only node accepted by the detection procedure would be the (corrected) original version. A linear-time 1-correction procedure for robust contiguous lists is still a subject for current research.

After this digression to discuss the robustness of isolated nodes in the B-tree, we return to the CTB-tree in

Section 5. After recalling its structure, we present a linear-time 2-detection procedure for it.

5. THE CTB-TREE

We are now able to give a more precise definition of a CTB-tree, which we have shown to be 2-detectable and 1-correctable.

1. A CTB-tree consists of a header and a possibly empty set of leaf nodes and interior nodes. The header contains an identifier field; a count field, whose value is the number of nodes connected to the header; a chain pointer; and a root pointer.
2. A CTB-tree is a B-tree, as defined above.
3. Interior nodes consist of an identifier field and a robust contiguous list whose <u>data</u> elements are the B-tree pointers.
4. Leaf nodes consist of an identifier field, a chain pointer, a thread pointer, and a robust contiguous list whose data elements are the data elements of the B-tree.
5. The leaf nodes are connected in in-order by the chain pointers, beginning with the chain pointer in the header, and ending in the final leaf with a chain pointer back to the header.
6. The thread pointer in a leaf is non-null if and only if that leaf is the rightmost leaf in the leftmost

subtree of a node X, in which case the thread points to X.

Figure 3 shows a sketch of a 2-detection procedure for CTB-trees which requires O(N) execution time for an N node CTB-tree, and space proportional to the height of the tree. The procedure relies on a procedure "check_contig_list" to check the robust contiguous list at each node. It can be shown that the procedure terminates without error if the header points to a proper CTB-tree, and otherwise terminates indicating an error.

As for a correction procedure, the General Correction Theorem mentioned above is constructive, in that it gives an algorithm for performing the correction. As indicated above, a similar correction procedure could be used for each contiguous list. The execution time for the general algorithm is expressed in terms of an unfortunately large polynomial in the size of the instance. Finding a special purpose, efficient 1-correction routine for CTB-trees is a subject of currect research.

6. PERFORMANCE IMPLICATIONS

Obviously, the robustness of CTB-trees (or any other robust data structure) is achieved at the expense of extra storage to store the redundancy, and extra execution time to maintain and exploit it for error detection and/or correction. Only the cost of system failure or

```
procedure check_ctb(header, N, h)
  /*  Check the CTB-tree of order N and height h
   *  purportedly attached to "header". */
  begin
  procedure ch_ctb(h, i, top, ch, th, highkey)
     /*  Check the interior CTB-tree pointed to by top.
      *  Verify that ch is the leftmost leaf, and update it
      *  to the expected leftmost leaf in the next subtree.
      *  Update th to the thread coming out of the tree, and
      *  highkey to the highest key in the tree.
      *  i is a count of the number of nodes seen. */
     begin
     if i > count(header) then error endif ;
     if id(top) incorrect then error endif ;
     if count(top) < N then
        if top ~= root(header) or count(top) <= 0 then
           error endif ;
        endif ;
     if key[1](top) <= highkey then error endif ;
     i := i + 1 ;
     check_contig_list(top, 2 * N) ; /* Check this node. */
     if h = 0 then  /* Leaf node */
        if ch ~= top then error endif ; /* Bad chain */
        ch := chain(top) ;
        th := thread(top) ;
        highkey := key[count(top)](top) ;
     else  /* Interior node */
        for j from 0 to count(top) - 1 do
           ch_ctb(h-1, i, ptr[j](top), ch, th, highkey) ;
           if j = 0 then
              if th ~= top then error endif ;
           else if th ~= null then error endif ;
              endif ;
           if highkey > key[j + 1](top) then error endif ;
           highkey := key[j + 1](top) ;
           endfor ;
        ch_ctb(h-1, i, ptr[count(top)](top), ch, th, highkey) ;
        /* Note th and highkey set for caller. */
        endif ;
     end ch_ctb ;
  /*  Body of detection procedure.  */
  ch := chain(header) ;
  highkey := 0 ;   /* Assume all keys > 0 */
  i := 0 ;
  ch_ctb(h, i, root(header), ch, th, highkey) ;
  if i ~= count(header) then error endif ;
  if ch ~= header then error endif ;
  if th ~= header then error endif ;
  return("correct") ;
end check_ctb ;
```

Figure 3. CTB-tree Detection Procedure.

unavailability can determine whether an investment in robustness is justified, but we wish to show in this section that the costs associated with CTB-trees are not prohibitive.

The main extra storage cost for CTB-trees is for the difference fields in interior (index) nodes. Assuming that identifier, count, key, difference, and pointer fields all require the same amount of storage, the index space required for a CTB-tree is roughly 1.5 times that required for a B-tree. However, if, as usual in files organized as B-trees, leaf node sizes are significantly larger than interior node sizes, or if the order of the tree is large (say greater than 10), this is a small space overhead. The addition of three words of storage to each leaf node for the identifier field, chain pointer, and thread pointer, and of a difference field for each data element, results in a negligible space overhead for reasonable data element sizes.

Although the actual increase in storage required for a CTB-tree as compared to a corresponding B-tree is not large, it may have an effect on the height of the tree, and hence on the cost of search and update operations. If we assume that the optimum interior node size is fixed by external considerations (device characteristics), and if a B-tree has optimal order n, the corresponding CTB-tree will have order 2n/3. This is due to the difference fields added in the robust contiguous list in each node, which add an extra word

of storage for each key/pointer pair. For a B-tree of k nodes and order n, its height, h, is bounded by:

$$\lceil \log_{2n} k \rceil \leq h \leq \lceil \log_{n} k \rceil$$

($\lceil X \rceil$ indicates the ceiling function; $\log_b$ the logarithm to the base b.) For a CTB-tree with the same number of nodes, and same interior node size, we have:

$$\lceil \log_{4n/3} k \rceil \leq h \leq \lceil \log_{2n/3} k \rceil$$

Ignoring errors due to the integer ceiling function, the minimum height is multiplied by a factor $(\log_{2n} 4n/3)^{-1}$ or $(1 + \log_{2n} 2/3)^{-1}$, and the maximum by $(1 + \log_{n} 2/3)^{-1}$. A simple evaluation of the minimum and maximum heights for the two types of tree and various values of n and k shows that the difference is very rarely greater than one. For many pairs (n, k), the minimum (maximum) heights of the B-tree and CTB-tree are equal.

Another cost imposed by the CTB-tree is in the number of I/O operations required for node splits and underflows. For leaf splits (underflows), none are required. For splits (underflows) above the leaves, extra operations are required to update thread pointers. Each split or underflow requires following a path down to the appropriate leaf (read operations), and updating its thread pointer (write operation). Considering the relative frequency of splits above leaf level, this overhead is negligible as well.

We consider one further cost associated with CTB-trees: the cost of error detection. A detection routine may be run

periodically, as well as when trouble is suspected. By making the period longer, the cost may be made as small as desired. However, this must be weighed against the increasing risk of multiple errors being introduced, or existing errors being propagated. Again, the expected error rate and the cost of system failure are important factors to consider.

7. CONCLUSIONS, AND FURTHER WORK

One interesting aspect of the robustness of CTB-trees is that each node is individually 2-detectable and 1-correctable with respect to count, key, and difference fields. This suggests that the effective robustness may be much higher, since any number of key/count/difference errors in different nodes may be detected/corrected. On the other hand, the structural (identifier field, global count, and pointer field) robustness of the CTB-tree is a global property. Still, the effective robustness should be much higher, as the 2-detectability and 1-correctability is a worst case result: there is at least one set of three changes which is undetectable. For an error source which is not "malicious", one would expect such fortuitous conjunctions of errors to be extremely rare. Experimental results along these lines have been obtained for other robust data structures; see [6].

Many areas for further work can be identified. It was

mentioned above that linear time 1-correction routines for CTB-trees and robust contiguous lists have yet to be developed. It would be interesting to examine the robustness of B-trees or CTB-trees which have secondary indices. More generally, our research attempts to develop a theory of robustness in data structures, and to find specific robust data structures providing adequate robustness at acceptable cost.

In this paper, we have presented a new storage structure for B-trees, the CTB-tree, and examined its robustness. We have shown that it is 2-detectable and 1-correctable, and that this degree of robustness can be achieved at an acceptable cost. We also discussed the problem of finding a robust implementation of a contiguous list, which was necessary at each node of the CTB-tree. We believe that the CTB-tree is a useful tool for increasing the reliability and fault tolerance of data base systems, and that this is achieved at acceptable cost.

BIBLIOGRAPHY

1. Anderson, T., and B. Randell (eds.). Computing Systems Reliability. Cambridge University Press, 1979.

2. Avizienis, Algirdas. Fault-tolerance: The survival attribute of digital systems. Proceedings of the IEEE, vol. 66, no. 10 (October 1978). pp1109-1125.

3. Bayer, R., and C. McCreight. Organisation and maintenance of large ordered indexes. Acta Informatica, vol.1, no. 3, 1972. pp173-189.

4. 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.

5. Randell, Brian. Operating systems: The problems of performance and reliability. Information Processing 71, Proceedings of IFIP Congress 71, Ljubljana, Yugoslavia, August 23-28, 1971. pp281-290.

6. Taylor, D. J., D. E. Morgan, and J. P. Black. Redundancy in data structures: Improving software fault tolerance. Accepted for publication in IEEE Transactions on Software Engineering.

7. Taylor, D. J., D. E. Morgan, and J. P. Black. Redundancy in data structures: Some theoretical results. Accepted for publication in IEEE Transactions on Software Engineering.

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, David J. Theoretical foundations for robust data structure implementations. Computer Science Research Report, CS-78-52, University of Waterloo, Waterloo, Ontario, Canada.

10. Tompa, Frank W. Data structure design. Data Structures, Computer Graphics, and Pattern Recognition, edited by A. Klinger, et al. New York, Academic Press, 1977. pp3-30.