UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

A Locally Correctable B-tree Implementation

D.J. Taylor
J.P. Black

Data Structuring Group
CS-84-51

December, 1984

A Locally Correctable B-tree Implementation

David J. Taylor

James P. Black

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

ABSTRACT

A storage structure for B-trees is presented which is robust, in that many errors and combinations of errors in its structural data can be detected and corrected. The structure presented here is superior to a previous robust B-tree in a number of ways: insertion and deletion are simpler, the classes of errors which can be detected and corrected are larger, and it is simpler to implement a correction routine. These advantages are achieved without any significant increase in cost; storage space requirements and update time are almost unchanged. This paper describes briefly both the old and new B-tree implementations, and makes comparisons between them.

1. Introduction

In choosing a storage structure for a data structure, the space and execution time costs of the alternatives must be considered. If the storage structure is to be used in an environment where reliability and availability are important, the robustness of the storage structure must also be considered.

Previous papers [4,8,9,10] have provided justification for our approach to improving the fault tolerance of a system by making storage structures robust, and have provided precise definitions of terminology. Thus, we provide here only a brief, informal definition of terminology used in later parts of the paper.

The fundamental idea in making a storage structure robust is that it should be possible to detect and correct damage to instances of the structure. We have concentrated primarily on damage to structural data, as opposed to user data, contained in a storage structure. Thus, a storage structure is *n-detectable* if any set of n or fewer errors in structural data can be detected. That is, if a storage structure is n-detectable, any correct instance when modified by n or fewer errors becomes a detectably incorrect instance. Various definitions of "correct instance"

can be made. For practical purposes, there is usually a *detection routine* for a storage structure, which returns a binary value indicating the correctness of an instance. A storage structure is *n-correctable* if there is a procedure which can take an instance containing n or fewer errors and undo all the errors. A storage structure is *locally correctable* if there is a procedure which can take an instance containing arbitrarily many errors and undo all of them, provided no two errors are too close to each other. (It is very difficult to provide an informal but reasonably precise definition of local correctability. A more precise definition requires the construction of a "locality" for each field in an instance of the storage structure. Then, a storage structure is locally correctable if successful correction is guaranteed by the requirement that an error in a field implies no other errors in the locality of that field. For details, see [3].)

The next section of the paper briefly describes B-trees and a previous robust B-tree, and then presents a new robust B-tree, the LB-tree. Section 3 derives robustness properties of LB-trees and compares these with the previous robust B-tree. Section 4 provides the results of implementation experience with the LB-tree. Section 5 presents some conclusions and possibilities for future work.

2. B-trees

The B-tree [1] is a very commonly used data structure, particularly in data base systems. As a starting point for the development of robust B-trees, we define a B-tree of order n as follows.

1. All nodes except the root contain between n and $2n$ keys. The root contains between 1 and $2n$ keys.
2. Nodes are of two kinds: leaf nodes, which have no pointers to other nodes, and branch nodes, which point to leaves or other branch nodes.
3. All paths to leaves have the same length.
4. Branch nodes having k keys contain $k+1$ pointers to subtrees; all keys in the i'th subtree are less than or equal to the i'th key and all keys in the $(i+1)$'th subtree are greater than the i'th key. Keys in each node are strictly increasing.

This form of B-tree is sometimes referred to as a B^+-tree [6]. An example of a B-tree of order 2 is shown in Figure 1.

This B-tree is not at all robust. A precise analysis requires specification of the representation of the B-tree nodes, in particular how the number of keys stored in a node is represented. For any simple scheme, such as storing a count or filling unused key or pointer positions with a special value, it is possible to delete an entire subtree of the B-tree with a single error, by reducing the number of keys in a node by one. Thus, a standard B-tree is 0-detectable and 0-correctable.

In order to construct a robust B-tree, we need a robust pointer structure and a robust representation for the keys stored at each node. The two B-trees described in this section use the same representation for the keys of a node, a robust contiguous list (RCL), so the RCL structure is described first.

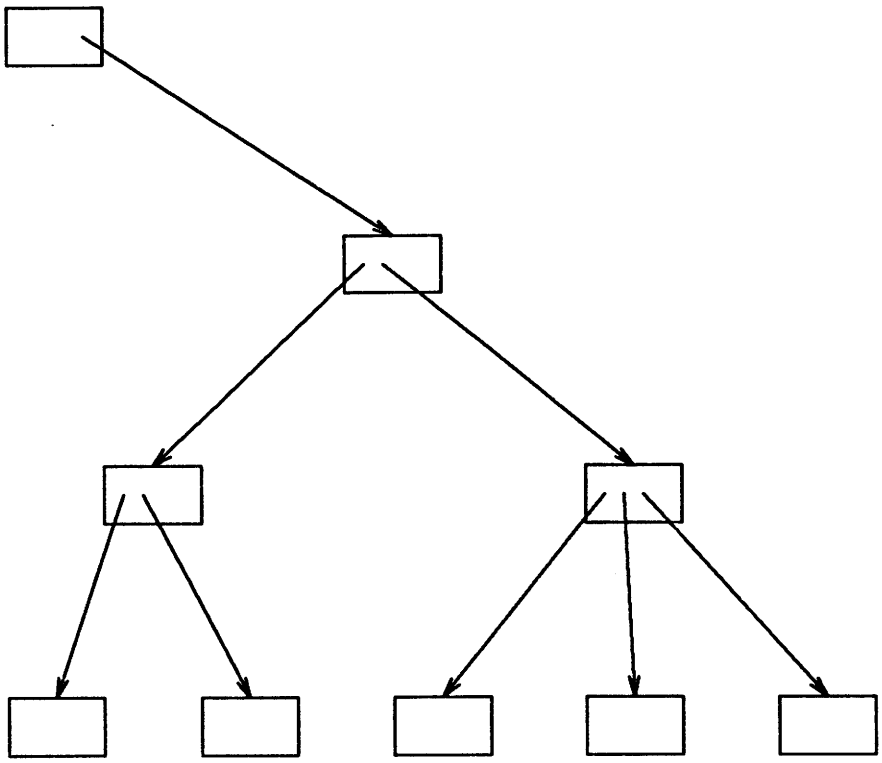

Figure 1. B-tree of order 2.

A robust contiguous list is intended to store a sequence of keys so that the number of keys and the values of the keys are protected. Specifically, we desire a 2-detectable and 1-correctable structure. It is also possible to associate additional data with each key, but the value of such data is not protected by the RCL. For example, in a B-tree branch node, a pointer is associated with each key: the number of such pointers is protected by the RCL, but the values of the pointers must be protected by the overall pointer redundancy.

The following definition for an RCL of m elements has been published previously [5]. The fundamental idea is simply to add a difference field, containing the difference between two successive keys. Justification for the particular redundancy chosen and proofs of detectability and correctability can be found in the referenced paper.

In the following, key fields are denoted by K_0, K_1, ..., K_m, difference fields by D_0, D_1, ..., D_m, and the count field by ct.

1. K_0 contains an arbitrary value, R.
2. For $0 \leq j \leq ct$, $D_j = K_{j+1} - K_j$. (Addition modulo $ct+1$.)

3. Fill values: for $ct < j \leq m$, $D_j = K_j = 1 - 2K_0$.

A first attempt to construct a robust B-tree used pointers between successive leaves and pointers from the leaves to the interior of the tree. It is referred to as a chained and threaded B-tree (CTB-tree) [5]. Its definition, as published previously, follows.

1. A CTB-tree consists of a header and a possibly empty set of leaf nodes and branch 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. Branch nodes consist of an identifier field and a robust contiguous list whose data 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 inorder 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.

A CTB-tree of order 2 is shown in Figure 2.

The CTB-tree is 2-detectable and 1-correctable, and both error detection and error correction can be performed in time $O(n)$ for an n-node tree. The implementation of CTB-trees proved to be rather complex, as will be discussed further in Section 4, so an alternative robust B-tree was sought. One part of the redundancy in a CTB-tree is the set of chain pointers linking the leaf nodes. It seemed appropriate to investigate an implementation in which each level of branch nodes was also linked in this way.

Assuming that we have some way of locating the beginning of each level in the tree, this gives us almost enough information to reconstruct all the B-tree pointers. As we scan across a level, we know that the level above contains exactly one pointer to each node in this level, but we do not know how these pointers are grouped into nodes. We could resolve this simply by putting a flag in the first (or last) son of each branch node, but to provide a more positive indication of the correspondence between levels, we decided to place a thread pointer in the last son of each branch node, pointing to the father of the node. All other thread pointers are null. Note that in a CTB-tree, thread pointers exist only in leaf nodes and can span an arbitrary number of levels in the tree structure, but in this new structure, all thread pointers are to the next level of the tree.

We call this new B-tree structure a linked B-tree (LB-tree). We assume that there is a header node for each level in the tree, but continue to ignore the problem of locating these headers. Then we can define an LB-tree as follows.

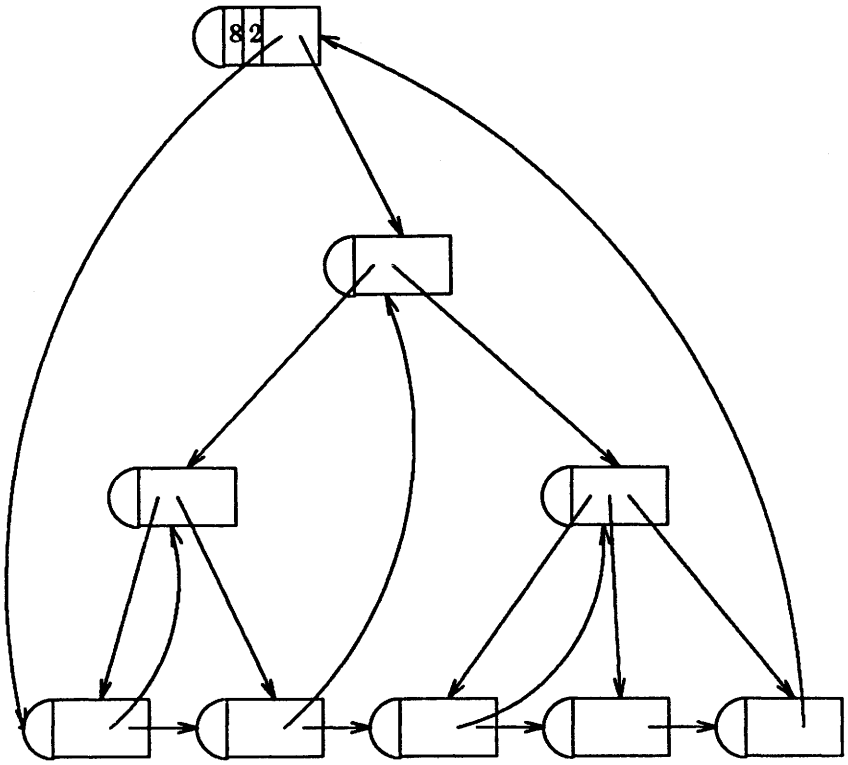

Figure 2. CTB-tree of order 2.

1. An LB-tree consists of a main header, an additional header for each level of the tree, and a possibly empty set of leaf nodes and branch nodes.
2. An LB-tree is a B-tree, as defined above.
3. The main header contains an identifier field, a count field, whose value is the number of leaf and branch nodes connected to the header, and a pointer to the root of the tree.
4. "Level header" nodes contain an identifier field, a pointer to the first (left-most) node on that level, and a count of the number of nodes on that level.
5. Branch nodes consist of an identifier field, a chain pointer, a thread pointer, and a robust contiguous list whose data elements are the B-tree pointers.
6. 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.
7. The chain pointer in a node points to the next node (on the right) on the same level. The chain pointer in the right-most node on a level points back to the level header for that level.

8. The thread pointer in a node which is the rightmost son of a branch node X points to X; the thread pointer in the root points to the main header. All other thread pointers are null.

An LB-tree of order 2 is shown in Figure 3.

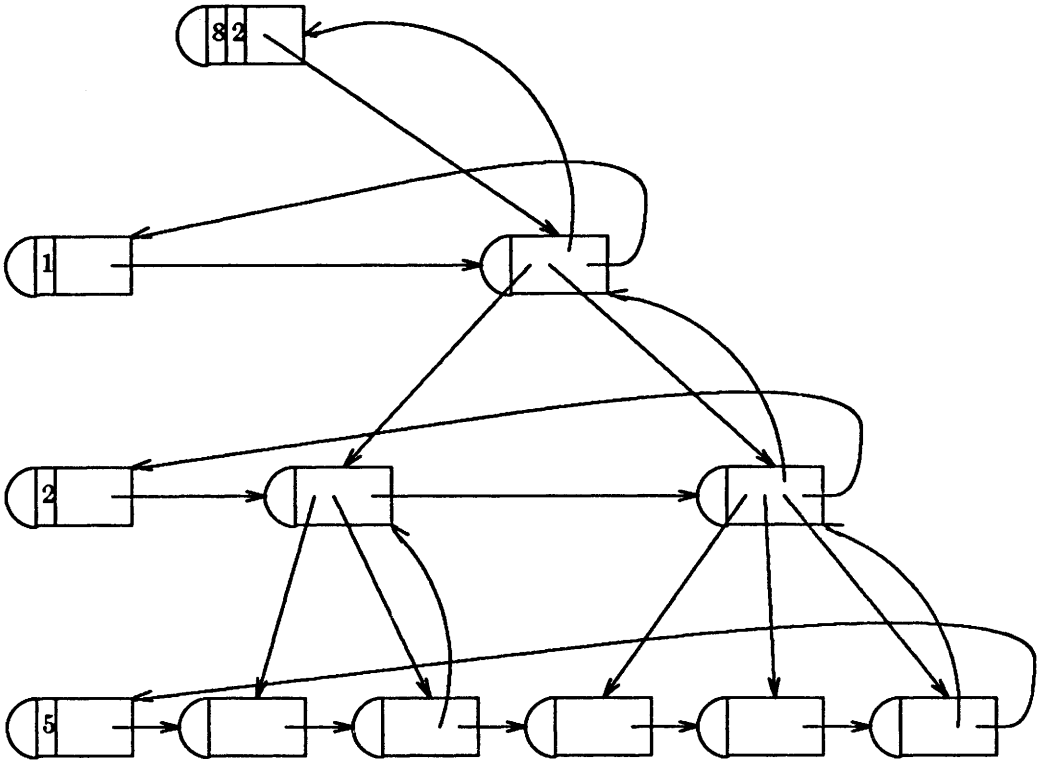

Figure 3. LB-tree of order 2.

The remaining problem is accessing the level headers. Having several headers is not, in itself, a problem. If a storage structure requires k headers, we can simply allocate a contiguous vector of k headers. The difficulty is that the number of headers changes as the height of the tree changes. There are at least three possible solutions.

The first solution is to use a contiguous vector of headers which grows and shrinks as the tree height changes. The obvious difficulty with this solution is that when the vector needs to be extended, the required record may not be free. Thus, this solution is only feasible if it is possible to relocate a record in order to extend the vector of header records. If we have a file containing only an LB-tree, then this is indeed possible.

The second solution is to allocate a contiguous vector of headers initially which is large enough to handle the largest anticipated tree. Since the number of tree nodes is exponential in the height of the tree, this need not represent much wasted space. Indeed, if we have a file of fixed size devoted to storing a single

LB-tree, we can calculate fairly accurately how many headers will be needed when the file is full.

With either of these solutions, presumably the first level header will be for the leaf level and the last level header will be for the root level, in order to avoid moving data between headers when the tree height changes. Also, it is necessary to store the number of levels in the tree. It is important to note that this field, presumably stored in the main header, is crucial in using the level header vector. Thus, error detection and correction routines must deal with this field before making use of the vector of level headers.

The third solution is to put the level headers onto some kind of linked list. The idea is simply that each level header node will contain the data described above plus linked list data connecting the level headers "vertically." We can use any kind of linked list we please, but it is obviously desirable to use one which is as robust as the rest of the LB-tree structure. The robustness properties of an LB-tree are derived in the next section: the modified(2) double-linked list has matching robustness [8].

This last solution is the one selected for our implementation of the LB-tree, with the choice of list structure left to the user. If the user chooses a list structure with k headers, then the LB-tree has $k+1$ headers: the first being the main LB-tree header, as described above, and the remaining headers being the list headers. Each level header node is divided into two sections, the first containing the level header data, the second containing a standard linked list node.

3. Properties of the LB-tree

In this section, we present the important robustness properties of the linked B-tree. We do not attempt to provide formal proofs of the properties, but hope that the informal arguments presented will convince the reader.

First, we wish to show that the pointer structure of the LB-tree is 3-detectable. To do this, we will use the principle of dividing undetectable modifications into three classes: those which do not change the set of nodes in the instance, those which change the set of nodes, but not the number of nodes, and those which change the number of nodes.

1. Rearrangement. Given the constraint on key ordering, any undetectable rearrangement of nodes will be extremely expensive. Ignoring the changes required to keys, the least expensive rearrangement is to exchange two adjacent leaf nodes containing null threads. This requires changing three chain pointers at the leaf level and the two pointers from a branch node to the two leaves. Any change involving branch nodes will be very expensive, since the pointers out of the branch nodes involved will have to be changed, or else changes will have to be made at the next lower level in the tree. Thus, rearrangement requires at least five changes, not counting changes to keys.
2. Replacement. The least expensive replacement involves a single node, since we assume that no node outside the LB-tree can have a correct identifier field. Thus, each node replaced requires a change to an identifier field. The minimum cost for replacing a leaf node occurs when the leaf has a null

thread pointer. In this case, we must change the identifier field and chain pointer in the new leaf node, and the chain and branch node pointers to the leaf. The minimum cost for replacing a branch node occurs for one with two sons. In this case, we must change the identifier field, the chain pointer, and both son pointers in the node, plus the chain and son pointers to the node. For the leaf case, there are four changes; for the branch case, six.

3. Change in number. Deletion is cheaper than insertion, since correct identifier field values do not have to be supplied. There are two possibilities: delete the whole tree and delete a proper subtree. Deleting the whole tree involves changing the root pointer and count in the main header and making all the level headers disappear. We will assume that making the level headers disappear requires at least two changes. (In fact, for the modified(2) double-linked list structure, four changes will be required.) Deleting a proper subtree requires chain pointer changes at the level of the subtree root, and at all lower levels, so the cheapest case is deleting a single leaf. This involves changing the count in the main header and in the level header for the leaf level, a chain pointer at the leaf level, and a branch node pointer to the leaf. Thus, at least four changes are required.

Note that in the above, no reliance has been placed on the presence of an RCL at each node. For example, it is not possible to change a branch node pointer to null simply by altering its value--the RCL will indicate that there should be a non-null pointer in that position. The intention is to show the overall robustness of an LB-tree independent of the representation chosen for the list structure at each node.

Since each of the above cases requires at least four changes, we conclude that the overall pointer structure of an LB-tree is 3-detectable. However, we cannot say that the LB-tree is 3-detectable, because the RCL is only 2-detectable. Thus, if three fields of a leaf node are changed, the resulting tree may be correct but contain a different set of keys. If the RCL were replaced by a more robust structure, we would then have an LB-tree which was 3-detectable.

Having obtained the detectability of the LB-tree, we can apply the General Correction Theorem [10] to obtain the correctability. The General Correction Theorem says that a structure is r-correctable if it is $2r$-detectable and there are $r+1$ edge-disjoint paths to each node. We know that the LB-tree is 2-detectable and there are two edge-disjoint paths to each node, one using the B-tree pointers and one using the level headers and chain pointers, thus we conclude that the LB-tree is 1-correctable.

The General Correction Theorem does not guarantee the existence of an efficient correction algorithm, but Theorem 7.5.1 of [2] does, provided there are also $r+1$ determining sets in the structure. (A *determining set* is a subset of the structural data which allows reconstruction of all other structural data. More details may be found in [9].) In the development of the LB-tree, in the previous section, thread pointers were added so that chains and threads could be used to reconstruct the B-tree pointers. Thus, the B-tree pointers and the chains and threads each form a determining set. (We must also be able to partition the structural data in the level headers. If we use a modified(2) double-linked list, this can be done.) By showing that each of these sets can be used to reconstruct

all other structural information in linear time and applying the theorem mentioned above, we can prove the existence of a linear time correction algorithm.

We would like to show that the LB-tree is locally correctable. A complete treatment, together with a definition of local correctability may be found in [3]. Here, we will simply explain how it is possible to correct a large number of errors in an LB-tree under specific constraints about the relative location of errors. The Appendix contains the algorithm described informally below.

We assume that the level headers form a locally-correctable structure. This is indeed the case if a modified(2) double-linked list is used [8]. Thus, we begin by performing local error correction on the level headers, then proceed to the LB-tree itself. The algorithm scans the tree from left to right, examining two adjacent levels on each pass. On the first pass, it examines the root and the sons of the root, on the second pass, it examines the level below the root and the second level below the root, and on the last pass, it examines the last level of branch nodes and the leaf level.

On each pass, the algorithm follows chain pointers on two levels and examines the B-tree pointers and thread pointers connecting the two levels. The chain pointers on the "upper" level are known to be correct, since they were handled on the previous scan, but all other pointers may contain errors. At each step of the scan, the algorithm attempts to reach the next node on the "lower" level, by following a chain pointer on the lower level, and a B-tree pointer from the upper level. If these agree, the scan continues; if not, the algorithm must decide which pointer is in error. To make the decision, the algorithm advances one more node on the lower level and checks for disagreement. Since an error in a B-tree pointer affects only one node in the traversal sequence, but an error in a chain pointer affects all subsequent nodes, agreement on this next node implies an incorrect B-tree pointer, disagreement implies an incorrect chain pointer.

Once it is known which pointer is in error, that pointer can easily be corrected, by setting it equal to the other pointer. Errors in thread pointers, identifier fields, level header counts, and the main count are also easily handled, once the chain and B-tree pointers are known to be correct.

Consider a sequence of three consecutive nodes on some level, call them B, C, and D (as shown in Figure 4). If there is an error in the B-tree pointer from A to C or the chain pointer from B to C, then we rely on the correctness of the B-tree pointer from A to D and the chain pointer from C to D in order to correct the error. However, assuming that we have not previously encountered some error which could not be corrected, we do not need to rely on the correctness of any other fields in the tree. Thus, provided each such "substructure" consisting of four pointers contains at most one error, we can correct arbitrarily many errors in an LB-tree. Therefore, the LB-tree is called locally correctable.

Having obtained these properties for the LB-tree, we can make a comparison with the CTB-tree. Both structures are 2-detectable and 1-correctable, but we observe that a more robust list structure at each node will make the LB-tree 3-detectable. For the CTB-tree, the detectability cannot be increased without changing the pointer structure.

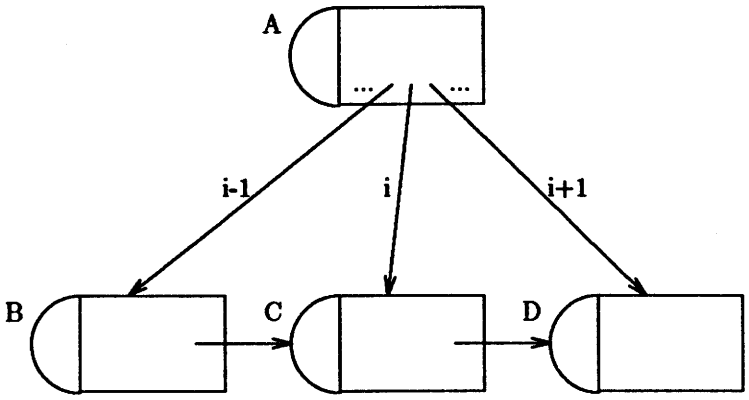

Figure 4. Substructure for local correction in LB-tree.

A more important difference is that the LB-tree is locally correctable and the CTB-tree is not. It is possible to correct many errors in a CTB-tree if, roughly speaking, they are all in distinct subtrees. For example, many errors can be corrected in leaf nodes or the level of branch nodes immediately above the leaves. Unfortunately, an error in the pointer from the header to the root of the tree lies in only one subtree--the entire tree. Such an error can be corrected only if all other pointers are correct.

A further comparison between the two B-tree representations, based on implementation experience, is presented in the next section.

4. Experience with the LB-tree

The previous section showed that the LB-tree was superior to the CTB-tree, if one considered detectability, correctability, and local correctability. The LB-tree requires slightly more storage: for the level headers and for a chain and thread pointer in each branch node. For typical node sizes and numbers of nodes, these additional storage costs are negligible. The remaining point of comparison is ease of implementation, which is addressed in this section.

Both the CTB-tree and the LB-tree have been implemented in the ISSS environment [7]. (In this environment, storage structures can be treated as uniform building blocks, and thus combinations of storage structures can easily be constructed.) Each implementation included insertion and deletion of keys, error detection, and error correction. The deletion routine performs actual removal of keys, with any necessary balancing or merging, rather than simply marking keys as deleted. The LB-tree implementation used the CTB-tree implementation as a starting point, so it is not meaningful to compare programmer-days in performing the two implementations.

A B-tree implementation, even without added redundancy, will be rather complex, for example, see the algorithms in [12]. Thus, the insertion and deletion routines for both of our B-tree implementations are quite bulky. However, the CTB-tree routines are harder to understand, because of the thread structure. In either implementation, when branch nodes are split, merged, or balanced, thread

pointers must be changed. In an LB-tree, these pointers are located in the next level of the tree and are easily accessed. In a CTB-tree, these pointers are located in the leaves: accessing them adds significant mystery to the routines. Also, our implementations are designed to work for B-trees of arbitrary order, and deletion from a CTB-tree of order one produces special cases which do not arise in the LB-tree implementation. In practice, this last point may not be very important, since B-trees in data base systems normally have quite large orders.

The difference between the correction routines for the two structures is dramatic. The CTB-tree has a detection routine of 200 lines and a correction routine of 1500 lines. For the LB-tree, a single routine of 1000 lines performs both detection and correction, a flag argument being used to indicate which is desired. (Of course, each of the "routines" is actually a collection of functions, not a single function.) The CTB-tree correction routine can correct only a single error, whereas the LB-tree correction routine is a local correction routine, which can correct any single error and many cases of multiple errors.

Not only is the LB-tree correction routine much smaller than the CTB-tree correction routine, it is much easier to understand. Given the description of local correction in the preceding section, it is quite easy to read the code and understand how the routine works. The CTB-tree correction routine is based on two parallel traversals, and corrects errors based on disagreement between the two traversals [8]. In some cases, it is not possible to reach a definite conclusion about the error discovered, so the routine must make a guess and then check whether the guess was correct. Even given a fairly thorough understanding of the correction principles being used, the CTB-tree correction routine is very hard to understand.

Both correction routines take $O(n)$ time for a tree of n nodes. The LB-tree correction routine reads each branch node twice and each leaf node once. The CTB-tree correction routine reads each node in the tree once, and reads a subset of the nodes two additional times. The size of the subset depends on the location of the error. In a B-tree of large order, most of the nodes are leaf nodes, so the LB-tree correction routine will perform fewer reads than the CTB-tree correction routine, except when the CTB-tree correction routine encounters a very favourably located error.

Finally, it should be noted that because we were performing our implementation of the LB-tree in the ISSS environment, we could make use of an existing linked list implementation, including its correction routines. Thus, the size given above for the LB-tree correction routine does not include the modified(2) double-linked list correction routine. This may introduce some bias in favour of the LB-tree, but it is very difficult to include the linked list code as part of the LB-tree and obtain a fair comparison. The difficulty is that the linked list implementation is very general and would be much less complex if we had implemented it just for the purpose of chaining together LB-tree level headers. Also, the use of a contiguous vector of level headers would make error detection and correction for the level headers quite simple.

5. Conclusions

It seems clear from the discussion in the two preceding sections that the LB-tree is superior to the CTB-tree. It is easier to implement, both with respect to insertion and deletion and with respect to error detection and correction. It is also possible to perform local error correction in an LB-tree, whereas only global error correction is possible in a CTB-tree. Thus, we believe that if a robust B-tree is desired, the LB-tree should be chosen in preference to the CTB-tree.

The only likely problem in implementing the LB-tree is in managing the sequence of level headers. For our purposes, a linked representation seemed appropriate, but we suspect that a contiguous vector of fixed size will be appropriate in many cases. Such an approach should make management of the level headers quite straightforward.

There are a number of areas which warrant further study. One such area is the investigation of alternative redundancy strategies in B-trees. In particular, Vandendorpe [11] has proposed a different robust B-tree. He is concerned principally with the problem of updates which do not complete, for example, because of a system crash. Therefore, his B-tree contains redundancy intended to allow partially completed correct updates to be discovered and then successfully completed. His objectives are thus different from ours, but it may be useful to look at his structure in the light of our requirements, and look at our structures in the light of his requirements.

Another area for future investigation is concurrency in accessing robust B-trees. We implicitly assume that the entire tree is locked for update during the execution of an error detection or correction routine. It might be possible to lock only part of the tree at any time, but since the LB-tree detection and correction routine moves through the tree in an unconventional pattern, interactions with update routines would have to be investigated carefully, in particular for the possibility of deadlock. Another possibility would be to unlock the tree at the end of each level, in order to allow updates to proceed. This would make the assumption that the upper level chain pointers are correct somewhat suspect. More seriously, since most of the work occurs when scanning the leaf level, this wouldn't help to reduce the maximum locking interval very much.

Thus, we believe that the LB-tree is a useful robust B-tree implementation, appropriate for use in systems which require a degree of fault tolerance. However, further investigation is required concerning concurrency aspects of the implementation.

ACKNOWLEDGEMENTS

The error detection and correction algorithm for the LB-tree was implemented by M. V. Ramakrishna. Several programmers also helped in the implementation of CTB-trees and LB-trees: Debbie Cullon, Mark Ingram, and Leslie Park.

This work was supported by the Natural Sciences and Engineering Research Council of Canada under Grant A3078 and a Postgraduate Scholarship.

References

1. R. Bayer and C. McCreight, "Organisation and maintenance of large ordered indexes," *Acta Informatica* **1**(3) pp. 173-189 (1972).
2. J. P. Black, *Analysis and Design of Systems of Robust Storage Structures,* Ph. D. Thesis, University of Waterloo, Ontario, Canada (July 1982).
3. J. P. Black and D. J. Taylor, "Local correctability in robust storage structures," CS-84-44, Dept. of Computer Science, University of Waterloo (December 1984).
4. J. P. Black, D. J. Taylor, and D. E. Morgan, "An introduction to robust data structures," *Digest of Papers: Tenth Annual International Symposium on Fault-Tolerant Computing,* pp. 110-112 (1-3 October 1980).
5. J. P. Black, D. J. Taylor, and D. E. Morgan, "A robust B-tree implementation," *Proceedings of the Fifth International Conference on Software Engineering,* pp. 63-70 (9-12 March 1981).
6. D. Comer, "The ubiquitous B-tree," *Computing Surveys* **11**(2) pp. 121-137 (June 1979).
7. D. J. Taylor and J. P. Black, "Experimentation with data structures," CS-84-52, Dept. of Computer Science, University of Waterloo (December 1984).
8. D. J. Taylor and J. P. Black, "Principles of data structure error correction," *IEEE Transactions on Computers* **C-31**(7) pp. 602-608 (July 1982).
9. D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in data structures: Improving software fault tolerance," *IEEE Transactions on Software Engineering* **SE-6**(6) pp. 585-594 (November 1980).
10. D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in data structures: Some theoretical results," *IEEE Transactions on Software Engineering* **SE-6**(6) pp. 595-602 (November 1980).
11. J. E. Vandendorpe, *A crash tolerant B-tree structure for database retrieval systems,* Ph.D. Thesis, Illinois Institute of Technology (December 1980).
12. N. Wirth, *Algorithms + Data Structures = Programs,* Prentice-Hall, Englewood Cliffs, N.J. (1976).

APPENDIX

The algorithms for insertion and deletion in LB-trees are straightforward modifications of the standard algorithms, as found in [12] for example. Thus, this appendix contains only the correction algorithm for LB-trees. The actual correction code is much too bulky to present here. The bulk arises largely from the necessity to read and write nodes of the B-tree, so for purposes of displaying the correction algorithm, it is assumed that the B-tree is in main storage.

The code presented is C, but constructions peculiar to C have been avoided. (Note that in C, '&' represents the address of its operand, '*' dereferences a pointer, '++' is an increment operation, and '--' a decrement operation.) One point which may require explanation is the use of the macro FORCE. It simply tests its two arguments, and if they are unequal copies the second argument into the first. Although the same effect could be achieved with a simple assignment statement, FORCE is used for two reasons. The first reason is that in a realistic implementation, it would be necessary to rewrite records which had been modified, log the correction, and possibly other activities. Although such actions are not included here, the use of FORCE indicates where they would be required. The second reason is simply to highlight the assignments which are corrections, so FORCE is used even in cases for which the equality test is redundant.

The code related to the level header structure is omitted. The existence of four functions is assumed: *corr_lvl_hdr* which checks and corrects the level header structure, *lvl_nodecnt* which counts the level headers, *first_lvl_hdr* which returns a pointer to the first level header, and *next_lvl_hdr* which returns a pointer to the next level header. These functions clearly require parameters, but since the precise level header structure is not specified, the parameters are omitted. As well, the function which corrects the Robust Contiguous List structure at each node is omitted. The routine is named *corr_rcl* and is passed a single parameter, the address of a node.

The code below is organised as four functions: *correct* is the main routine for LB-tree correction, *corr_sub* corrects one level of the LB-tree structure, *corr_root* handles the special case of the root node, and *vote* performs the "voting" algorithm to check, and possibly correct, errors in tree and chain pointers. The following field names are used for components of nodes: *rt* the root pointer in the header, *hdr_cnt* the count field in the header and the level headers, *key* the key (level number) field in the level headers, *chain* the chain pointer in the level headers and B-tree nodes, *thread* the thread pointer in the B-tree nodes, *br* the vector of B-tree pointers in B-tree branch nodes, and *id* the identifier field in all nodes (with value HDR_ID, LVL_ID, BRANCH_ID, or LEAF_ID, as appropriate). In addition, the following user-defined types are used: HDR for the main header, LVLHDR for the level headers, and NODE for an arbitrary node (other than the main header).

```
#define FORCE(a, b) \
        if (a != b) { \
                a = b; \
        }

correct(hdr)  /*  main routine for LB-tree correction  */
HDR *hdr;
{
        int     maxlvl, lvlno, node_cnt, lvl_ncnt;

        if (corr_lvl_hdr() < 0) return(-1);
        FORCE(hdr—>id, HDR_ID);
        maxlvl = lvl_nodecnt();

        /*  special case—null tree  */

        if (maxlvl == 0) {
                FORCE(hdr—>rt, hdr);
                FORCE(hdr—>hdr_cnt, 0);
                return(0);
        }

        /*  check root  */

        lvlhdr = first_lvl_hdr();
        corr_root(hdr, lvlhdr, maxlvl);

        /*  main loop  */

        node_cnt = 1;
        for (lvlno = maxlvl-1; lvlno > 0; --lvlno) {
                prevhdr = lvlhdr;
                lvlhdr = next_lvl_hdr();
                if (corr_sub(lvlhdr, prevhdr, &lvl_ncnt, lvlno) < 0) return(-1);
                node_cnt += lvl_ncnt;
                FORCE(lvlhdr—>id, LVL_ID);
                FORCE(lvlhdr—>key, lvlno);
                FORCE(lvlhdr—>hdr_cnt, lvl_ncnt);
        }
        FORCE(hdr—>hdr_cnt, node_cnt);
        return(0);
}
```

```
corr_sub(lvlhdr, prevhdr, lvl_ncnt, lvlno)  /*  correct one level of the tree  */
LVLHDR  *lvlhdr, *prevhdr;
int     *lvl_ncnt, lvlno;
{
    NODE    *a_node, *b_node, *c_node, *d_node;
    int     endlvl, index, thrflag;

    a_node = prevhdr—>chain;
    b_node = lvlhdr;
    endlvl = *lvl_ncnt = index = 0;
    while (!endlvl) {
        if (index < a_node—>n) {
            thrflag = 0;
            d_node = a_node—>br[index+1];
        } else {  /*  have reached end of node on "upper" level  */
            next_a = a_node—>chain;
            thrflag = 1;
            if (next_a == prevhdr) {  /*  end of entire level  */
                endlvl = 1;
                d_node = lvlhdr;
            } else {
                d_node = a_node—>br[0];
            }
        }
        if (vote(a_node, &(a_node—>br[index]), b_node, d_node, thrflag)) {
            return(-1);
        }
        c_node = a_node—>br[index];
        corr_rcl(c_node);

        /*  determine type of node on lower level
         *  and correct its identifier field
         */
        if (lvlno == 1) {
            FORCE(c_node—>id, LEAF_ID);
        } else {
            FORCE(c_node—>id, BRANCH_ID);
        }

        /*  thrflag indicates whether there should be a thread
         *  in the lower level node
         */
        if (thrflag) {
            FORCE(c_node—>thread, a_node);
        } else {
            FORCE(c_node—>thread, NULL);
        }
```

```
            /* at end of level, fix last chain pointer */
            if (endlvl) {
                FORCE(c_node->chain, lvlhdr);
            }

            /* thrflag also indicates whether the last pointer
             * has been reached in the upper level node
             */
            if (thrflag) {
                index = 0;
                a_node = next_a;
            } else {
                ++index;
            }
            b_node = c_node;
            ++*lvl_ncnt;
        }
        return(0);
}

corr_root(hdr, lvlhdr, maxlvl) /* correct root node */
HDR     *hdr;
LVLHDR  lvlhdr;
int     maxlvl;
{
        /* locate root */
        if (vote(hdr, &(hdr->rt), lvlhdr->chain, lvlhdr, 1)) {
            return(-1);
        }
        if (corr_rcl(hdr->rt) < 0) return(-1);
        FORCE(hdr->rt->chain, lvlhdr);
        FORCE(hdr->rt->thread, hdr);

        /* root id should be leaf id iff one level in tree */
        if (maxlvl == 1) {
            FORCE(hdr->rt->id, LEAF_ID);
        } else {
            FORCE(hdr->rt->id, BRANCH_ID);
        }

        /* fix first level header */
        FORCE(lvlhdr->id, LVL_ID);
        FORCE(lvlhdr->hdr_cnt, 1);
        FORCE(lvlhdr->key, maxlvl);
        return(0);
}
```

```
/*
 * Vote on the next node in the traversal:  *a_ptr and b_ptr—>chain
 * are the two alternatives. d_node should be the second next node,
 * used to check chain pointers in alternatives.
 * a_node is the parent node, used to check thread pointers (if any)
 * in the alternatives.
 */

vote(a_node, a_ptr, b_ptr, d_node, thrflag)
NODE    *a_node, **a_ptr, *b_ptr, *d_node;
int     thrflag;
{
        int     vote_a, vote_b;

        if (*a_ptr != b_ptr—>chain) {
                vote_a = vote_b = 0;
                if((*a_ptr)—>chain == d_node) ++vote_a;
                if (thrflag && (*a_ptr)—>thread == a_node) ++vote_a;
                if (b_ptr—>chain == d_node) ++vote_b;
                if (thrflag && b_ptr—>chain—>thread == a_node) ++vote_b;
                if (vote_a > vote_b) {  /*  more votes for a:  it is correct  */
                        FORCE(b_ptr—>chain, *a_ptr);
                } else
                if (vote_b > vote_a) {  /*  more votes for b:  it is correct  */
                        FORCE(*a_ptr, b_ptr—>chain);
                } else {  /*  a tie:  multiple errors in substructure  */
                        return(-1);
                }
        }
        return(0);
}
```