Crash Recovery for Binary Trees

D.J. Taylor

Research Report
CS-86-66

December 1986

Crash Recovery for Binary Trees

David J. Taylor

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

ABSTRACT

A robust storage structure is intended to provide the ability to detect and possibly correct damage to the structure. Previous papers have described various binary tree implementations which provide some degree of robustness. This paper examines the usefulness of such robust binary trees when the source of damage is the partial completion of an update operation, due to a "crash" of the program or system performing the update. General principles concerning the use of robust storage structures for crash recovery are applied to binary trees: in most cases, crash recovery can be easily accomplished. One case of deletion causes significant problems, so alternative techniques for deletion, which allow crash recovery, are described and evaluated.

1. Introduction

Various robust binary trees have been described in the literature: the chained and threaded binary tree [8], the double binary tree [4], the mod(2) chained and threaded binary tree [5], and the chained and threaded binary tree with LR-tagging [6]. Each of these is intended to provide the ability to detect, and possibly correct, damage to the tree representation. The detection and correction capability is expressed in terms of the total number of components which can be damaged while still guaranteeing error detection or correction.

When a correct update routine is unable to complete for any reason, possibly because of a system crash, the resulting partially updated instance can be viewed as being the "before" instance containing a certain set of damaged components, or the "after" instance with a different set of damaged components. Thus, it is possible to use standard detection and correction procedures to attempt to repair the damage, and hence recover from the effects of the crash.

A previous paper has provided a basic foundation concerning the use of robust storage structures for crash recovery [9]. That paper also provided an extensive analysis of crash recovery in linked list structures, but did not consider other structures. This paper applies the results to binary trees, and also makes a small but important extension to the basic theory in order to deal effectively with binary trees. The paper concentrates on the "basic" chained and threaded binary tree, but briefly discusses the other three tree structures listed above.

In the next section, the essential previous terminology and results are summarised. In Section 3, these results are applied to the chained and threaded binary tree, to show that crash recovery can be easily accomplished for most update operations. In Section 4, the difficult update operation (deletion of a node with two sons) is considered. To cope with this case, the definition of crash recovery is relaxed slightly and the update operation is performed in an unusual manner. Finally, in Section 5, the results are summarised, crash recovery in other binary trees is discussed briefly, and areas for further work are outlined.

2. Previous Definitions and Results

Crash recovery attempts to deal with partially completed update operations. The instance which existed prior to the update operation is referred to as the *before instance,* the instance which would have been created if the update operation had completed successfully is referred to as the *after instance,* and the instance which actually exists, because of the crash, is referred to as the *partially updated instance*. Crash recovery is *successful* if a correction (or other) procedure applied to the partially updated instance yields either the before or after instance. (If the before instance can be recreated, then the update operation can be repeated.)

If the update operation is the insertion of a new node, then denote the number of components changed in the new node by c_1 and the total number of other changes made by the update routine by c_2. Denote the number of changes made to components in node M, a node in the before instance, by m_M. Denote the number of pointers to the new node which are *not* in node M by p_M. Denote the total number of pointers to the new node by p. Then, the following are the two main results of the previous paper on crash recovery.

Theorem 1: For an instance in main storage, recovery by an r-correction routine from a crash during an insertion operation can be guaranteed iff (a) $c_1+c_2 \leq 2r+1$ or (b) $c_2 \leq 2r+1$ and $p \leq r+1$.

Theorem 2: For an instance on external storage, recovery by an r-correction routine from a crash during an insertion operation can be guaranteed iff there exists a node M such that (a) $c_1+c_2-m_M \leq 2r$ or (b) M contains a pointer to the new node, $c_2-m_M \leq 2r$, and $p_M \leq r$.

Correction routines are described extensively elsewhere [6,7,8]. For purposes of this paper, the only important property is that an r-correction routine is guaranteed to correct up to r erroneous components in an instance. If more than r components are erroneous, its behaviour is unspecified.

These definitions and results can easily be restated for operations other than insertion. For deletion, change "new node" to "node to be deleted" throughout, and require M to be a node in the after instance. For modifications which do not change the set of nodes in the instance, let $c_1=0$ and $p=p_M=0$. Here, requiring M to be a node in the before instance is equivalent to requiring it to be a node in the after instance. Note that for modifications which do not change the set of nodes in the instance, the two cases of Theorem 1 become identical and the second case of Theorem 2 is not applicable.

Details concerning the above results may be found in the referenced paper, but it is useful to mention why the results are different for main storage and external storage. In main storage, as soon as a component is changed, the change becomes visible. On external storage, a node may be read and various components changed. Then, when the node is rewritten, all those changes appear to occur simultaneously. Because of this, crash recovery in main storage is a stricter requirement than crash recovery on external storage. If the conditions of Theorem 1 are satisfied, the conditions of Theorem 2 are also necessarily satisfied.

3. Chained and Threaded Binary Trees

A chained and threaded binary tree is a robust binary tree in which otherwise unused pointer fields are used to increase the robustness of the storage structure. Specifically, each unused right pointer is used to store a pointer to the inorder successor of the node (a *thread* pointer) and unused left pointers are used to form a single-linked list, in inorder, of all nodes having no left subtree (*chain* pointers). Each pointer must be tagged to indicate whether it is a "normal" tree pointer or a chain or thread pointer. This tag bit is assumed to be stored in the same word as the pointer value, so that both may be modified (correctly or erroneously) together. In addition, each node contains an identifier field, whose value uniquely identifies the node as belonging to a specific binary tree instance. A chained and threaded binary tree has a header containing an identifier field, a count of the number of non-header nodes in the instance, a right pointer to the root of the tree, and a left pointer to the first node with no left subtree. The last inorder node has a thread pointer to the header and the last inorder node with no left subtree has a chain pointer to the header. An example of a chained and threaded binary tree is shown in Figure 1. Tree pointers are shown as solid lines. Thread and chain pointers are shown as dashed lines. The identifier field value is shown as "ID".

The use of right pointers as threads is a standard technique which has various advantages, including simpler traversal of the tree [2, pp319-320]. The use of left pointers as chains is a novel aspect of the chained and threaded binary tree, but is inspired by a technique often used in B-trees. A chained and threaded binary tree is 1-correctable [8], that is, any single error in structural data can be corrected.

Insertion of new keys into such a binary tree is quite easy. There are two slightly different cases, depending on whether the new node inserted appears as a left or a right son. If 11 is inserted into the tree shown in Figure 1, it becomes a left son. Insertion requires changing all components in the new node, plus the left pointer in the node containing 12, the left pointer in the node containing 7, and the count in the header. If 13 is inserted into the tree shown in Figure 1, it becomes a right son. Insertion requires changing all components in the new node, plus the right pointer in the node containing 12, the left pointer in the node containing 12, and the count in the header. When inserting to the right, the two pointers changed in existing nodes need not be in the same node, but otherwise these two cases demonstrate the general pattern for insertion. Actual insertion algorithms must deal with the problem of locating all the nodes involved, but here the only concern is the number of pointers changed and whether they are in different nodes.

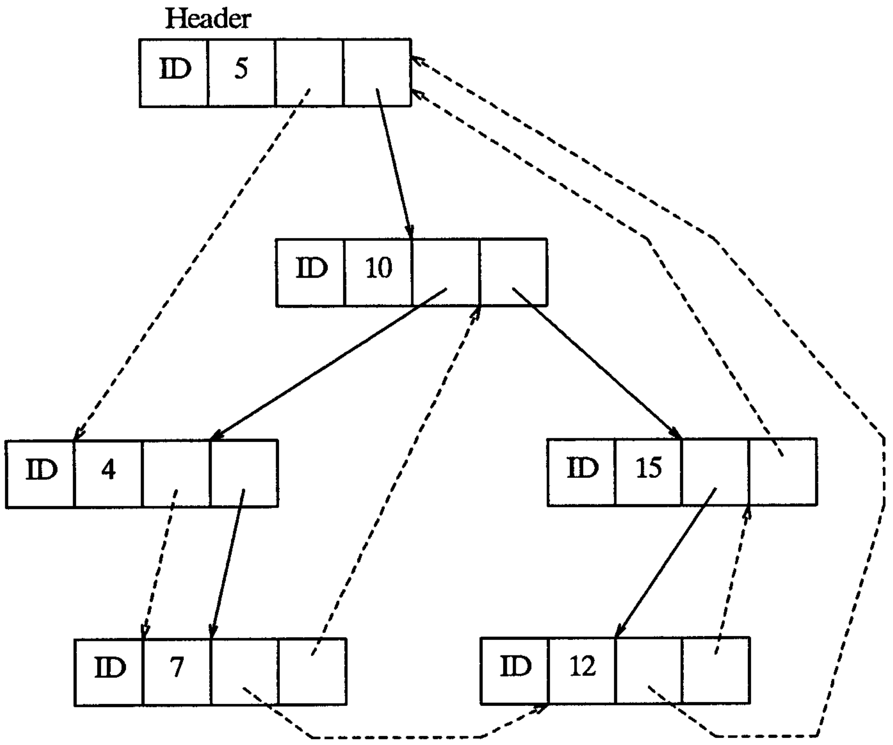

Figure 1. A chained and threaded binary tree

In both cases, the total number of changes in the new node is four and there are three other changes. The number of pointers to the new node is two. Thus, in the notation given in the preceding section, c_1=4, c_2=3, p=2. Considering the (stricter) requirements for crash recovery in main storage, both $c_2 \leq 2r+1$ and $p \leq r+1$ are satisfied, so crash recovery can be guaranteed. The proof for the theorem effectively describes the order to change components in order to achieve crash recovery: first, all changes in the new node, then the count, then one of the pointers to the new node, and finally the other pointer to the new node. Before changing the count, no changes are visible to a correction algorithm examining the binary tree, so recovery trivially takes place to the before instance. After changing the count, the correction routine will simply change the count back, again giving recovery to the before instance. After creating the first pointer to the new node only one change remains, so the correction routine will make that change, giving recovery to the after instance. After creating the second pointer to the new node, the insertion is complete, so recovery trivially takes place to the after instance.

As discussed more fully in the earlier paper on crash recovery, recovery to the before instance may leave a node which is not part of the instance but also no longer is on a free space list. Some cleanup of such nodes is required in addition to the correction routine which is used for the instance.

For deletion, there are three essentially different cases: deletion of a leaf, deletion of a node with one son, and deletion of a node with two sons. Deletion of a leaf is exactly symmetric with insertion, so crash recovery can be guaranteed, as it is for insertion.

There are two cases for the deletion of a node with one son, depending on whether the node has a right or a left son. In the tree of Figure 1, the node containing 15 has only a left son. Deleting 15 requires changing the right pointer in the node containing 10 to point to the node containing 12, changing the thread pointer in the node containing 12 to point to the header, changing the count, and changing the identifier and pointer fields in the deleted node. In the tree of Figure 1, the node containing 4 has only a right son. Deleting 4 requires changing the left pointer in the node containing 10 to point to the node containing 7, changing the left pointer in the header to point to the node containing 7, changing the count, and changing the identifier and pointer fields in the deleted node.

It is necessary to change the identifier field and pointer fields in a node which is removed from an instance, because an assumption underlying detection and correction routines is that identifier values for an instance and pointers into an instance occur only within the instance. (The precise assumption, the Valid State Hypothesis [1], is slightly weaker than this, but the distinction is unimportant in this context.)

As in the insertion examples given earlier, these two deletion cases give the general pattern for deleting a node with one son. Thus, the relevant parameters for crash recovery are $c_1=3$, $c_2=3$, and $p=2$. These values again satisfy $c_2 \leq 2r+1$ and $p \leq r+1$, so crash recovery in main storage, and hence also on external storage, can be guaranteed. The correct ordering for changes is to change the two pointers which point to the node being deleted, then the count and the components in the deleted node.

The remaining case is deletion of a node with two sons, such as the node containing 10 in the tree of Figure 1. The usual algorithm for deleting such nodes is to locate the inorder predecessor (or successor) of the node, which is necessarily not a node with two sons, delete it, and then use it to replace the node containing the key which is to be deleted. To delete key 10, assuming the predecessor node is used, requires the following changes: left and right pointers in the node containing 4, left and right pointers in the node containing 7, right pointer in the header, count, and identifier and both pointers in the deleted node. Here, $c_1=3$, $c_2=6$, and $p=2$. Thus, $c_1+c_2 \leq 2r+1$ is false and $c_2 \leq 2r+1$ is also false, so crash recovery in main storage cannot be guaranteed. For external storage, various choices of node M are possible. In all cases, the number of fields changed in a node other than the node being deleted is at most 2, so $m_M \leq 2$. Hence, $c_1+c_2-m_M \leq 2r$ is false for all M and $c_2-m_M \leq 2r$ is also false for all M. Thus, crash recovery also cannot be guaranteed on external storage.

If the successor is used rather than the predecessor, there are some small differences, but crash recovery still cannot be guaranteed. There are also special situations, such as one of the sons being a leaf, but the existence of such special cases does not help the case described above.

An alternative which is sometimes used is to copy the key (and any associated data) from one node to another, rather than moving a node. Since the correction routine does not deal with key or data fields, this alternative is not possible here. In particular, if the key and data were copied before the node originally containing the key was deleted, a crash immediately after the copy operation would cause the correction routine to detect an error, because of the duplicated key, but it would be unable to correct it. If the node were deleted before copying the key and data, a crash immediately before the copy operation would leave a correct tree, but with the wrong key deleted.

Thus, using any standard deletion algorithm, crash recovery cannot be guaranteed when a node with two sons is deleted. In the next section, two non-standard deletion techniques are considered which allow crash recovery to be guaranteed. One of the techniques can only guarantee crash recovery if the definition is changed slightly, but the modified definition is intuitively appealing. For the standard algorithms discussed in this section, it seems clear that no modification of the definition which would still match the intuitive meaning of "crash recovery" will guarantee crash recovery by a 1-correction algorithm for these update algorithms.

4. Deletion of Nodes with Two Sons

In this section, alternatives to the standard deletion algorithms are described. These algorithms will handle nodes with zero or one sons in the standard manner, but differ in handling deletion of nodes with two sons.

To achieve crash recovery in main storage, Theorem 1 offers two alternatives. The first alternative is $c_1+c_2 \leq 2r+1$. Here, $2r+1=3$ and $c_1=3$, so this is only possible if $c_2=0$, implying no changes except in the node being deleted, which is clearly impossible. The second alternative is $c_2 \leq 2r+1$ and $p \leq r+1$. Since there are two pointers to any node and $r=1$, the second condition is satisfied, independent of the deletion algorithm. The first inequality requires that the number of changes made outside the node being deleted be at most three. This set of changes must include the count, the tree pointer to the node being deleted, and a thread pointer to the node being deleted. In addition, pointers to the two sons of the deleted node must be created. All this can be accomplished by changing only three components in exactly one way: change the tree pointer to the deleted node to point to the left son of the deleted node and change the thread pointer to the deleted node into a tree pointer to the right son of the deleted node. In the tree shown in Figure 1, the root can be deleted by changing the right pointer in the header to point to the node containing 4 and changing the right pointer in the node containing 7 into a tree pointer to the node containing 15.

Using this unconventional deletion algorithm, the conditions which guarantee crash recovery in main storage, and hence also on external storage, are satisfied. In principle, this completes the discussion. The difficulty is that this deletion algorithm may cause the tree to degenerate quite badly. Figure 2 shows the complete tree on fifteen nodes, and the tree which results from deleting the root. The height of the tree increases significantly, and the effect of a sequence of such deletions is potentially quite serious.

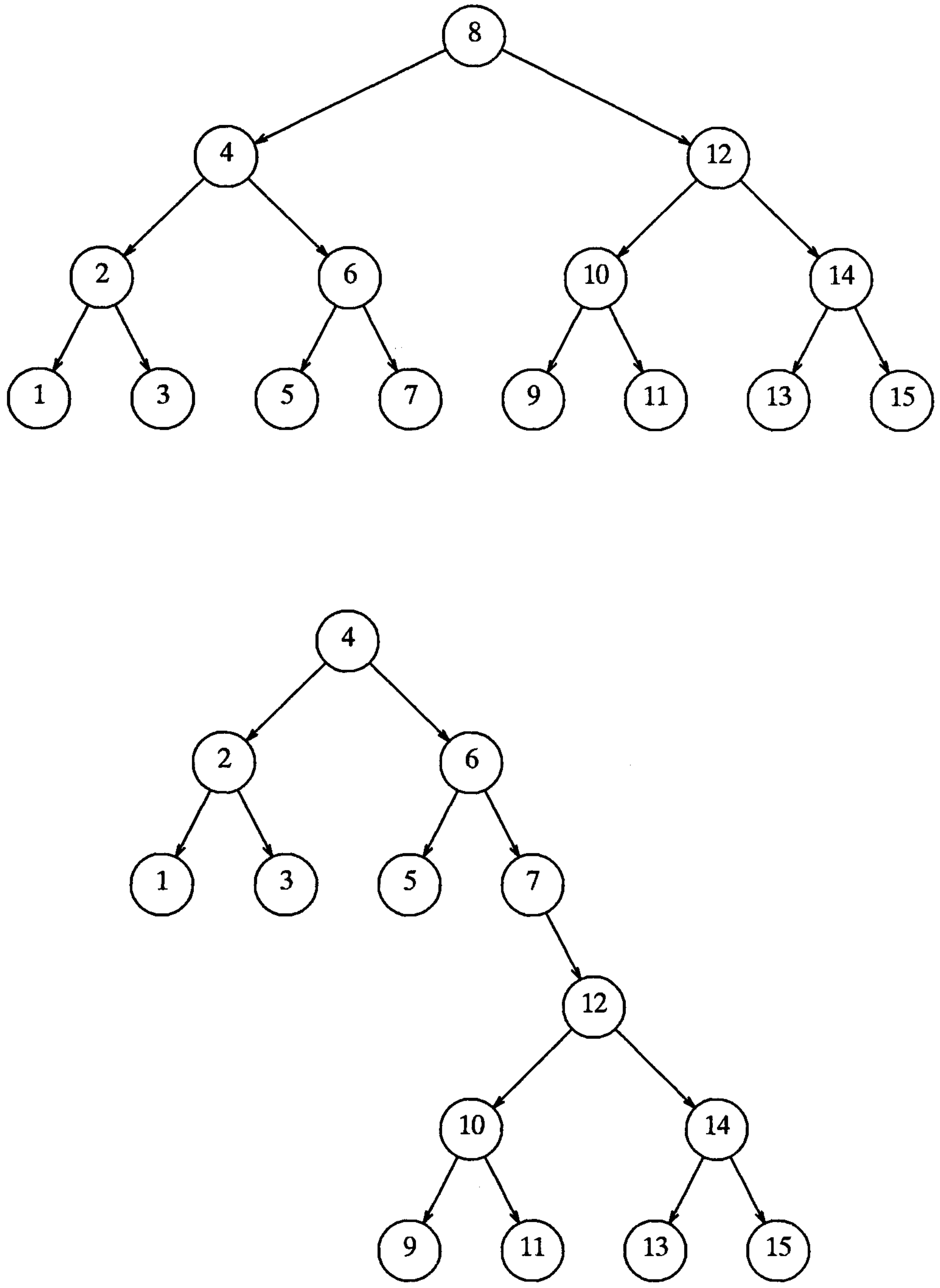

Figure 2. Deleting the root of a balanced tree

An alternative is to attempt to preserve the shape of the tree as much as possible, moving the node to be deleted by a sequence of rotations until it becomes a node without two sons. This presents a problem with respect to the definition of crash recovery. Intuitively, we can recover from a crash during the deletion operation provided crash recovery is possible for each rotation and for the final step during which the deletion is actually performed. Unfortunately, this does not satisfy the definition, so a weaker definition is required.

Thus, the definition in Section 2 is replaced by the following. Crash recovery is *successful* if a correction (or other) procedure applied to the instance yields the after instance, or yields an instance which produces the after instance when the crashed update operation is applied to it. The difference between this definition and the previous one is in recovery to the before instance. This definition only requires recovery to an instance for which the crashed update operation yields the after instance. Satisfying the previous definition automatically satisfies this definition, so the "if" parts of Theorems 1 and 2 remain true with respect to this definition, but the "only if" parts do not. However, each step in this case is constrained exactly according to the conditions of those theorems.

Before giving the deletion algorithm in detail, it is best to describe the general principle. It is possible to change the shape of a search tree by modifying pointers in a small number of nodes, while preserving the inorder of the keys. This phenomenon is essential to updating balanced trees [3, p454]. The simplest such modification is a *single rotation,* which is illustrated in Figure 3.

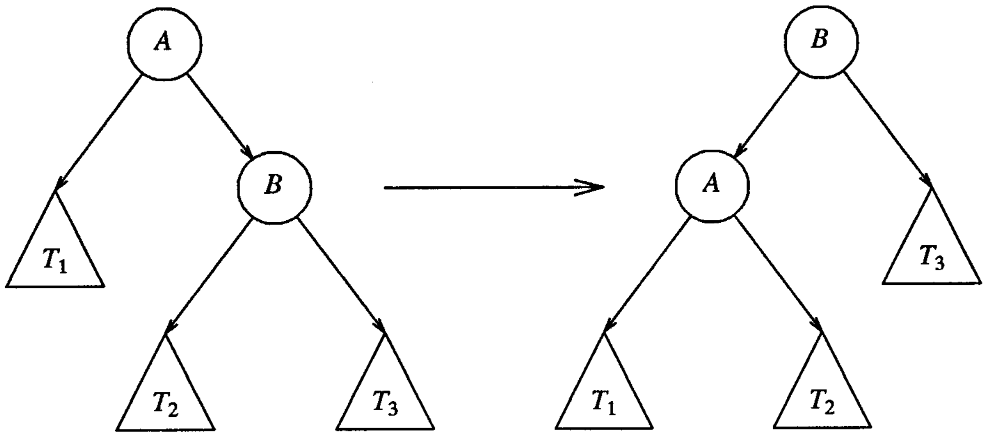

Figure 3. Single rotation to the left.

The rotation shown in the figure moves node A one level away from the root. A sequence of such rotations must eventually cause node A to become a node with less than two sons. This can be proven as follows. Let $h(T)$ denote the height of a tree, and let $S(X)$ denote the sum of the heights of the two subtrees of node X. Then, before the rotation, $S(A)=h(T_1)+\max(h(T_2),h(T_3))+1$, and, after the rotation, $S(A)=h(T_1)+h(T_2)$. Since $S(A)$ strictly decreases when a rotation is performed, a sequence of rotations about the same node must eventually cause one subtree to become empty. If all rotations are to the left, as in the figure, the right subtree will become empty, and all the nodes in subtree T_1 will move one level away from the root for each rotation. This suggests that a better scheme is to perform rotations alternately to the left and to the right, to minimise the increase in tree height.

Using this idea, the following algorithm may be applied repeatedly, as long as the node, X, to be deleted has two sons.

1. If the left son of X has no right son, delete X directly, as shown in Figure 4.

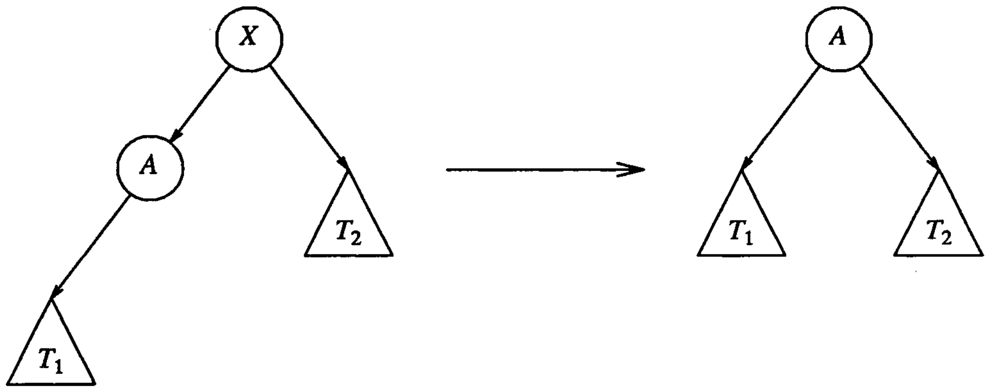

Figure 4. Special case for deletion of node with two sons.

2. If the left son of X has a right son, but the right son of X has no left son, do a single rotation to the right at X.
3. If neither case (1) nor case (2) applies, and X is at an even level (the root or an even number of steps from the root), do a single rotation to the right at X.
4. If neither case (1) nor case (2) applies, and X is at an odd level, do a single rotation to the left at X.

The deletion in case (1) requires changing a pointer in the parent of X, the right pointer in A, and the count, in addition to the pointers and identifier field in X. The left rotations, using the notation of Figure 3, require changing a pointer in the parent of A, the right pointer in A, and the left pointer in B. The right rotations are symmetric. If T_2 is null, then, on a left rotation, the left pointer of B changes from a chain to a tree pointer and the right pointer of A changes from a tree pointer to a thread. In all other cases, pointers simply change from one tree pointer to another, so no problems occur for the chain and thread structure. In the case of a null T_2, the disappearance (or appearance) of a chain pointer necessitates an additional pointer change, in the preceding node with a chain pointer. This additional change will make crash recovery impossible, so rotations involving a null T_2 are avoided by cases (1) and (2), above.

The deletion in case (1) and all the rotations satisfy the conditions for crash recovery in main storage, so according to the revised definition presented earlier in this section, crash recovery is guaranteed for this deletion operation. In order to satisfy the definition, it is necessary that re-applying the deletion operation, after crash recovery, yield exactly the same instance. The deletion algorithm is carefully specified so that the sequence of rotations will be exactly the same. That is, which of the four cases is selected does not depend on previous rotations performed by the algorithm, but only on the level of the node in the tree and the existence of null subtrees at particular locations near the node being deleted.

5. Conclusions and Further Work

Sections 3 and 4 have shown that it is possible to guarantee crash recovery when performing insertions and deletions in binary trees, but only by using an unconventional deletion algorithm. One deletion algorithm allows the original definition of crash recovery to be satisfied, but can badly degrade the shape of the tree. Another deletion algorithm does not change the shape of the tree as drastically, but requires a somewhat relaxed definition of crash recovery. It also may require significantly more effort to accomplish a deletion.

Crash recovery depends on the existence of a correction algorithm. A correction algorithm for chained and threaded binary trees has been published [6]. Unfortunately, the chained and threaded binary tree for which that correction algorithm works uses a different pointer tagging scheme from the one described here. In that tree, the types of the pointers in a node are indicated by the identifier field of the node and each pointer is tagged to indicate the types of pointers in the node it points to. This tagging scheme was adopted to simplify the correction algorithm. Unfortunately, when a node is inserted or deleted, it is then necessary to change additional tag values, and these additional changes violate the conditions for crash recovery.

Thus, to achieve effective crash recovery, it was necessary to create a new correction algorithm for chained and threaded binary trees, using the simple pointer tagging scheme. For reasons of space, it is not possible to describe the new algorithm here, but it is based on the correction algorithm cited above, with additional guessing to compensate for the reduced information available from tags.

Since the chained and threaded binary tree with LR-tagging [6] uses the more complex pointer tagging scheme and adds further tag information to identifier fields, to indicate whether a node is a left or right son, it requires even more changes for insertion and deletion. Thus, crash recovery cannot be guaranteed for LR-tagged trees.

The mod(2) chained and threaded binary tree [5] has a more complicated chain structure, requiring an additional chain pointer to be changed when a node is inserted. Since this tree has a greater detectability, but is still only 1-correctable, crash recovery also cannot be guaranteed for mod(2) chained and threaded binary trees.

The double binary tree [4] uses a very different redundancy scheme from the above trees, but is also 1-correctable. A detailed analysis has not yet been performed, but deletion in such trees is quite difficult, apparently requiring many changes in some cases, so it appears that crash recovery also cannot be guaranteed in double binary trees.

Thus, only the simple-tagged implementation of the chained and threaded binary tree allows crash recovery to be guaranteed. The other trees have certain advantages but no greater correctability, and a cost of those advantages is additional changes for update operations. Those additional changes prevent crash recovery.

In the case of linked lists, there are many robust storage structures available to study, including a variety of locally correctable lists. For linked lists, local correction proved remarkably successful for crash recovery. To continue the study of crash recovery in binary trees, it will be necessary to develop and study alternative robust binary trees. The study of linked lists has suggested that massive redundancy often prevents crash recovery, but it should be possible to devise a binary tree storage structure which is more robust than the simple chained and threaded binary tree and still allows crash recovery. It would also be beneficial if such a structure had a less complicated correction routine.

6. References

1. J. P. Black, *Analysis and Design of Systems of Robust Storage Structures,* Ph.D. Thesis, University of Waterloo, Ontario, Canada (July 1982).
2. D. E. Knuth, *The Art of Computer Programming, vol. 1: Fundamental Algorithms,* Addison-Wesley, Reading, Mass. (1973). 2nd ed.
3. D. E. Knuth, *The Art of Computer Programming, vol. 3: Sorting and Searching,* Addison-Wesley, Reading, Mass. (1973).
4. J. I. Munro and P. V. Poblete, Fault tolerance and storage reduction in binary search trees, *Information and Control* **62**(2/3) pp. 210-218 (August/September 1984).
5. S. C. Seth and R. Muralidhar, Analysis and design of robust data structures, *Digest of Papers: Fifteenth Annual International Symposium on Fault Tolerant Computing*, pp. 14-19 (19-21 June 1985).
6. D. J. Taylor and J. P. Black, Guidelines for storage structure error correction, *Digest of Papers: Fifteenth Annual International Symposium on Fault-Tolerant Computing*, pp. 20-22 (19-21 June 1985).
7. 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).
8. 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).
9. D. J. Taylor and C.-J. H. Seger, Robust storage structures for crash recovery, *IEEE Transactions on Computers* **C-35**(4) pp. 288-295 (April 1986).