A Locally Correctable AVL Tree

I. J. Davis

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

ABSTRACT

Classical binary search trees are not robust, since single errors are often undetectable. Several methods of adding redundancy to binary trees have been proposed that allow a small constant number of errors to be corrected in the resulting structure. This paper presents a method of adding redundancy to an AVL tree that allows the resulting structure to be corrected whenever a small constant number of correct components exist within the vicinity of each error encountered. The expected number of random errors that can be corrected therefore increases as the structure grows. Insertion, deletion, and retrieval continue to be logarithmic operations, while correction can be accomplished in linear time, using logarithmic space.

1. Introduction

A binary tree is a storage structure which allows rapid retrieval of data. The structure comprises a collection of *nodes* that each contain two *link* pointers and a *key*. Each node with the exception of the *header* node is addressed by exactly one link residing in its *parent* node. Obviously, since the structure is finite, some links are unused. These links generally contain some special value indicating that they are null. In a binary search tree the keys within the structure are arranged in such a way that all keys reached by following a "left" link out of any node are smaller than the key recorded in that node, while all keys reached by following a "right" link are larger than this key.

Classical binary trees are not robust. Errors in keys are undetectable, unless these errors affect the key ordering, while errors in non-null links disconnect the structure [9]. A number of binary trees have been proposed that allow a limited number of errors to be detected and corrected, by performing a global examination of the erroneous storage structure instance [1, 6, 7, 8, 11, 14]. We do not permit such global examinations of the structure. Instead all components in the storage structure are visited in some order, and corrected if necessary as encountered, before being considered *trusted*. Each erroneous component is identified and corrected by examining previously trusted components and at most some constant number of potentially erroneous *untrusted* components. This bounded set of untrusted components forms a *locality*, which is assumed to contain at most some constant number of errors. Informally, these are the constraints that are imposed on a local correction procedure. More precise characterisations of such procedures [2] are too complex to be attempted here.

2. Proposed structure

In a height balanced (AVL) binary tree, the heights of the left and right subtrees below any node differ by at most one [5]. An identifier exists in each node which indicates the current direction of any such imbalance in the two subtrees below this node. Because the tree is height balanced, expected retrieval times are reduced, and worst case insert and delete operations remain logarithmic.

The AVL tree structure being considered will be made more robust by adding additional redundancy to the nodes of the structure. In addition to the height balancing information present in each node identifier, each node identifier will also contain two flags explicitly identifying the location of null links within this node. Each node will also contain an *arc* pointer, that ensures that the tree is 2-connected. If desired, keys may also be protected by associating checksums with them.

Nodes will be labelled N and distinguished by subscripts. Left links will be labelled l, right links r, and arcs a. Arbitrary links will be labelled c. Identifiers will be labelled id. Components will be prefixed by the node in which they reside, or by extension a path that addresses them. The symbol $\varnothing$ will be used to denote null pointers. Null pointers contain some value that indicates that they address no node.

Let the header node be labelled N_H. Then within a correct structure all pointers in the header address the root node if this exists. Links address child nodes as expected, and arc pointers form a cyclic single linked list which links nodes in the order defined by the following node traversal:

Visit(N_H)
If $N_H \cdot r \neq \varnothing$ {
Visit($N_H \cdot r$)
Traverse($N_H \cdot r$)
}
	Traverse(N) {
	If $N \cdot l \neq \varnothing$ Visit($N \cdot l$)
	If $N \cdot r \neq \varnothing$ Visit($N \cdot r$)
	If $N \cdot l \neq \varnothing$ Traverse($N \cdot l$)
	If $N \cdot r \neq \varnothing$ Traverse($N \cdot r$)
	}

Fig 1. Arc traversal order in the proposed tree

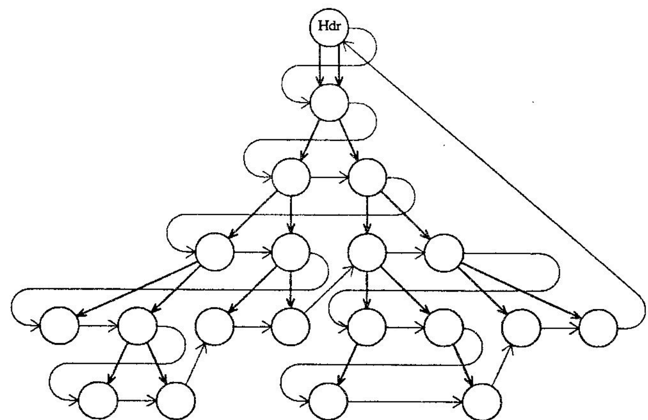

Fig 2. An example of the proposed AVL tree

0731-3071/87/0000/0085$01.00 © 1987 IEEE

3. Global characteristics

The structure described above has the following global characteristics. It can be traversed using either the links or the arcs and is thus 2-connected. It can be reconstructed by either using correct links, or by using correct arcs and identifiers, even if identifiers do not contain height balance information. The structure is therefore 2-determined [10]. We will show that the structure can be corrected when at most one error occurs in every bounded correction locality even if height balance flags are absent. The structure is therefore 1-locally-correctable, and thus trivially both 1-local-detectable and 1-correctable. However, without these flags certain pairs of changes within subtrees are undetectable, as shown in figure 3, since they leave the structure appearing internally to be correct. Thus, if height balance flags are absent, this structure is unusual since it has exactly the same detectability, local detectability, correctability, and local correctability.

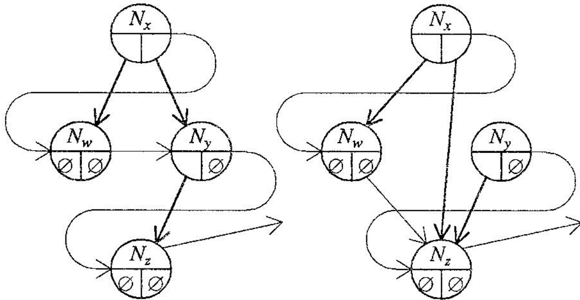

Fig 3. A pair of undetectable changes in $N_w \cdot a$ and $N_x \cdot r$

Given that node identifiers do contain height balance flags, the structure is 2-detectable since any undetectable transformation of the instance requires at least three changes, and certain sets of three changes are indeed undetectable as shown in figure 4.

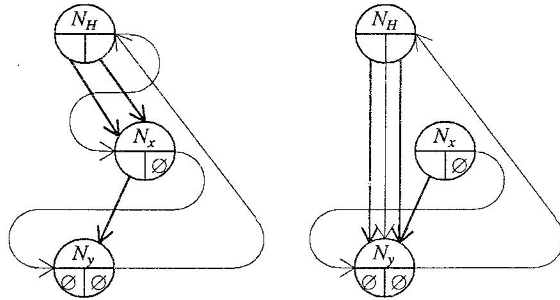

Fig 4. Three undetectable changes in $N_H \cdot l$, $N_H \cdot r$ and $N_H \cdot a$.

4. Local correction

The task of performing local correction can be divided into two cooperating sub-tasks. Firstly, a local detection procedure must exist, which when presented with the headers of a structure to be corrected, traverses the components of this instance in some deterministic order, by using values in the components already visited, and detects any single erroneous component after examining at most some bounded number of additional correct components. Having determined that an error exists in some bounded set of untrusted components called a locality, a correction procedure must identify the erroneous component and its correct value by using only components already seen by the detection procedure. Then having corrected this error, detection and correction can be repeated, until no more errors exist in the structure being corrected.

Since a correct storage structure can be traversed in the same sequence by using either arcs or links, any single error in a pointer will cause the resulting traversals to differ. Thus the local detection procedure can detect single errors in pointers if it can perform both traversals in parallel, while examining at most some bounded number of new components during each step of this parallel traversal.

The only possible traversal using arcs is to follow the single linked list formed by these arcs. Each step of this traversal involves examining one new arc. Following the same traversal using links is considerably more complex, but still involves examining at most a bounded number of new components at each step, as justified below.

Suppose that we have arrived at some non-null link $N_x \cdot c$ and wish to identify the non-null link $N_m \cdot c = N_x \cdot c \cdot a$ so that we can proceed to the next step of the traversal. Then, as shown in figure 5, at most four new null links will be examined before it is determined that N_x has no children or grandchildren that might contain this link. The search for this link then continues by proceeding up the tree from N_x, until we arrive at a node N_y having N_x in its left subtree. Since the tree is balanced, the node addressed by $N_y \cdot r$ exists and therefore contains the next two links in the ordering. If this node is a leaf node then two further null links will be encountered, before repeating the ascent of the tree from N_y until we encounter some N_z having N_y in its left subtree. Because the tree is balanced, the node addressed by $N_z \cdot r$ exists and has at least one child. Thus in the worst case we will encounter a seventh null link $N_z \cdot r \cdot l$, before encountering the non-null link $N_z \cdot r \cdot r$. Conversely, if no further non-null link exists, then during one of the two ascents up the tree the header node will be encountered, signalling that the traversal is complete.

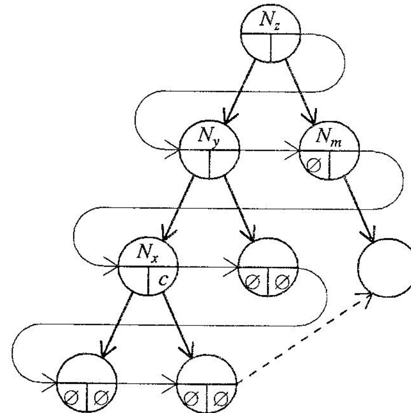

Fig 5. Maximum null links $\varnothing$ between $N_x \cdot c$ and $N_m \cdot c$

Having detected a discrepancy between arcs and links as a result of performing the above parallel traversal, any single error causing this discrepancy must occur in the last arc examined, or in the links examined during the last two steps of the parallel traversal, since either a null link encountered in the previous step of the traversal erroneously contained the same value as the desired link, or some error was encountered during the current step of the traversal.

Since we know that a single error exists in the above components we can assume that no error occurs in a bounded number of other new components that the correction procedure wishes to examine, and which the detection procedure has therefore been instructed to include in the locality, following the detection of an error. Identifying null links that contain erroneous values and non-null links that have erroneously become null is therefore trivial

since node identifiers contain flags indicating the location of null links within these nodes. Correcting such links is also trivial since null links correctly contain a known value, and non-null links correctly contain the same value as the last arc examined.

So assume that null links within the locality being corrected contain no errors, and that non-null links appear non-null even if erroneous. Then the error within this locality must occur either in the last arc examined, or in the non-null link that was expected to contain the same value as this last arc. Having determined the location of this error it can be trivially corrected since these two pointers agree when correct.

If the erroneous pointer addresses a non-existent node this can be detected when we attempt to access this node. Similarly if the erroneous pointer addresses a node outside of the instance being corrected, this can be detected by examining the identifier in this node.

So suppose that $N_x \cdot l$ and $N_w \cdot a$ address different nodes within the instance being corrected, as shown in figure 6, and let N_z be the node that both should address. Then one further traversal step using only correct links can be performed, arriving at the node addressed by $N_m \cdot c = N_z \cdot a$. This is because N_x either has a right child addressed by $N_z \cdot a$, or the node correctly addressed by $N_x \cdot l$ can be assumed to be a leaf since the tree is balanced. Having identified the correct value of $N_z \cdot a$, using only correct pointers, we can identify N_z and thus the erroneous pointer, since N_z is not N_w, and no other correct node within the structure contains an arc with the same value as N_z.

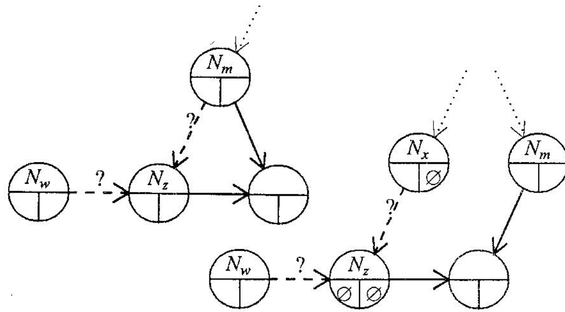

Fig 6. Possible configurations if left link or arc in error

Now suppose that $N_x \cdot r$ and $N_w \cdot a$ address different nodes within the instance being corrected. If $N_x \cdot l$ is null or addresses an internal node, then the next non-null link $N_m \cdot c$ within the link traversal does not depend on $N_x \cdot r$, and correction can be performed as described above.

A problem arises however if $N_x \cdot r \neq N_w \cdot a$ and $N_x \cdot l$ addresses a leaf node, since in this case determining the correct value of $N_m \cdot c$ involves determining the correct value of $N_x \cdot r$. This occurs for example if one of the changes depicted in figure 3 occurs. Fortunately, in this case $N_x \cdot r$ correctly addresses a subtree containing at most three nodes, since the tree is balanced. Thus if either $N_w \cdot a$ or $N_x \cdot r$ address a subtree containing more than three nodes, then the other pointer must be correct. Otherwise, the algorithm locates the next non-null link $N_n \cdot c$ not under N_x, by temporarily assuming that $N_x \cdot r$ is null. Now locate the node N_z visited last within each subtree, and reject the possibility that $N_x \cdot r$ correctly addresses this subtree if $N_z \cdot a \neq N_n \cdot c$. If neither subtree is rejected during this process, then both contain N_z. Since the two subtrees are distinct but each contain at most three nodes, one subtree must contain the single node N_z, while the other has as its root the parent of N_z. Thus both $N_w \cdot a$ and $N_x \cdot r$ correctly address this larger subtree.

5. Tidying up

Having corrected all pointers, identifiers can be corrected. Correction of the height balance flags in each node can be accomplished efficiently by using a post-order traversal, if the height of each left subtree visited is stacked, until the height of the corresponding right subtree has been established. Correcting the other information in each node identifier is trivial.

While performing this post-order traversal of the structure, keys can be locally corrected if desired. Add to each node a checksum component s which is the same size as the key components, and let N_{x+1} immediately follow N_x within a cyclic post-order traversal. Ensure during updates that $N_x \cdot s = N_x \cdot k + N_{x+1} \cdot k$, for all N_x. Thus if $N_x \cdot k$ is inserted, deleted or changed $N_x \cdot s$ and $N_{x-1} \cdot s$ must also be updated. N_x is updated anyway, and $N_{x-1} \cdot s$ can be updated by performing one additional probe. $N_{x+1} \cdot k$ need not be explicitly retrieved when updating $N_x \cdot k$ since prior to any change $N_{x+1} \cdot k = N_x \cdot s - N_x \cdot k$. Thus keys and checksums can be updated efficiently.

Since the key in the header is unused it can be assigned some constant value, and trivially corrected if in error, before being added to the set of trusted components. The correct values of successive keys are determined by recognising that $N_x \cdot k$, $N_{x-1} \cdot s - N_{x-1} \cdot k$ and $N_x \cdot s - N_{x+1} \cdot k$ should all agree on the value of $N_x \cdot k$, and that a single error in any component used by one of these expressions, leaves the other two expressions agreeing on the correct value for $N_x \cdot k$. Having used these votes to correct $N_x \cdot k$ if necessary, this component and $N_{x-1} \cdot k$ both have trusted values, and therefore $N_{x-1} \cdot s = N_{x-1} \cdot k + N_x \cdot k$ can be corrected if erroneous, before itself becoming trusted. Once $N_x \cdot k$ and $N_{x-1} \cdot s$ have become trusted, we iteratively correct $N_{x+1} \cdot k$ and $N_x \cdot s$, until no more untrusted components remain.

6. Conclusions

The above structure is of considerable interest, since no previously published binary search tree is locally correctable. It is surprisingly hard to develop binary trees that are easily used, efficient, and robust. It is even harder to construct such binary trees which are locally correctable.

The correction algorithm presented in this paper can be applied to unconstrained binary trees, if it is modified so that it makes no assumptions about the size of correction localities. Obviously the resulting algorithm will not perform local correction, but will continue to behave well on most tree structures. However, as instances degenerate into structures resembling linked lists with duplicated forward pointers, the algorithm degenerates into one that only performs 1-correction.

A variety of other structures have been developed that are locally correctable [3, 4], and a locally correctable B-tree has been previously presented [12]. We have implemented a very much simpler unconstrained locally correctable binary tree that includes two checksum components in each node. By using a generalised perfect Hamming code, these two checksums allow any single error in the five components consisting of key, links, and checksums to be corrected. The resulting structure is very robust when exposed to certain types of error, but when two or more errors occur in a single node, erroneous corrections will always occur. This is very unfortunate since software errors in keys will almost certainly propagate to associated checksums. An efficient binary search tree has also been developed that uses one redundant pointer, and occasionally keys, to perform local correction. This structure is more efficient than the structure presented here, and much easier to update, but requires that keys be both ordered, and themselves locally correctable. Many other designs for locally correctable binary trees have been considered and subsequently rejected, either because they were unappealing, or because they contained subtle flaws.

It is our intention to implement the proposed storage structure, so that its efficiency, robustness and crash resilience [13] can be further studied, and compared to the other locally correctable structures described above. It would also be of interest to analyse the behaviour of the local correction algorithm presented in this paper, when operating on binary trees having less restrictive height balance constraints.

7. Acknowledgements

I would very much like to thank my supervisor, Dr. D. J. Taylor, for his enthusiastic support of this research, and specifically for the interest that he showed in this paper. Without his many helpful comments it would have taken much longer to complete. I am also greatly indebted to my wife, A. R. Crowe, for her emotional support, and for her willingness to continue financing my research.

References

1. J. P. Black, D. J. Taylor, and D. E. Morgan, A compendium of robust data structures, *Digest of Papers: 11th Annual Int. Symp. on Fault-Tolerant Computing*, pp. 129-131 (24-26 June 1981).
2. 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). Submitted to *IEEE Transactions on Software Engineering*
3. I. J. Davis and D. J. Taylor, Local correction of mod(k) lists, CS-85-55, Dept. of Computer Science, University of Waterloo (December 1985).
4. I. J. Davis, Local correction of helix(k) lists, CS-86-30, Dept. of Computer Science, University of Waterloo (August 1986).
5. D. E. Knuth, *The Art of Computer Programming, vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass. (1973).
6. 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).
7. M. C. Sampaio and J. P. Sauve, Robust trees, *Proc. 15th Int. Symp. on Fault Tolerant Computing*, pp. 23-28 (June 19-21, 1985).
8. S. C. Seth and R. Muralidhar, Analysis and design of robust data structures, *Proc. 15th Int. Symp. on Fault Tolerant Computing*, pp. 14-19 (June 19-21, 1985).
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 and J. P. Black, Principles of data structure error correction, *IEEE Transactions on Computers* **C-31**(7) pp. 602-608 (July 1982).
11. D. J. Taylor and J. P. Black, Guidelines for storage structure error correction, *Proc. 15th Int. Symp. on Fault-Tolerant Computing*, pp. 20-22 (June 19-21, 1985).
12. D. J. Taylor and J. P. Black, A locally correctable B-tree implementation, *Computer Journal* **29**(3) pp. 269-276 (June 1986).
13. D. J. Taylor and C. J. Seger, Robust storage structures for crash recovery, *IEEE Transactions on Computers* **C-35**(4) pp. 288-295 (April 1986).
14. K. Yoshihara, Y. Koga, and T. Ishihara, A robust data structure scheme with checking loops, *Digest of Papers: 13th Annual Int. Symp. on Fault Tolerant Computing*, pp. 241-248 (June 28-30, 1983).