[15] S. Anderson, "The Looping algorithm extended to base 2^t rearrangeable switching networks," *IEEE Trans. Commun.*, vol. COM-25, pp. 1057-1063, Oct. 1977.
[16] J. Lenfant and S. Tahé, "Permuting data with the Omega network," *RADC Final Rep.*, Nov. 1978.
[17] H. J. Siegel and S. D. Smith, "Study of multistage SIMD interconnection networks," in *Proc. 5th Annu. Symp. Comput. Architecture*, Apr. 1978, pp. 223-229.
[18] T. Feng, C. Wu, and D. Agrawal, "A microprocessor-controlled asynchronous circuits switching network," in *Proc. 6th Annu. Symp. Comput. Architecture*, Apr. 1979, pp. 202-215.
[19] K. E. Batcher, "The multi-dimensional-access memory in STARAN," in *Proc. 1975 Sagamore Comput. Conf.*, p. 167; also in *IEEE Trans. Comput.*, vol. C-26, pp. 174-177, Feb. 1977.

Chuan-lin Wu (M'80), for a photograph and biography, see p. 702 of the August 1980 issue of this TRANSACTIONS.

Tse-yun Feng (S'61-M'67-SM'75-F'80), for a photograph and biography, see p. 702 of the August 1980 issue of this TRANSACTIONS.

Concurrent Search and Insertion in AVL Trees

CARLA SCHLATTER ELLIS, MEMBER, IEEE

***Abstract*—This paper addresses the problem of concurrent access to dynamically balanced binary search trees. Specifically, two solutions for concurrent search and insertion in AVL trees are developed. The first solution is relatively simple and is intended to allow several readers to share nodes with a writer process. The second solution uses the first as a starting point and introduces additional concurrency among writers by applying various parallelization techniques. Simulation results used to evaluate the parallel performance of these algorithms with regard to the amount of concurrency achieved and the parallel overhead incurred are summarized.**

***Index Terms*—Concurrent access, data bases, parallel processing, performance evaluation, search trees.**

INTRODUCTION

DYNAMICALLY balanced binary search trees are valuable data structures for implementing symbol tables and directories. This paper deals with the problem of concurrent access to trees built by one of the most widely studied of the balancing techniques, namely, AVL trees. It has been shown [1] that the AVL tree construction is the most efficient method of balancing binary search trees when operations are limited to insertion and searching.

It is not difficult to imagine an application in which concurrent insertion and retrieval of items in a table maintained as an AVL tree would be desirable. For example, a compiler designed to operate in a parallel processing environment might be organized such that several processes require access to the symbol table. In an earlier paper [2] we considered this problem of parallel compilation and found that sharing the symbol table among the proposed parallel processes presented a major conflict. Therefore, investigating the possibility for concurrency in the manipulation of these data structures is important.

In this paper we present algorithms for concurrent search and insertion in AVL trees. This work is related to similar studies with B-trees [3], [4], [8]-[10] and uses the same basic approach of placing locks on nodes of the tree. Another recent paper [7] discusses concurrent manipulation of binary search trees. Our presentation begins by defining our notation in terms of the AVL insertion algorithm for a sequential environment. Next, we outline the parallelization techniques applied in the design of two solutions for concurrent search and insertion. Finally, simulation results on the performance of these parallel algorithms are summarized.

Manuscript received August 25, 1979; revised April 20, 1980. This work was supported in part by the National Science Foundation under Grant MCS 76-09839.

The author was with the Department of Computer and Information Science, University of Oregon, Eugene, OR 97403. She is now with the Department of Computer Science, University of Rochester, Rochester, NY 14627.

DEFINITIONS

We assume that the reader is familiar with the terminology and operations associated with binary search trees. An AVL tree is defined to be a binary search tree such that for any node n in the tree,

$$|\text{height}\ (T_l(n)) - \text{height}\ (T_r(n))| \leq 1$$
(where $T_l(n)$ and $T_r(n)$ denote the left and right subtrees of n).

Detailed algorithms for manipulating AVL trees can be found in [6]. However, we will briefly describe the insertion algorithm in order to establish the terminology and because an

0018-9340/80/0900-0811$00.75

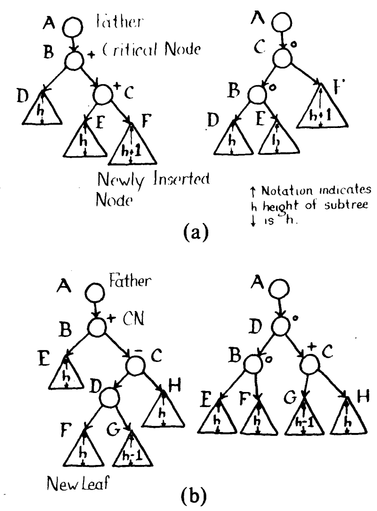


Fig. 1. Rotations in AVL tree. (a) Single rotation. (b) Double rotation.

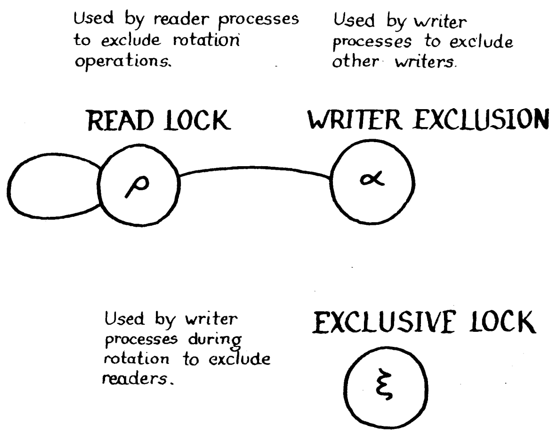


Fig. 2. Compatibility graph for locks.

understanding of the sequential algorithm is necessary to understand the concurrent algorithms. Each node n consists of four fields: *leftson* and *riteson,* pointers to the roots of $T_l(n)$ and $T_r(n)$, respectively, or to NIL if the subtree is empty, a data field called *key,* and *bf,* which indicates whether the height of the right son is greater than ($bf = +1$), equal to ($bf = 0$), or less than ($bf = -1$) the height of the left son. The balanced property of AVL trees is maintained by two transformations on the tree called single and double rotations. The situations which trigger each type of rotation and the modifications made to the tree are illustrated in Fig. 1.

The algorithm for insertion of a leaf with $key = k$ in an AVL tree is as follows.

1) Search the tree to find the appropriate place of insertion and keep a pointer to the last node on the path of insertion with nonzero *bf* (the root if no such node exists). This is the critical node *cn.* Insert the new leaf.

2) Adjust the *bf* fields of nodes on the insertion path between the *cn* and the new leaf. For each such node n, if the path to the place of insertion is to its left, $k < key(n)$, $bf(n)$ is changed to -1; otherwise, $bf(n)$ becomes $+1$.

3) Rotate if necessary. If $bf(cn) = 0$, the tree has become higher in the direction of insertion and *bf* must be modified appropriately. If $bf(cn)$ indicates that its higher subtree is in the opposite direction from the direction of insertion (e.g., $bf(cn) = +1$ and $k < key(cn)$), *bf* is changed to 0. If $bf(cn)$ and the direction of insertion coincide (e.g., $bf(cn) = -1$ and $k < key(cn)$), a rotation occurs according to Fig. 1.

CONCURRENCY IN AVL TREES

We now consider two solutions that allow a number of processes to operate concurrently on an AVL tree. Both solutions use various locks on the nodes of the tree to selectively exclude other processes.

Locking Solution

In the first solution the goal is to allow concurrency between a number of readers and a writer doing an insertion. The approach is straightforward, namely, during its search phase, a writer will lock other writers out of those nodes which may be involved in a rebalancing operation. Readers will be locked out of the fewest nodes possible and only during a rotation. Thus, readers can share nodes with a writer while it is searching for the place of insertion and the critical node, adjusting *bf* fields along the insertion path, and determining if a rotation is necessary. The solution uses three types of locks: ρ-locks for readers, α-locks for excluding other writers along the path from the father of the critical node to the place of insertion, and ξ-locks to exclude readers from nodes modified during a rotation.

Fig. 2 shows the compatibility relations these locks satisfy. An edge between any two nodes in this graph means that two different processes may simultaneously hold these locks on the same node of the tree. Thus, a node may be ρ-locked by several readers while it is α-locked by one writer. However, if a writer holds a ξ-lock on a node, no other process can hold any other locks on it. A single rotation requires ξ-locks on the father of the critical node and the critical node. A double rotation requires an additional ξ-lock on the son of the *cn* which lies on the insertion path. Fig. 3 gives an example of concurrent single rotation and read operations. The ρ-locks belonging to a reader are identified by using that reader's search key as a subscript.

The algorithms for the reader and writer are given as follows.

READER

```
RHO-LOCK pointer to root;
current ← pointer to root;
son ← root;
while son ~ = NIL and k~ = key[son] do
begin
    RHO-LOCK son;
    release RHO-LOCK on current;
    current ← son;
    /* determine appropriate son */
    if k < key[current] then son ← leftson[cur-
    rent]
    else son ← riteson[current]
end;
release RHO-LOCK on current;
if son = NIL then fail
else succeed
```

	WRITER INSERTING k=32	READER 1 SEARCHING FOR k=26	READER 2 SEARCHING FOR k=14
	insert	.	.
	adjust balance factors	.	.
		RHO-LOCK H	.
		release RHO-LOCK on D	
			RHO-LOCK A
(a)	XI-LOCK B		
	XI-LOCK D		
			compare 14 with key[A]
			appropriate son is B
			wait to RHO-LOCK B
	riteson[D] ← J		.
(b)	leftson[H] ← D		.
		determine appropriate	.
		son to be D	.
		wait to RHO-LOCK D	.
	riteson[B] ← H	.	.
	adjust bf	.	.
	release XI-LOCKs	.	.
		.	RHO-LOCK B
		.	release RHO-LOCK on A
		RHO-LOCK D	.
		release RHO-LOCK on H	.
(c)	release ALPHA-LOCKs	.	.
	terminate	.	.

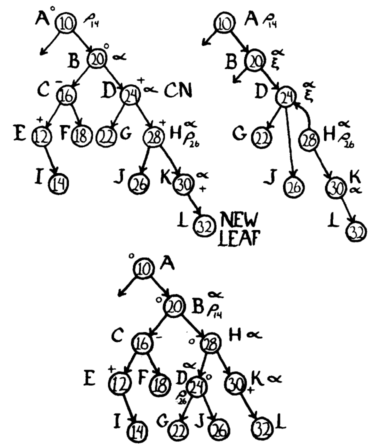


Fig. 3. Concurrent single rotation and reading.

```
WRITER
    ALPHA-LOCK pointer to root;
    current ← pointer to root;
    father of cn ← current;
    son ← root;
    cn ← root;
    /* search—resulting in path from father of cn
    to place of insertion remaining ALPHA-LOCKed
    */
    while son ~ = NIL and k~ = key[son] do
    begin
        ALPHA-LOCK son;
        if bf[son]~ = 0 then begin
            /* change cn pointer*/
            father of cn ← current;
            cn ← son;
            release ALPHA-LOCKs on ancestors of
            current
        end;
        current ← son;
        determine appropriate son
    end;
    if son = NIL then insert new node with key = k as
    appropriate son of current
    else release all ALPHA-LOCKs held by this process
    and terminate
    /* adjust balance fields between cn and new node
    as in sequential insertion */
    if k < key[cn] then begin
        direction ← −1;
        r ← current ← leftson[cn]
    end
    else begin
        direction ← +1;
        r ← current ← riteson[cn]
    end;
    retrace path from current to new node changing bf
    appropriately;
    /* rotate if necessary */
    case on bf[cn]
        0: bf[cn] ← direction;
        -direction: if bf[r] = direction then
         begin
            XI-LOCK father of cn;
            XI-LOCK cn;
            do single rotation;
            release all XI-LOCKs held by this process
         end
         else begin
            XI-LOCK father of cn;
            XI-LOCK cn;
            XI-LOCK r;
            do double rotation;
            release all XI-LOCKs held by this process
         end
    esac
    release all ALPHA-LOCKs held by this process
```

In this algorithm, a writer α-locks its path during the search phase so that the nodes along this path from the father of the *cn* to the place of insertion remain locked for the insertion, rebalancing, and rotation operations. This has the effect of locking subsequent writers out of the entire subtree rooted at the father of the *cn* rather than just those nodes on the first writer's path.

Claiming Solution

The second solution uses the first as a starting point and introduces additional concurrency through the use of various parallelizing techniques. The goal in this next algorithm is to increase concurrency among writers by allowing writers whose restructuring paths (i.e., the path between the *father of cn* and the newly inserted node) are disjoint to operate concurrently. The solution discriminates against writers that share the same path. Hopefully, the keys to be inserted by the parallel writer processes will tend to be spread throughout the tree rather than clustered.

The strategy used is summarized as follows. The writer process will search for the place of insertion using ρ-locks and will place an exclusive claim on the probable *father of cn.* (The *cn* pointer is set to the last node on the insertion path with nonzero *bf* whose father is not already claimed by another writer. As we shall see, this may not be the true critical node if another writer shares the insertion path.) This node is claimed rather than locked so that another writer may read past it and place a claim within this subtree if another potential *cn* is found. The new node with *key* = *k* will be inserted, while other inserting writers are excluded from the place of insertion. In its rebalancing phase, the writer excludes other rebalancing and rotating writers from nodes on the path from the *father of cn* to the first new node encountered on the insertion path (note that this new node was not necessarily inserted by this writer) and balance fields between *cn* and the new node will be adjusted. It is during the restructuring phase that writers which share the same path are penalized: one writer will claim the father of the lowest node with nonzero *bf.* Other writers will claim nodes higher in the tree and may find a lower *cn* during their rebalancing operations. Then the *cn* pointer must be moved down and the *bf* fields readjusted. Thus, writers along a shared path may do useless work that will need to be undone. Finally, rotation will take place if necessary with ξ-locks protecting the nodes involved.

This solution requires a modification in the data structure of the previous algorithm. In addition to the *key, leftson, riteson,* and *bf* fields, each node will contain one field, the *guardian* field, which will indicate which process is responsible for the rebalancing and rotation operations associated with the insertion of this node. If those operations have already been taken care of, this field indicates that this is an "old" node. In practice, this could be implemented with a single bit *guardian* field to signify "old" or "new" and an associative table pairing new nodes with their guardian processes. Or since only three codes are used in the two bit *bf* field and all new nodes have *bf* = 0, the remaining code could be utilized to indicate a new node, thus eliminating the *guardian* field altogether. This algorithm also uses a slightly different locking scheme. We will still need ρ-locks for reading and ξ-locks for exclusion of other processes during rotation. ι-locks will be used to enforce mutual exclusion among writers trying to insert a new node at the same place. The most significant change lies in replacing the α-lock of previous algorithms with an α'-lock and a *mark* bit that explicitly implement a lock to enforce mutual exclusion among rebalancing and rotating writers. The special feature of this implemented lock is that in addition to the operations of requesting the lock (which implies that the process is to wait if the request cannot be granted) and releasing the lock, a process will be able to test the *mark* bit to determine whether or not a node is locked without waiting for a lock request to be granted. This is necessary so that a writer in its search phase may place a virtual α-lock on its claim to stop rebalancing writers from proceeding down its path without being blocked itself by requesting to lock another writer's claimed node. The new compatibility relation is shown in Fig. 4. The virtual α-lock implemented by the *mark* bit is represented by the dotted portion of the compatibility graph.

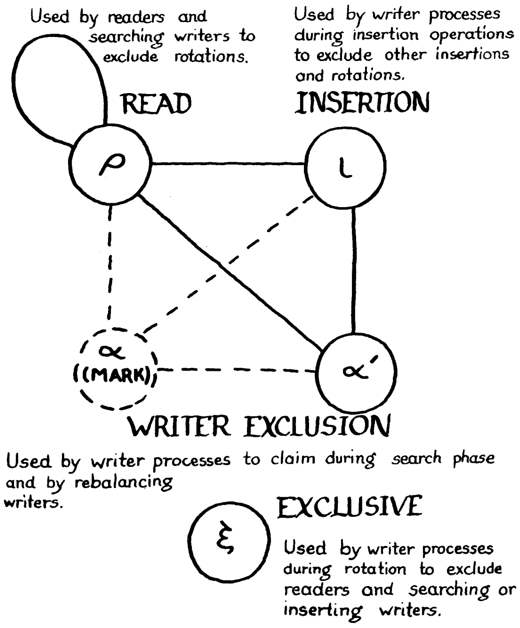


Fig. 4. Compatibility graph for locks in claiming solution.

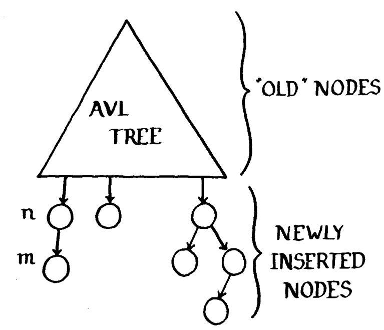


Fig. 5. Modified AVL tree.

There are a few key ideas that promote concurrency in this solution. The first important technique is to allow a temporary degradation of the tree structure. Since a writer can search and insert with a nominal amount of interference from other processes, it is possible that the tree could become quite unbalanced (i.e., no longer satisfying the AVL definition) after a number of processes have inserted but not yet restructured (cf. Fig. 5). The second technique is a relaxation of a process's responsibility to do its own work. Let n and m be two newly inserted nodes such that n is an ancestor of m. The restructuring operations associated with n should be done before those associated with m, but it is possible that the writer which inserted m, process 2, performs its rebalancing phase before process 1. In this solution, the two processes will essentially trade responsibilities with process 2 rebalancing for n and process 1 rebalancing for m. Because of the top-down nature of the restructuring pass, this trading must be explicitly done. A message will be sent to process 1 telling it to search for m's *key* during its restructuring pass. The final technique is made possible by the new locking scheme. The new lock has a different effect on searching writers than on rebalancing writers, thus essentially delaying the blocking of one process by another.

Readers in this solution are identical to readers in the

Locking solution. The writer algorithm is given below. We first present the procedures called by the main program followed by the main program itself.

```
procedure TRY TO CLAIM;
    begin
        ALPHA'-LOCK current;
        if bf[son] ~ = 0 and current is unmarked then
        begin
            mark current;
            cn ← son;
            if claim~ = pointer to root then unmark
            claim;
            claim ← current
        end
        release ALPHA'-LOCK on current
    end
procedure TRY TO INSERT;
    begin
        IOTA-LOCK current;
        determine appropriate son of current again;
        if son = NIL then
          insert newnode with key = k;
        release IOTA-LOCK on current
    end
procedure WAIT TO MARK(node);
/* essentially ALPHA-LOCKing */
    begin
        ALPHA'-LOCK node;
        while node is marked do
        begin
            release ALPHA'-LOCK on node;
            while node is marked do;
        ALPHA'-LOCK node
        end;
        mark node;
        release ALPHA'-LOCK on node
    end
WRITER
RHO-LOCK pointer to root;
claim ← current ← pointer to root;
son ← cn ← root;
/* search and claim potential father of cn */
while son ~ = NIL and k~ = key[son] do
begin
    RHO-LOCK son;
    if bf[son]~ = 0 and current is unmarked and current~
    = claim
    then TRY TO CLAIM;
    release RHO-LOCK on current;
    current ← son;
    determine appropriate son;
    if son = NIL then TRY TO INSERT
end;
if son~ = NIL then begin
/* k = key[son] so no insertion necessary */
    if claim~ = pointer to root then unmark claim;
    release RHO-LOCK on current;
    terminate
end;
release RHO-LOCK on current;
/* rebalance—mark path from father of cn to place of inser-
tion
and adjust balance factor fields */
if claim = pointer to root then WAIT TO MARK (claim);
WAIT TO MARK (cn);
current ← cn;
while guardian[current] is "old" do
begin
    if k < key[current] then
    begin
        r ← son ← leftson[current];
        direction ← −1
    end
    else begin
        r ← son ← riteson[current];
        direction ← +1
    end;
WAIT TO MARK (son);
while bf[son] = 0 and guardian[son] = "old" do
begin
    current ← son;
    if k < key[current] then
    begin
        bf[current] ← −1;
        son ← leftson[current]
    end
    else begin
        bf[current] ← +1;
        son ← riteson[current]
    end;
    WAIT TO MARK (son)
end
if bf[son]~ = 0 then
/* another process has been rebalancing on the path
already */
begin
    cn ← son;
    claim ← current;
    /* unmark from old claim to node above new claim
    and restore bf fields to zero from son of old cn to
    new claim */
    current ← old cn;
    unmark old claim;
    while current~ = claim do
    begin
        determine appropriate son;
        bf[son] ← 0;
        unmark current;
        current ← son
    end
    current ← cn
end
else begin
/* son hasn't been rebalanced for yet */
    if son is not this process's new node then
    begin
        /* trade new nodes with process now responsible
```

```
        for son node */
        send guardian[son] message to reset its k to
        key[newnode]
        and its newnode pointer to newnode;
        guardian[newnode] ← guardian[son]
    end;
    current ← son
  end
end
/* rotate if necessary */
case on bf[cn]
    0: bf[cn] ← direction;
    -direction: bf[cn] ← 0;
    direction: if bf[r] = direction then begin
        XI-LOCK claim;
        XI-LOCK cn;
        XI-LOCK r;
        do single rotation;
        release all XI-LOCKs held by this process
    end
    else begin
        XI-LOCK claim;
        XI-LOCK cn;
        XI-LOCK r;
        XI-LOCK appropriate son of r;
        do double rotation;
        release all XI-LOCKs held by this process
    end
esac
guardian[current] ← "old";
unmark all nodes marked by this process
```

Informal correctness proofs for these solutions can be found in [5].

EVALUATION

The primary goal in the design of these parallel algorithms was to increase the amount of concurrency possible between readers and writers and among numerous writers themselves. In the attempt to increase parallelism, a certain amount of parallel overhead was incurred (e.g., locking and unlocking of nodes, extra fields per node). Therefore, concurrency and parallel overhead will be the two most important factors to be considered in the evaluation of our algorithms. Results of simulation experiments will be summarized here. For a detailed discussion of the simulation study and a more complete presentation of the results see [5]. Very briefly, the approach is to simulate a fixed number of reader and writer processes executing steps of these algorithms scheduled according to randomly generated execution times which reflect fluctuations due to factors such as memory interference and differences between physical processors.

Among the measures of interest are the average number of concurrently busy processes during an interval of time (where busy means that the process is not waiting to be able to lock a node and is not finished with its operation), the improvement ratio (i.e., time for sequential steps/elapsed time for parallel execution), average path length and longest path (indications of how the tree degrades), and measures of the degree of locking in the tree and the amount of work done in placing and releasing locks.

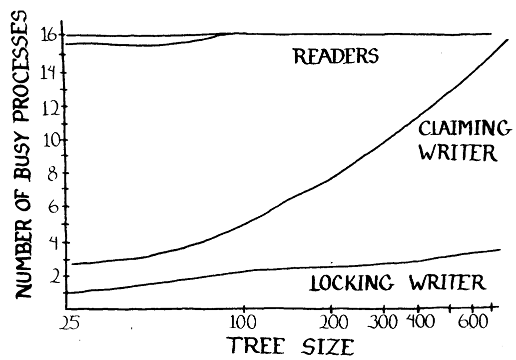


Fig. 6. Concurrency among readers and writers.

Fig. 6 deals with the amount of concurrency achieved by each solution as the tree grows. The results displayed are based on experiments with 16 readers and 16 writers active in the system. As could be expected, the first solution allows much less concurrency among writers than does the second algorithm. Also not surprisingly, there is a considerable amount of parallel overhead involved in executing these algorithms. One approach to evaluating this overhead requires the processor utilization and the improvement ratio to yield a value x which indicates that the cumulative time to execute all steps of each busy process is about x times the execution time spent doing work that would correspond to steps of a sequential execution. This value is 2.7 for our first solution and 2.5 for the second.

The degradation in the tree structure for the second algorithm is not significant for a randomly chosen set of keys.

The average number of locks placed and released per insertion is used to estimate the overhead cost of locking in the following way.

L = (cost of placing α-lock $\times$ average number of α-locks placed)
+ (average number of ξ-locks placed)
+ (average number of ρ-locks placed).

Let I be the average insertion path. Then we have for the first solution, $L = (3 \times I) + 1.1 + 0$, and for the second solution, $L = (3 \times 6.4) + 1.5 + I$.

Finally, the maximum number of locks which a writer would be expected to hold at some time during its insertion is a measure of potential concurrency. Since there is very little difference between these two solutions with respect to this measure, the concurrency among writers executing the second algorithm can be explained by the delay in locking rather than fewer locks held.

With regard to storage overhead, we compare the requirements of these concurrent solutions with the data structure of the sequential solution. One notable difference is the space which must be devoted to the various locks. In addition, the second algorithm calls for an associative table pairing active writer processes with their newly inserted nodes.

Conclusion

In this paper we have presented algorithms for concurrently searching and inserting in AVL trees. The solutions illustrate parallelizing techniques such as relaxing a process's responsibility to do its own work, allowing limited degradation of the structure, and delaying locking. These techniques should prove to be useful for introducing concurrency in other problems. The measurements presented indicate a reasonable increase in the amount of concurrency achieved by applying these techniques. In spite of the overhead, parallel execution yields a speedup.

Acknowledgment

The author wishes to thank J.-L. Baer for many helpful discussions.

References

[1] J.-L. Baer and B. Schwab, "A comparison of tree-balancing algorithms," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 322–330, May 1977.

[2] J.-L. Baer and C. Ellis, "Model, design, and evaluation of a compiler for a parallel processing environment," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 394–405, Nov. 1977.

[3] R. Bayer and M. Schkolnick, "Concurrency of operations on *B*-trees," *Acta Inform.*, vol. 9, pp. 1–22, 1977.

[4] C. Ellis, "Concurrent search and insertion in 2–3 trees," Dep. Comput. Sci., Univ. Washington, Seattle, TR-78-05-01, 1978; *Acta Inform.*, to be published.

[5] ——, "Design and evaluation of algorithms for parallel processing," Dep. Comput. Sci., Univ. Washington, Seattle, TR-79-07-01, 1979.

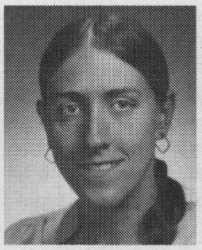
[6] D. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Reading, MA: Addison-Wesley, 1973.

[7] H. T. Kung and P. Lehman, "A concurrent data base manipulation problem: Binary search trees," presented at the 4th Int. Conf. Very Large Data Bases, Sept. 1978; *Ass. Comput. Mach. Trans. Database Syst.*, to be published.

[8] Y. S. Kwong and D. Wood, "Concurrency in *B*-trees, *S*-trees and *T*-trees," Dep. Comput. Sci., McMaster Univ., Hamilton, Ont., Canada, TR79-CS-17, Aug. 1979.

[9] P. Lehman and S. B. Yao, "Efficient locking for concurrent operations on *B*-trees," Preliminary Rep., May 1979.

[10] R. Miller and L. Snyder, "Multiple access to *B*-trees," in *Proc. Conf. Inform. Sci. Syst.*, Mar. 1978.



Carla Schlatter Ellis (M'79) was born in Toledo, OH, on August 20, 1950. She received the B.S. degree in mathematics and computer science from the University of Toledo, Toledo, OH, in 1972, and the M.S. and Ph.D. degrees in computer science from the University of Washington, Seattle, in 1977 and 1979, respectively.

In 1978 she joined the Department of Computer and Information Science, University of Oregon, Eugene, where she is currently an Assistant Professor. Her research interests include parallel processing and data structures.

Dr. Ellis is a member of the Association for Computing Machinery and the IEEE Computer Society.