Concurrent Manipulation of Expanded AVL Trees

Zhang Yin (章 寅) and Xu Zhuoqun (许卓群)

Department of Computer Science and Technology, Peking University, Beijing 100871, P.R. China

Received October 3, 1996; revised June 16, 1997.

Abstract

The concurrent manipulation of an expanded AVL tree (EAVL tree) is considered in this paper. The presented system can support any number of concurrent processes which perform searching, insertion and deletion on the tree. Simulation results indicate the high performance of the system. Elaborate techniques are used to achieve such a system unavailable based on any known algorithms. Methods developed in this paper may provide new insights into other problems in the area of concurrent search structure manipulation.

Keywords: AVL tree, data structure, binary search tree, concurrent algorithm, concurrency control, locking protocol.

1 Introduction

Expanded AVL tree (EAVL tree) is an implementation of the famous AVL tree[1], which, as an efficient search structure[2,3], finds applications to databases, operating systems, symbol tables in compilers, etc. The topic of this paper is the design of a system which can support any number of concurrent processes performing searching, insertion and deletion on an EAVL tree. Detailed algorithms are developed. Simulation results on the performance of the system are also summarized.

Substantial work has been done on developing concurrent algorithms for manipulation of search structures such as binary search trees[4–6], 2-3 trees[7], k-HB trees[8], AVL trees[9,10], B-tree and its variations[11–16], etc. However, none of these algorithms can be applied to achieve full concurrency of all three kinds of operations on an AVL tree, say searching, insertion and deletion. In this paper, we solve this problem completely on an implementation of AVL tree, namely, EAVL tree.

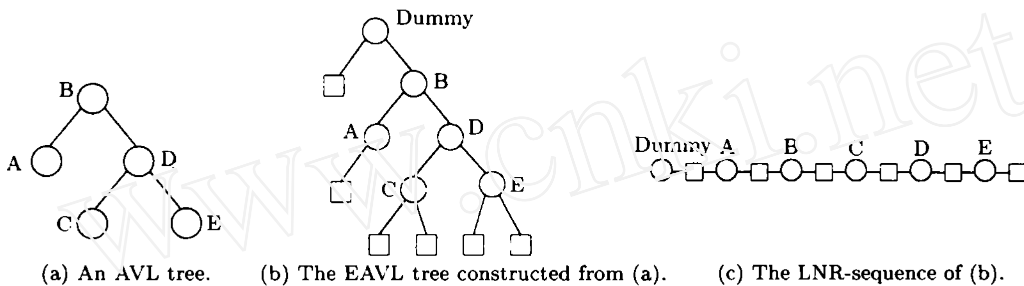
2 Terminology and Background

2.1 Definitions

We assume that the reader is familiar with the terminology associated with binary search trees. An AVL tree is a binary search tree in which the heights of the left and right subtrees of any node may differ by at most one. The balance factor of any node n in the tree is defined as height$(T_l(n))-$ height$(T_r(n))$, where $T_l(n)$ and $T_r(n)$ denote the left and right subtrees of n respectively. An expanded AVL tree (EAVL tree) is constructed from an initial AVL tree with k nodes by adding a dummy node and $(k+1)$ leaves into the tree. The root of the initial AVL tree becomes the right child of the dummy node in the resulting EAVL tree. The newly added leaves are called the L-nodes. The nodes in the initial AVL tree together

© 1994-2008 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

with the dummy node are called the I-nodes. The LNR-sequence of a binary search tree is a node sequence generated by an LNR (in order) traversal. It is evident that for an EAVL tree, (1) the LNR-sequence is headed by the dummy node; (2) the L-node and the I-node alternatively take position in the LNR-sequence. Fig.1 illustrates an initial AVL tree, the EAVL tree constructed from it, and the LNR-sequence of the resulting EAVL tree. L-nodes and I-nodes are denoted by rectangles and circles respectively in Fig.1. In an EAVL tree, each I-node has a key field according to which the I-nodes are ordered. The key is taken from a totally ordered set. The key of the dummy node has the value of $(-\infty)$. The L-nodes have no such fields. For any node n in an EAVL tree, the predecessor of n is defined as the I-node with the greatest key among all the I-nodes preceding n in the LNR-sequence (i.e. being nearer to the dummy node than n is in the LNR-sequence). If no such I-node exists, the predecessor of n is defined as NULL. Similarly, the successor of n is the I-node with the smallest key among all the I-nodes succeeding n in the LNR-sequence (i.e. being farther to the dummy node than n is in the LNR-sequence). If no such I-node exists, the successor of n is defined as NULL.



(a) An AVL tree. (b) The EAVL tree constructed from (a). (c) The LNR-sequence of (b).

Fig.1

2.2 Sequential Operations on EAVL Trees

The sequential algorithms for EAVL trees are quite similar to those for AVL trees. So we only give a brief introduction to them here. Interested readers can refer to [3] for detailed algorithms for AVL trees.

2.2.1 Searching

The search process traverses the tree downward (i.e. in the direction from the dummy node to the leaves) searching for a given key. If it finds an I-node with the given key, it succeeds; otherwise, it fails.

2.2.2 Deletion

The algorithm for a delete process consists of the following four steps:

1. Descending. In this step, the delete process behaves exactly the same as a search process. If it finds an I-node (denoted by *cur*) with the given key, it goes to Step 2; otherwise, it fails.

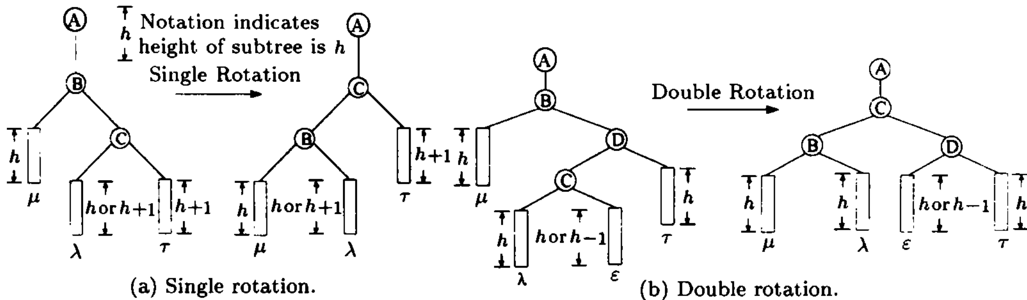
2. Find the appropriate I-node for physical deletion. *cur* is marked for physical deletion if and only if at least one of its children is an L-node. If this condition is not satisfied, it must be true that 1) the right child of *cur*'s predecessor is an L-node; 2) *cur*'s predecessor is below *cur* in the tree. The delete process makes a key substitution, i.e. substitutes the key of *cur* with the key of *cur*'s predecessor, and then makes *cur*'s predecessor the I-node for physical deletion.

3. Physical deletion. The I-node marked for physical deletion and one of its children (an L-node) are physically deleted from the tree.

4. Ascending. The delete process traverses the tree upward (i.e. in the direction from the leaves to the dummy node) and rebalances the tree by rotations if necessary. There are two kinds of rotations: the single rotation and the double rotation. Fig.2 illustrates the modification made

© 1994-2008 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

on the tree during rotations. The delete process may need to perform rotations more than once. (Note: Only part of the triggering conditions of rotations are illustrated in Fig.2. Interested readers can refer to [3] for detailed information about the triggering conditions.)



(a) Single rotation. (b) Double rotation.

Fig.2. Rotations.

2.2.3 Insertion

The algorithm for an insert process consists of the following three steps.

1. Descending. The insert process traverses the tree downward searching for a given key. If it finds an I-node with the given key, it fails; otherwise, it can find an L-node (denoted by *cur*) on which physical insertion is to be made.

2. Physical insertion. The given key is added into *cur*, thus *cur* is changed from an L-node into an I-node. Moreover, two L-nodes are generated and added into the tree as the two children of *cur*.

3. Ascending. The insert process traverses the tree upward and rebalances the tree by rotations if necessary. An insert process may perform rotation at most once. The triggering conditions of rotations performed by an insert process are slightly different from those of rotations performed by a delete process. Interested readers can refer to [3] for more information.

Comment: From the above sequential algorithms, we can see:

1) When a delete process makes the key substitution, temporary redundancy is introduced into the tree, i.e. two nodes with the same key value may coexist in the tree.

2) Before a delete process or an insert process finishes rebalancing the tree, the tree may be temporarily unbalanced, i.e. the heights of the left and right subtrees of some node may differ by more than one.

When processes are allowed to manipulate the tree concurrently, they may introduce temporary redundancy and/or unbalance into the tree simultaneously. However, as shown in Section 5, such degradation in the tree structure is not significant for a randomly chosen set of keys.

3 Data Structure

3.1 The Model of Computation

All the data of an EAVL tree are stored in a single shared global memory to which an unbounded number of concurrent processes have access. Each process can perform one of the three operations (i.e. searching, insertion and deletion) on the tree. No centralized control governs the action of processes—they operate asynchronously and independently. Simultaneous read and write on a common data field in the shared memory are allowed, but the hardware must ensure the correctness of the result of a read operation, i.e. ensure that no values changed halfway can be read out.

© 1994-2008 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

3.2 Representation of the EAVL Tree

```
TYPE STATETYPE=(EMPTY, TRYING, WAITING);
     RESERVETYPE=(FREE, RIGHT, LEFT, SUCC);
     NODETYPE=(INODE, LNODE);
     NPTR=↑NODE;
     NODE=RECORD
         father, pred, target: NPTR;
         γ-state, α-state: STATETYPE;
         invalid: BOOL;
         CASE ntype: NODETYPE OF
           LNODE: (     );
           INODE:(
             left, right, succ: NPTR;
             key: KEYTYPE;
             γ-marked, α-marked, deleted: BOOL;
             bf: INTEGER;
             reservation: RESERVETYPE;  )
         END;
VAR dummy: NPTR;
```

An EAVL tree is denoted by a dummy, the pointer to the dummy node. We do not differentiate between a node and the pointer to it hereafter. For any node n in the tree, $n.father$ denotes the father node of n and is used by the processes to traverse the tree upward; $n.pred$ is the predecessor of n and is introduced to improve the efficiency of key substitution; $n.ntype$ indicates whether n is an I-node (if $n.ntype$ is INODE) or an L-*node* (if $n.ntype$ is LNODE); $n.left$, $n.right$ and $n.succ$ stand for n's left child, right child and successor respectively; $n.bf$ is the balance factor of n; $n.key$ stores the key of n. The other fields will be explained later.

3.3 Locks

We use the same basic approach of placing locks on nodes of the tree as in [9].

Altogether five kinds of locks are used in our system: α, β, γ, ξ and ρ. Fig.3 shows the incompatibility relations these locks must satisfy. Two kinds of locks (unnecessary to be different) are specified to be incompatible with each other (i.e. they cannot be placed on a node simultaneously) if and only if an edge exists between the two nodes in Fig.3. For any lock L, $L \in \{\alpha,\beta,\gamma,\xi,\rho\}$, the procedures to place and to release a lock L on node n are L-lock(n) and L-unlock(n) respectively. Placing a lock on a node does not prevent other processes from accessing any data field of the node. However, those processes who want to place a lock incompatible with a lock existing on the node will get blocked. The locking procedures can be implemented with semaphores and PV operations[17].

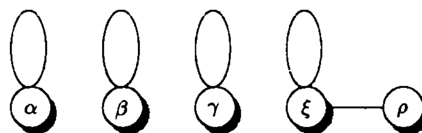


Fig.3. Incompatibility graph for locks.

4 Concurrent Operations on EAVL Trees

Based on the sequential algorithms described in Section 2, we develop the concurrent algorithms for operations on an EAVL tree in this section.

4.1 Search Process

All the descenders in our system including the search process use the same algorithm. Here, by descender, we refer to any process (either a search process, a delete process, or an insert process) traversing the tree downward. Each descender starts by ρ-locking the dummy node and making it the current node. It keeps descending from the current node to one of its children depending on the value of the key being searched. To move from the current node to the next one, the descender ρ-locks the destination node, makes it the new current node and then ρ-unlocks the previous current node. In this manner, the presence

© 1994-2008 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

of the descender is always indicated by the lock ρ on at least one node. Since the lock ρ is compatible with itself, multiple descenders can coexist on a common node.

If the search process finds an I-node with the given key, it ρ-unlocks the current node (with the given key) and succeeds; otherwise, it ρ-unlocks the current node and fails.

4.2 Delete Process

The algorithm for the delete process consists of five steps: 1) descending, 2) exclusion among multiple deletions, 3) some preparations, 4) selecting the appropriate I-node for physical deletion and 5) ascending. We give discussions to the concurrency issues involved in them step by step.

4.2.1 Descending

In this step, the delete process behaves exactly the same as a descender. It traverses the tree downward searching for an I-node with the given key. Its presence is indicated by the lock ρ on at least one node. If at the end of this step, an I-node (denoted by *cur*) with the given key is located, the delete process just proceeds to the next step; otherwise, it ρ-unlocks the current node and fails.

4.2.2 Exclusion among Multiple Deletions

Multiple delete processes may have the same key to delete. However, each node can be physically deleted at most once. Thus exclusion is necessary among multiple deletions on a common node. The lock β, which is incompatible with itself, is introduced for this purpose. Moreover, a field *n.deleted* is incorporated for each node n.

The delete process β-locks *cur* (the I-node found at the end of Step 1) and examines the value of *cur.deleted*. If the value is TRUE (i.e. someone else also intends to delete *cur.key* and comes earlier), the delete process releases all locks it holds and fails; otherwise, it sets *cur.deleted* to TRUE (thus preventing other delete processes from deleting *cur.key*), releases all locks it holds and proceeds to the next step. Here the lock β is used to enforce mutual exclusion on the critical region *cur.deleted*.

4.2.3 Some Preparations

In this step, two problems are to be solved:

1) As shown in Section 2, if neither child of *cur* is an L-node, the delete process needs to make a key substitution and thus making *cur.pred* the I-node for physical deletion. Therefore, it is possible that a delete process D_1 wants to delete $cur_1.key$ but needs to physically delete the node $cur_1.pred$, while another delete process, say D_2, wants to delete $cur_2.key$ and needs to physically delete the node cur_2, where cur_2 equals $cur_1.pred$. In this case, mutual exclusion is necessary between D_1 and D_2. However, since $cur_1.key \neq cur_2.key$, such exclusion cannot be achieved by Step 2.

2) In order to keep the consistency of the *pred* and *succ* fields, we must prevent different processes (delete processes and insert processes) from modifying a common *pred* or *succ* field simultaneously.

Both problems are solved by the incorporation of the field γ-marked into each node. The value of γ-marked indicates whether a node has been γ-marked by any process. Any delete process (or insert process) is required to γ-mark some I-nodes before it can make a physical deletion (or insertion). Specifically, for a delete process, supposing that *cur* is the I-node with the key it wants to delete, the process must γ-mark *cur*, *cur.pred*, *cur.pred.pred* before making a physical deletion. (Note: Among the three nodes, only *cur.pred.pred* may be nonexistent. This can be handled by simply eliminating any operation associated with *cur.pred.pred* from our concurrent algorithm. From now on, we assume that *cur.pred.pred*

© 1994-2008 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

does exist.) For an insert process, supposing that *cur* is the *L*-node on which the physical insertion is to be made, the process must γ-mark *cur.pred* before making the physical insertion. Fig.4 illustrates the nodes in the LNR-sequence which need to be γ-marked by a (n) delete (insert) process.

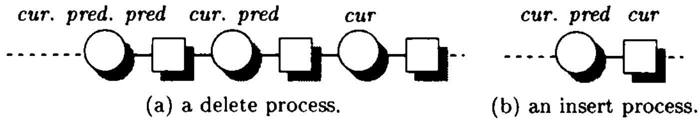


Fig.4. *I*-nodes to be γ-marked by (a) and (b).

γ-marking procedures are devised to implement the above strategy. An insert process calls ins-γ'-mark-pred(*cur*) to γ-mark *cur.pred*. A delete process calls γ-mark-node(*cur*) to γ-mark *cur* and then calls γ-mark-pred(*cur*) and γ-mark-pred(*cur.pred*) to γ-mark *cur.pred* and *cur.pred.pred*. Here we only describe the procedures involved in the delete process. The procedure ins-γ-mark-pred(*cur*) will be introduced later when we work on the insert process in Subsection 4.3.3.

• γ-mark-node(*cur*)

When a delete process D calls γ-mark-node(*cur*), another delete process, say D^*, may want to delete *cur* physically, where *cur* equals *cur*.pred* and *cur*.key* is the key that D^* wants to delete. In this case, D^* must have γ-marked *cur* and have substituted *cur**.key with the value of *cur.key*. Thus, there are two nodes with the key value of *cur.key* in the tree, say *cur* and *cur**. *Cur* will be physically deleted by D^*, thus D should try to delete *cur**. In order to provide a way of communication between D and D^*, two new fields, say *n.invalid* and *n.target*, are incorporated for each node n.

When D^* decides to physically delete *cur* (i.e.*cur*.pred*), it sets *cur.invalid* to be TRUE and let *cur.target* point to *cur**. Then D^* waits until *cur.deleted* (already been set to TRUE by D) is reset to FALSE.

When D calls γ-mark-node(*cur*), it keeps testing the value of *cur*.γ-marked. If *cur*.γ-marked is FALSE, D sets its value to TRUE and thus takes control of *cur*. Otherwise, D keeps testing the value of *cur.invalid*. Whenever *cur.invalid* is TRUE, D ρ-locks *cur.target*, informs D^* that it has known the new target *cur.target* (i.e. *cur**) by resetting *cur.deleted* to FALSE, and then makes *cur.target* as its new current node. After that, D just goes back to Step 2. The lock γ, which is incompatible with itself, is used here to enforce mutual exclusion on the critical region *cur*.γ-marked.

• γ-mark-pred(n)

When a delete process D tries to call γ-mark-pred(n), *n.pred* may get changed. If D can not always be informed of the latest value of *n.pred*, problems may occur. For example, the old value of *n.pred* may point to a physically deleted node, and if D tries to γ-mark a nonexistent node, the system will collapse. There must be some safety measurements against this. The field *n*.γ-*state* is introduced for this purpose. *n*.γ-*state* may have one of the following three values: EMPTY, TRYING and TESTING.

While D tries to γ-mark-pred(n), it keeps reading the value of the field *n.pred* and setting *n*.γ-*state* to TRYING until it is lucky enough to find that *n.pred*.γ-*marked* is FALSE. It then sets *n.pred*.γ-*marked* to TRUE and thus succeeds in γ-*marking n.pred*. Again the lock γ is used here to enforce mutual exclusion on the critical region *n.pred*.γ-*marked*.

After another process P (either a delete process or an insert process) changes the value of the field *n.pred*, if it finds that the value of *n*.γ-*state* is TRYING (i.e. someone else is TRYING to γ-mark *n.pred*), it calls a procedure γ-safety(n) to ensure the safety of the system. When calling γ-safety(n), P starts by setting *n*.γ-*state* to TESTING, then repeatedly tests *n*.γ-*state* until *n*.γ-*state* becomes TRYING again. Since D sets *n*.γ-*state*

© 1994-2008 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

to TRYING each time after it reads the latest value of the field *n.pred*, the transition of *n.γ-state* from TESTING to TRYING ensures that D has known the latest value of the field *n.pred*. Now P can do its own work safely.

4.2.4 Selecting the Appropriate I-Node for Physical Delection

The field α-marked is used to enforce mutual exclusion among ascenders (i.e. any processes traversing the tree upwards). The value of α-marked indicates whether a node has been α-marked by any process. The lock α is used to enforce mutual exclusion on the critical region α-marked. The related α-marking procedures are α-mark-node(n), α-mark-father(n). These procedures are quite similar to the γ-marking procedures in that the caller of the procedure just keeps testing the value of the α-marked field until it can α-mark the node. The lock α is used for mutual exclusion on the critical region α-marked. However, the procedure α-mark-father(n) needs some elaborate revisions as we will see later. Another procedure α-safety(n) is used to ensure the safety of the system. α-safety(n) is the same as γ-safety(n) except that any fields associated with γ now become the corresponding fields associated with α.

In the beginning of this step, the delete process has already γ-marked *cur*, *cur.pred* and *cur.pred.pred* (abbreviated as c, p and pp respectively). The object of the delete process is to α-mark the I-node for physical deletion and be ready to ascend the tree at the end of this step.

The delete process starts by α-mark-node(c). If either child of c is an L-node (i.e. c is the appropriate I-node to be physically deleted), the delete process proceeds to Step 5 directly.

Otherwise, the delete process needs to make a key substitution, i.e. to substitute *c.key* with the value of *p.key* and then make p the I-node for physical deletion.

When updating *c.key*, no descenders should have access to *c.key*. The lock ξ is introduced for this purpose. The delete process is required to *ξ-lock c* before updating the value of *c.key*. Since the lock ξ is incompatible with the lock ρ, no descenders can ρ-lock c and have access to *c.key* if c has been ξ-locked.

After updating *c.key*, the delete process cannot immediately physically delete p from the tree. The reason is that there may be some descenders on the path from c down to p searching for the key with the value *p.key* or trying to insert a key with the value belonging to the interval (*p.key*, key_1), where key_1 denotes the original value of *c.key*. These processes need to be driven off the path. To achieve this, the delete process just traverses the path from c down to p like a descender but applies the lock ξ instead of the lock ρ to drive other descenders off the path. It is not difficult to prove that when this has finished, all these processes either have found the key it searches for or have reached an appropriate child of p.

Having finished traversing the path from c to p, the delete process makes reservation for itself on p. For each node n, the field *n.reservation* is used for the *reservation*. *n.reservation* may have one of the four possible values: FREE, LEFT, RIGHT and SUCC. Here, FREE means that n has not been reserved by any process; RIGHT, LEFT and SUCC indicate that n has been reserved by a process present on *n.left*, *n.right* and *n.succ* respectively. Here the delete process just sets *p.reservation* to SUCC. Then it α-unmarks c and tries to α-mark-node(p).

As we can see later, ascenders usually α-mark nodes upward the tree. Here, p is below c in the tree. Therefore, in order to avoid deadlocks, the delete process is required to α-unmark c before it can α-mark p. However, this may lead to a problem: during the period between the moments that the delete process α-unmarks c and that it manages to α-mark p, the node p may get involved in some rotations and become the ancestor of c. Moreover,

© 1994-2008 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

both children of p may become I-nodes. In this case, c, instead of p, should be the node for physical deletion. This problem is solved by a combination of the mechanism of reservation, the algorithm for rotations and the procedure α-mark-father(n). Firstly, if a node is reserved as SUCC, it cannot be α-marked through α-mark-father(n) (ensured by the procedure α-mark-father(n)) and thus can only be α-marked by its reserver. This ensures that p may get involved in at most one rotation. Thus, if p becomes the ancestor of c during the (only) rotation it is involved in, it can only become the father of c. (This is evident according to the modification to the tree in a rotation.) The algorithm for rotations is devised to ensure that if p, a node reserved as SUCC, gets involved in a rotation, and after the rotation, neither child of p is an L-node, then c, which is the right child of p after the rotation, is α-marked after the rotation. Thus, when the delete process finally α-marks p, if p is no longer fit for physical deletion, the delete process only needs to α-unmark p and then make c as its current node for physical deletion. Since c has already been α-marked during the rotation, the delete process can now safely proceed to Step 5.

Note: We only describe part of the mechanism of reservation here. More will appear in Subsection 4.2.5.

4.2.5 Ascending

In this step, the delete process keeps ascending from the current node to its father node and makes rotations when necessary. The physical deletion is made after the first step upward the tree. Suppose that a delete process D wants to move one step upward from the current node (denoted by c). Let f denote the father node of c, g denote the grandfather of c (i.e. the father of f), b denote the brother of c (i.e. the other child of f) and n denote the nephew of c (i.e. an appropriate child of b that together with c, f and b, may get involved in a double rotation). The possible positions of c, g, f, b and n are illustrated in Fig.5. D takes the following steps:

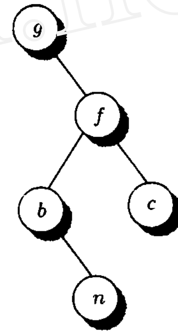


Fig.5. Nodes that may get involved in a rotation.

1) α-mark f.

2) If no rotation is necessary, then α-release(c). Here, by α-release(c) we mean that, if no node has ever been physically deleted by this process, c is physically deleted; otherwise, c is merely α-unmarked.

3) If a single rotation is needed, D α-marks b, α-release(c), α-marks g, and then makes the rotation.

4) If a double rotation is needed, D α-marks b, α-marks n, α-release(c), α-marks g in sequence and then makes the rotation.

From the above steps, we can see that D may need to α-mark a child, say *son*, of a node already α-marked by it, say *nd*. However, there may be another process P (either a delete process or an insert process) which has α-marked *son* but wants to α-mark *nd*. If both D and P persist in trying to α-mark their target nodes, a deadlock occurs. To break the deadlock, we require that D gives its control of *nd* to P. This is achieved by the mechanism of reservation. D first tests the value of *son.α-state*. If the value is TRYING, D knows that the process which has α-marked *son* is trying to α-mark *nd*. In this case, D reserves *nd* for P by setting *nd.reservation* to LEFT or RIGHT depending on whether *son* is the left or the right child of *nd* and then α-unmarks *nd*. The procedure α-mark-$father(n)$ is also revised so that a process can α-mark-$father(n)$ if and only if n.father has been neither α-marked nor reserved for any other process. Moreover, we allow a process to reserve a node as SUCC even if it has already been reserved as LEFT or RIGHT (i.e. RIGHT or LEFT may be overridden by SUCC).

© 1994-2008 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

Next we need to have some words on physical deletion. When a delete process D tries to delete an I-node v and one of its children ln (an L-node), it keeps descenders from v by placing the lock ξ on both v and its father node $v.father$. When modifying fields of the nodes, D calls the procedures α-safety and ξ-safety when necessary. Moreover, before it can safely free the space occupied by v and ln, D must do the following two things. Firstly, as mentioned in Subsection 4.2.3, D must drive those delete processes trying to γ-mark-node(v) off by setting the value of *v.invalid* and *v.target*. Secondly, D should drive those insert processes trying to ins-γ-mark-pred(ln) off ln by setting *ln.invalid* to be TRUE and letting *ln.target* point to some appropriate L-node. For details of ins-γ-mark-pred(ln), see Subsection 4.3.3.

The last thing we need to discuss is the rotations. When making a rotation, the delete process keeps descenders from nodes in modification by placing ξ locks on nodes involved in the rotation. Specifically, as shown in Fig.5, in case of a single rotation, the process ξ-locks g, f and b in sequence; in case of a double rotation, the process ξ-locks g, f, b and n in sequence. The procedures α-safety and ξ-safety are also called to ensure the safety of the system when fields of the nodes are modified. Moreover, as mentioned before, the process making the rotation needs to pay attention to nodes reserved as SUCC.

4.3 Insert Process

Now we come to the algorithm for the insert process which is quite similar to the algorithms presented in [8]. Compared with the algorithm for the delete process, this part is much simpler. The algorithm for the insert process also consists of five steps: 1) descending, 2) exclusion among multiple insertions, 3) γ-marking, 4) physical insertion and 5) ascending. We discuss them one by one.

4.3.1 Descending

In this step, the insert process behaves like a search process. If it finds a node with the given key, it fails; otherwise, it reaches an L-node and makes it the current node (denoted by *cur*).

4.3.2 Exclusion among Multiple Insertions

Multiple insert processes may want to make physical insertion on *cur*. In this case, only one process may make the physical insertion. The others must wait until the physical insertion has completed and then should go on descending. Specifically, the insert process starts by trying to β-lock *cur*. After it succeeds in β-locking *cur*, it examines *cur* to see if it is still fit for physical insertion. If someone else has already made an insertion on *cur* and thus making *cur* no longer fit for physical insertion, the insert process just goes on descending from *cur*. Otherwise, it proceeds to Step 3.

4.3.3 γ-marking

The insert process tries to γ-mark *cur.pred* in this step. Two problems are to be solved: 1) *cur.pred* may get changed; 2) *cur* itself.*invalid* becomes TRUE, the insert process just moves to a new node pointed to by *cur.target* and tries to make an insertion there. Meanwhile, like in the procedure γ-mark-pred, it keeps testing the value of *cur.pred* and setting *cur*.γ-state to TRYING. Such actions help to ensure that the insert process can always be informed of the latest value of *cur.pred*.

4.3.4 Physical Insertion

The insert process makes the physical insertion on *cur*. By ξ-locking *cur.father*, it excludes descenders from accessing *cur* during the physical insertion. The procedures α-safety and γ-safety are called when necessary. At the end of the physical insertion, the

© 1994-2008 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

insert process wakes up all the insert processes blocked on *cur* by the lock β and drives them down to the children of *cur*. To make sure that no other insert process is present on *cur* (indicated by the lock ρ on *cur*), the insert process first ξ-locks and then ξ-unlocks *cur*.

4.3.5 Ascending

The ascending step of the insert process is simpler than that of the delete process and no reservation is necessary here. The insert process traverses the tree towards its dummy node and α-marks each new node encountered in its path. It α-unmarks the nodes in such a manner that it always α-marks the last three nodes. This ensures a strict serialization among ascending processes. When rotation is necessary (at most once for each insert process), the process just makes the rotation. The algorithm for rotations has been described in Subsection 4.2.5.

Now, we have completed the design of the whole system. The pseudo-code for the system can be obtained from the authors.

5 System Evaluation

5.1 Correctness of the System

The correctness of the system consists of four aspects:

1) Conditional termination. If the number of the concurrent processes is finite, then all the processes will terminate in a finite period of time. We do not ensure absolute termination because the tree structure may be continuously changing, forcing a slow search process to search forever.

2) Although temporary degradation (i.e. unbalance and/or redundancy) in the tree structure is allowed, after all the processes have terminated, and the resulting tree should strictly satisfy the definition of an EAVL tree.

3) Each search process returns the right node at the time of termination. Each insert or delete process can correctly modify the tree structure as it intends to.

Due to the limitation of space, we are not allowed to give formal proof of the correctness of our system here. Instead, we list the following two facts which are useful in the proof:

Fact 1. Rotations have no influence on the LNR-sequence of the tree.

Fact 2. Each descender ρ-locks the nodes downward the tree. Each ascender usually α-marks the nodes upward the tree. Each insert/delete process γ-marks the nodes in the direction from the rear to the head of the LNR-sequence. Such locking orders help to deny the circular wait condition of deadlocks[18] and thus help to ensure a deadlock-free system.

5.2 Simulation Results on the Performance of the System

When evaluating the performance of the system, we mainly consider the degree of concurrency and the degree of the degradation in the tree structure.

In our simulation experiment, all the keys are generated using a uniform random number generator function. The locking procedures are implemented by semaphores and PV operations[17]. In a time unit, a process can only run a single step. The operation that can be performed in a single step can be a request for PV operation, a read/write on a shared data field or just some kind of local calculation. To simplify the simulation, multiple read/write on a common data field is allowed in a time unit. The PV operations are atomic and in a time unit, at most one request for PV operation can be granted on a common semaphore. In any time unit, the order in which all the active processes run is randomly generated with a uniform random number generator function.

In the experiment, an initial EAVL tree with k I-nodes is manipulated concurrently by n search processes, n delete processes and n insert processes. Each process on termination

© 1994-2008 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

starts a process performing an operation with the same type. The system status is sampled at frequent intervals and the overall performance is evaluated over a long enough period of time. Specifically, the system runs for $k = 25, 50, 100, 200, 400, 800$ and $n = 1, 2, 4, 8, 16, 32$. For each (k, n), the system runs continually for 10,000 time units and the system status is sampled every 10 time units. The items for statistics are as follows:

1) S, the speedup of the system. S is defined as SS/T, where SS is the sum of the steps that all the processes run concurrently and T is the total time units the system runs. T equals 10,000 in our experiment.

2) AH, the average height of the tree and MH, the maximum height of the tree.

3) AB, the average balance factor of the tree and MB, the maximum balance factor of the tree.

4) AR, the average redundancy of the tree and MR, the maximum redundancy of the tree. The redundancy of the tree at time t is defined as $(I(t) - D(t))$, where $I(t)$ is the number of the I-nodes in the tree at time t and $D(t)$ denotes the number of the different keys in the tree at time t.

Among these items, S indicates the degree of concurrency and the others show the degree of the degradation in the tree structure. Detailed simulation results can be found in Appendix. From these results, we can see that our system enjoys a high degree of concurrency at the expense of insignificant degradation in the tree structure.

6 Conclusion

In this paper, we have presented a fully concurrent EAVL tree system. To the authors' knowledge, the system is not available based on any known algorithms. Elaborate techniques used in this paper, such as the mechanism of reservation, the α-marking scheme etc., may prove effective in solving other problems. Future efforts can be extended to other search structures such as the k-HB trees[8] etc.

References

[1] Adel'son-Vel'skii G M, Landis E M. An algorithm for the organization of information. *Soviet Math. Doklady,* 1962, 3: 1259–1263.

[2] Baer J L, Schwab B. A comparison of tree-balancing algorithms. *Comm. ACM,* 1977, 20: 322–330.

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

[4] Kung H T, Lehman P L. Concurrent manipulation of binary search trees. *ACM Trans. Database Systems,* 1980, 5: 354–382.

[5] Manber U. Concurrent maintenance of binary search trees. *IEEE Trans. Software Eng.,* 1984, 10: 777–784.

[6] Manber U, Ladner R E. Concurrency control in a dynamic search structure. *ACM Trans. Database Systems,* 1984, 9: 439–455.

[7] Ellis C S. Concurrent search and insertion in 2-3 trees. *Acta Inform.,* 1980, 14: 63–86.

[8] Varshneya A, Madan B B, Balakrishnan M. Concurrent search and insertion in K-dimensional height balanced trees. In *Proc. 8th International Parallel Processing Symposium,* Mexico, 1994, pp.883–887.

[9] Ellis C S. Concurrent search and insertion in AVL trees. *IEEE Trans. Computers,* 1980, 29: 811–817.

[10] Medidi M, Deo N. Parallel dictionaries on AVL trees. In *Proc. 8th International Parallel Processing Symposium,* Mexico, 1994, pp.878–882.

[11] Kwong Y S, Wood D. A new method for concurrency in B-trees. *IEEE Trans. Software Eng.,* 1982, 8: 211–222.

[12] Lehman P L, Yao S B. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Systems,* 1981, 6: 650–670.

[13] Bayer R, Schokolnick M. Concurrency of operations on B-trees. *Acta Inform.,* 1977, 9: 1–22.

© 1994-2008 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

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

[15] Samadi B. B-trees in a system with multiple users. *Inf. Process. Lett.*, Oct. 1976, 5: 107–112.

[16] Miller R, Snyder L. Multiple access to B-trees. In *Proc. Conf. Information and Systems,* Johns Hopkins Univ., Baltimore, Md., March 1978.

[17] Dijkstra E W. Cooperating Sequential Processes. Mathematics Dept., Technological University, Eindhoven, The Netherlands, 1965. (Reprinted in Genuys F (ed.), Programming Languages, Academic Press, New York, 1968.)

[18] Havender J W. Avoiding deadlocks in multitasking systems. *IBM Systems Journal,* 1968, 7(2): 74–84.

Zhang Yin received his B.S. degree in Computer Science from Peking University in 1997. He is currently a Ph.D. Candidate in Department of Computer Science, Cornell University. His research interests include parallel processing, distributed systems, computer networks and algorithm design.

Xu Zhuoqun received his B.S. degree in Mathematics from Peking University in 1957. He is currently a Professor in Department of Computer Science and Technology, Peking University. His research interests include parallel computation, GIS and AI.

Appendix. Simulation Results

Table 1. Results for $n = 1$

Item \ k	25	50	100	200	400	800
S	2.93	2.96	2.985	2.989	2.986	2.993
MH	5	7	8	9	10	12
AH	4.86	6.01	7.08	8.10	9.09	11.02
MR	1	1	1	1	1	1
AR	0.18	0.11	0.11	0.06	0.12	0.16
MB	2	3	3	2	2	2
AB	0.32	0.33	0.32	0.31	0.29	0.33

Table 2. Results for $n = 2$

Item \ k	25	50	100	200	400	800
S	5.70	5.81	5.88	5.93	5.95	5.95
MH	6	7	8	9	10	12
AH	5.18	6.15	7.14	8.15	9.04	11.04
MR	2	2	2	2	2	2
AR	0.37	0.35	0.24	0.30	0.22	0.28
MB	3	2	3	3	2	2
AB	0.40	0.38	0.34	0.36	0.30	0.33

Table 3. Results for $n = 4$

Item \ k	25	50	100	200	400	800
S	10.80	10.95	11.40	11.65	11.71	11.79
MH	7	8	8	9	10	12
AH	5.34	6.46	7.30	8.30	9.32	11.07
MR	3	3	3	3	3	3
AR	0.70	0.67	0.60	0.46	0.59	0.61
MB	4	3	3	2	2	2
AB	0.45	0.41	0.35	0.35	0.32	0.34

Table 4. Results for $n = 8$

Item \ k	25	50	100	200	400	800
S	15.15	21.08	21.63	22.13	22.64	22.83
MH	5	7	8	9	11	12
AH	3.42	6.37	7.54	8.66	10.00	11.06
MR	2	6	4	5	5	5
AR	0.17	1.38	1.26	1.21	1.10	1.29
MB	3	3	3	3	4	3
AB	0.49	0.41	0.39	0.37	0.36	0.34

Table 5. Results for $n = 16$

Item \ k	25	50	100	200	400	800
S	32.88	36.68	37.41	39.05	38.38	38.97
MH	6	8	9	10	10	12
AH	4.74	6.62	7.76	8.92	9.68	11.06
MR	3	6	5	6	6	6
AR	0.54	1.26	1.36	1.86	2.15	2.27
MB	3	4	3	3	3	3
AB	0.42	0.51	0.40	0.40	0.35	0.34

Table 6. Results for $n = 32$

Item \ k	25	50	100	200	400	800
S	51.62	66.31	66.63	53.22	62.26	46.56
MH	6	9	9	9	11	12
AH	5.93	8.70	7.95	8.73	10.24	10.79
MR	2	4	3	9	5	8
AR	0.52	1.04	0.27	1.69	1.03	2.65
MB	2	4	4	3	4	3
AB	0.65	0.84	0.46	0.38	0.37	0.35

© 1994-2008 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net