Concurrent Manipulation of Binary Search Trees

H. T. KUNG and PHILIP L. LEHMAN
Carnegie-Mellon University

The concurrent manipulation of a binary search tree is considered in this paper. The systems presented can support any number of concurrent processes which perform searching, insertion, deletion, and rotation (reorganization) on the tree, but allow any process to lock only a constant number of nodes at any time. Also, in the systems, searches are essentially never blocked. The concurrency control techniques introduced in the paper include the use of special nodes and pointers to redirect searches, and the use of copies of sections of the tree to introduce many changes simultaneously and therefore avoid unpredictable interleaving. Methods developed in this paper may provide new insights into other problems in the area of concurrent database manipulation.

Key Words and Phrases: databases, data structures, binary search trees, concurrent algorithms, concurrency controls, locking protocols, correctness, consistency
CR Categories: 3.73, 3.74, 4.32, 4.33, 4.34, 5.24

1. INTRODUCTION

As the construction of large multiprocessors (such as Cm* [22]) becomes practicable, much thought has been given to methods of exploiting these powerful computers. One natural and important application is the use of the multiprocessing power to manipulate large databases. Multiprocessors might be used to service the needs of several database users simultaneously, or to reduce the time necessary for a single complex task. In order to gain experience in this direction, we studied the use of multiprocessors in manipulating a simple data structure known as a binary search tree. As a result, we designed systems that could support any number of the concurrent operations of insertion, deletion, and reorganization (specifically, rebalancing) on the tree. Although the systems are designed for implementation on multiprocessors, they are also useful for implementation on uniprocessors that support multiprogramming. This paper presents these systems and discusses the ideas behind them.

This research was supported in part by the National Science Foundation under Grant MCS 78-236-76 and the Office of Naval Research under Contract N00014-76-C-0370.
Authors' address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

1.1 Some General Techniques Used

One problem often encountered by concurrent systems is the necessity of doing a set of operations simultaneously or indivisibly for correctness reasons. This occurs where any partial completion of the set may lead to a temporary inconsistency in the data structure. To solve this dilemma, we introduce the idea of "copies" of sections of the binary search tree. These copies are to be created outside the tree, modified as appropriate to reflect the result of the set of operations on the tree, and then introduced into the tree structure with a single, indivisible operation. This technique may be used to simultaneously replace all of the pieces of an old version of that section of the tree, effectively performing many modifications simultaneously.

In using the copying technique, a substantial amount of work is done before the results of that work are introduced into the data structure. Conversely, we also use the technique of "postponement": delaying any work that need not be done immediately. Each process only does "what it has to do." Other processes can perform the postponed work separately. With this technique, the multiprocessing capability supplied by multiprocessors is utilized. This is particularly advantageous in the case where work *cannot* be done immediately, and a process would have had to wait; instead, it can relegate the work to another process, to be done when feasible.

Another difficulty generally encountered in asynchronous concurrent processing is that the actions of one process may serve to invalidate some decisions made by another process. It may be the case that a process will see the tree "change out from under it." For this possibility, we provide a recovery mechanism for "confused" processes, in the form of "back pointers" that redirect processes whose position in the tree has been invalidated by the actions of other processes. These back pointers are attached to "blue nodes" which signal the process that it is lost in the first place.

In designing algorithms to use these techniques, we tried to keep the general design of the algorithms simple and efficient. For example, our locking scheme uses no reader locks; nor do we permit any process to exclusively lock a node. We only use writer-exclusion locks that prevent simultaneous update of a node by more than one process. The locking scheme itself is also quite simple. No complex queuing mechanism is required to administrate lock requests, on whose order the well-being of the system might depend. In addition, the number of nodes that any process can lock at one given time is bounded by a very small constant.

Utilizing the ideas mentioned above, we build a set of tree-mutating processes. In addition, we study garbage collection mechanisms that make available for reuse nodes that have been deleted from the tree. While garbage collection processes are not specifically tree mutators, they are necessary for the completeness and correct functioning of the systems. We illustrate two such mechanisms: a simple version with a single garbage collection process, and a version that allows concurrent garbage collection (many collectors) to operate at the same time as tree mutators. Here we note another illustration of the idea of postponement: It is generally unnecessary to collect garbage immediately after it is generated.

Developing these algorithms has strengthened our belief that concurrent al-

gorithms are for the most part far less intuitive than sequential algorithms. This is one reason that much attention has been given recently to the proof of the correctness of concurrent programs (following in the footsteps of the work on verification of sequential programs). We offer verifications of our systems, and include a sketch of the correctness proof for the concurrent garbage collector.

Substantial work has been done on developing concurrent algorithms for the manipulation of B-trees, which are a popular data storage structure, especially for large collections of data (see Appendix B). These algorithms have steadily improved, using as a measure the size of the B-tree region locked by a process. An adaptation of the results in the present paper allows yet another improvement to B-tree algorithms along these lines (see [16]). Further generalizations of the ideas presented here may suggest highly concurrent algorithms for manipulating other data structures.

1.2 Outline of the Paper

In Section 2 we define the concurrent manipulation problem studied in the paper, state our assumptions, and set up the definitions to be used in our correctness proofs. In Sections 3, 4, and 5 we develop our concurrent systems. In Section 6 we propose a simple garbage collection mechanism. A summary and concluding remarks are given in Section 7. In Appendix A we elaborate upon a concurrent garbage collection mechanism. In Appendix B we give some background for this problem area and describe related work that has been done. In Appendix C we offer a natural correctness criterion for concurrent search systems and argue that the properties we have proved for our systems together constitute a sufficient condition for that criterion.

2. THE PROBLEM, BASIC DEFINITIONS, AND ASSUMPTIONS

As mentioned above, the problem considered in this paper is the design of systems that can support concurrent manipulations on a binary search tree. (For a general discussion of binary search trees, see, e.g., [10].) We hope to achieve maximum concurrency without impairing the correctness of the systems. In what follows we first describe the data structure shared by all of the concurrent processes and then define the problem more precisely.

2.1 The Data Structure

The data structure consists of a directed graph and a queue, called GC-queue. The binary tree is embedded in the directed graph. Let the nodes of the graph be labeled by integers $1, \ldots, M$, and the node labled by n be in memory location n for all $n = 1, \ldots, M$. Node n (or simply n) is used to refer to either the node labeled by n or the pointer to that node, depending on the context. For the purpose of this paper, we assume that each node contains six fields: a left pointer field, a right pointer field, a back pointer field, a color field, a value field, and a lock field. A pointer field contains either a pointer to a node or the null pointer "λ." The value field contains a value from a linearly ordered set. The color field contains the color of the node, which is "white" or "blue"; nodes on the binary tree are always white and blue nodes are never on the tree. (The use of this

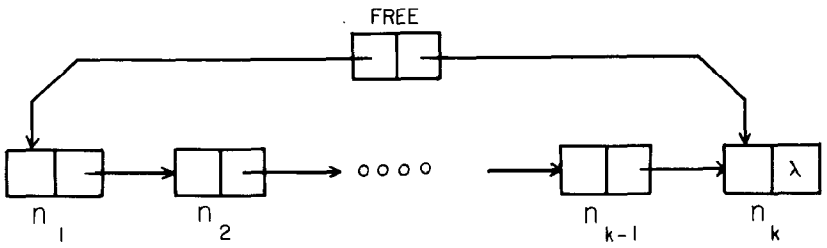

Fig. 1. The FREE node and free list.

notation was motivated by the availability of colored chalk.) The lock field of a node is set by a process in order to gain the right to modify that node. Only one process at a time may have any given node locked.

The pointer contained in the left, right, or back pointer field of node n is called the *left, right,* or *back pointer* of n and is denoted by n.left, n.right, or n.back, respectively. Similarly, the contents in the color, value, and lock fields of node n are denoted by n.color, n.value, and n.lock, respectively. The topology of the graph is determined by the pointers of the nodes in the graph. Let m and n be any two nodes. If m.left (respectively, m.right, m.back) = n, we say that n is the *left (right, back) son of* m and that m *points to* n *through the left (right, back) pointer of* m. There are two special nodes denoted by ROOT and FREE. Node ROOT corresponds to the root of the binary search tree. It is assumed that ROOT.value = "infinity," which is a value greater than any value one can search for. Node FREE points (through its left pointer) to the first node of a list, called the *free list* (cf. Figure 1), which is a sequence of any number of nodes n_1, n_2, . . . , n_k, satisfying the following properties:

F1. FREE.left = n_1, n_k.right = λ.
F2. For $1 \leq i < k$, n_i.right = n_{i+1}.
F3. For $1 \leq i \leq k$, n_i.color = white.
F4. FREE.right = n_k.

Garbage (blue) nodes are nodes that have been deleted from the tree. Garbage nodes are always inserted into the GC-queue. Through the garbage collection, nodes in the queue are appended to the free list and are then ready to be reused.

2.2 Concurrent Processes on the Tree

We wish to perform processes of the following types *concurrently* on the tree structure:

Insertion is the process of adding a value to the tree, if the value is not already in the tree.

Deletion is the process of removing an existing value from the tree.

Rotation is the process of "rotating" a (sub)tree so that the heights of its subtrees can be adjusted. Rotation is typically used for balancing a tree (see, for example, [10, p. 454]). In this paper rotation is also used for performing deletion (see Section 5).

Searching is the process of looking for a node with a given value v in the tree. If a node with value v exists, then the search is successful, otherwise it is unsuccessful. Searching does not modify the tree, and is often used by other processes. For example, if we wish to delete a value from the tree, then we must first search for that value in the tree, since if it is not present, we cannot delete it.

Garbage collection is the process of appending garbage nodes to the free list so that they can be reused.

For correctness reasons we allow a process to lock one or several nodes against modification by another process. But, for achieving a high degree of concurrency we require that the number of nodes locked by any process at any one time be bounded by a small constant. In addition, we try to delay searches as little as possible, since, in general, searching is done far more often than modifying. The operation of locking (respectively, unlocking) a node n is denoted by "lock(n)" ("unlock(n)").

2.3 Definitions for Correctness

We say that a concurrent system for manipulating a binary search tree is *correct* if the system possesses the following properties:

P1. *The tree is always consistent.* At any time, if we freeze the current tree, then an in order traversal (see, [10, p. 316]) of the tree generates the nodes with values in sorted order.

P2. *The termination position of a search is always consistent.* The *termination position* of a (successful or unsuccessful) search is defined to be the last node whose value is examined by the search before it is terminated. Consider a search for value v. Suppose that it terminates at node n at time t. We require that at the instant t if we freeze the tree and start a new search for the same value v from the root then n must be the termination position of the new search.

P3. *There is no deadlock.*

P4. *An intended update is always carried out.* An insertion, deletion, or rotation process will indeed insert, delete, or rotate as intended, before it terminates.

P5. *A value v can be added to or deleted from the tree only by the insertion or deletion process, respectively.* (These processes are defined later.) *In particular, only nodes which are no longer reachable by an existing or future search will be garbage collected, and all such nodes will be garbage collected.*

Property P1 is clearly needed for maintaining the binary search tree. The necessity of Properties P3, P4, and P5 is also obvious. For Property P2, we note that if it is not satisfied then two searches for the same value may conclude differently on the same tree. Property P2 is important to insertion and deletion processes, since searching is performed in those processes. In fact, in Appendix C we show that Properties P1–P5 are sufficient conditions for a natural correctness criterion for concurrent search systems.

2.4 Basic Assumptions

We shall prove correctness of a system based on the following assumptions:

A1. The tree is consistent initially, before any process has acted on it.
A2. The search, insertion, and rotation processes, which are defined later, are "correct" when executed alone, in the sense that, starting from a consistent tree, the search process will find a value if and only if the value is in the tree and the insertion and rotation processes preserve the consistency of the tree.
A3. Each process can read or write on individual fields of a node as an indivisible step.
A4. If process A attempts to lock a node which is already locked by process B, then A must wait for B to unlock the node. In this case, we say that process A is blocked (by process B) at the node.
A5. The procedures *create* and *append*, defined in Sections 3 and 6, for manipulating the free list are correct in the sense that they will preserve the properties of the free list (cf. F1–F4 in Section 2.1).

Note that to have processes satisfying A2 and A5 is quite standard. So for clarity in this paper we chose to assume A2 and A5 rather than to prove them.

2.5 Database Record Considerations

This paper is not primarily concerned with the problem of updating records associated with the keys in a database; rather, we focus on the problems of concurrent reorganization of the part of the database containing the key structure. Here we will digress briefly to suggest one possible method for associating records with the keys in the tree.

To each node, we would add an additional field (which is ignored in the remainder of this paper): the record field. This field contains a pointer to the record associated with the key stored in that node. A specific implementation may decide to put this record on the disk or in main memory. Regardless, the pointer in the node points to some large chunk of data that constitutes the associated record. For each individual record, we would view that record as a distinct database. To change information in this node, we might lock the whole record (as distinct from locking the node itself). Alternatively, since we view the record as a database, we could maintain its consistency as we would in a general database.

3. A SEARCH-INSERTION SYSTEM

In this section we describe a system which can support any number of concurrent searches and insertions on a binary search tree, and prove the correctness of this system. The procedures and correctness proof methods presented in this section will form the basis for constructing and proving more complex systems considered in later sections of the paper.

3.1 An Example

We want to demonstrate that a concurrent system consisting of the usual sequential searching and insertion processes without modifications is incorrect.

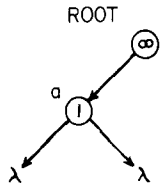

Fig. 2. A simple tree.

Consider Example 3.1 on a simple tree with ROOT.left = a and a.value = 1, as depicted in Figure 2. In the example, variables s and r are local to processes search(2) and insert(2), respectively.

Example 3.1

	search(2)	insert(2)
	⟨*previous steps—start at ROOT*⟩	
1.	$s \leftarrow a$	
2.		$r \leftarrow a$
3.	$2 > s$.value(= 1)	
4.	$s \leftarrow s$.right	
5.	s is λ	
6.		$2 > r$.value(= 1)
7.		$r \leftarrow r$.right
8.		r is λ
9.		Insert a node with value 2 as the right son of node a
10.	Value 2 does not exist!	

Note that at step 10 the search incorrectly concludes that the value 2 does not exist in the tree. Equivalently, Property P2 is not satisfied at the time the search terminates. The problem can be solved by introducing some locking scheme into the sequential processes. This modification is described below.

3.2 The System

The Search Process

Search. This procedure searches for a node in the tree with a given value v.

procedure search(v)
(f, dir)←find(ROOT, v);
$s \leftarrow f$.dir;
if $s \neq \lambda$ **then print** "Value v is at node s"
else print "Value v is not in the tree" fi;
unlock(f);

The procedure find(n, v) is defined below. It consists of the usual descent through a tree and is expressed recursively for clarity. It is readily seen that the termination position of search(v) (intuitively, the node for which we are looking) is f.dir if the search is successful and is f if the search is unsuccessful. The procedure *find* is an auxiliary procedure that is used by several processes considered in this paper.

Find. The following procedure searches for a node with value v, starting from node n, with n.value $\neq v$. (Recall that we assumed that ROOT.value = "infinity." Hence find(ROOT, v) is always well defined since ROOT.value $> v$ for all v.) The procedure returns a pair (f, dir), where f is a node and dir is a direction (left or

right). At the time the procedure returns (f, dir), node f is locked, and has the property that if value v exists in the tree, then f points through the pointer dir to a node whose value is v (i.e., [f.dir].value = v); otherwise f.dir = λ.

```
procedure find(n, v)
f←n;
if v < f.value then dir←left else dir←right fi;
s←f.dir;                                          /*Choose correct son*/
if s ≠ λ and s.value ≠ v then
     return find(s, v)                            /*Recurse*/
else
  lock(f);
  if s ≠ f.dir then                               /*It slipped away (see note below)*/
    unlock(f);
    return find(f, v)                             /*So recurse*/
  else return(f, dir) fi                          /*Found it*/
fi
```

Note that after the lock(f) operation the process *makes sure* that f is *still* the father of s (i.e., s = f.dir). This is necessary since another concurrent insertion process might have changed f.dir and unlocked f between the time that *find* decided that "$s = \lambda$ or s.value = v" and the time that *find* succeeded in doing lock(f). In this case, *find* must resume the search at node f again.

The Insertion Process

Insertion. This procedure inserts a node with value v into the tree (at one of the leaves), if no such value already exists in the tree.

```
procedure insert(v)
(f, dir)←find(ROOT, v);
if f.dir ≠ λ then
  print "Value v is already in the tree";
  unlock(f)
else
  create(w);                                      /*Build a node*/
  w.left←λ;
  w.right←λ;
  w.value←v;
  f.dir←w;                                        /*Point to it*/
  unlock(f)
fi
```

The procedure create(w), which is a standard free list manipulation procedure (with synchronization), is defined as follows:

Create. This procedure creates a new node named w, by removing it from the free list.

```
procedure create(w)
lock(FREE);
if FREE.left = FREE.right then
  Abort the process which calls create and inform
  the system that the free list is empty
else
  w←FREE.left;
  FREE.left←[FREE.left].right
fi
unlock(FREE);
```

At this point the reader is advised to convince himself that the locking scheme used in procedure *find* indeed solves the problem demonstrated by Example 3.1. It is also instructive to note that a search process is never blocked by other processes, except possibly at the time right before it terminates. This property holds for all the systems considered in the paper.

3.3 Property P2′—A Property for Proving P2

It is difficult to prove Property P2 defined in Section 2.3 by induction on time, since it is only meaningful as a property at the termination time of a search. Here we define another property, Property P2′, which implies Property P2 but is more convenient to use for the (inductive) correctness proofs in later sections of the paper.

Consider a search for value v (denoted by search(v)). Suppose that the search starts and terminates at time t_0 and t_1, respectively. For any t, $t \in [t_0, t_1]$, we define $\mathrm{TP}(t)$ and $\mathrm{TP}_{\mathrm{root}}(t)$ as follows. ($\mathrm{TP}(t)$ and $\mathrm{TP}_{\mathrm{root}}(t)$ denote termination positions.) Suppose that at time t we "freeze" the current tree (i.e., its structure) in the following sense. After time t, no process is allowed to make any change on any pointer field, but each process must proceed to the point where it must make a pointer change, or it is blocked by another process. As it so proceeds, it locks and releases the same locks as it would ordinarily. The important point here is that the structure of the tree is not changed, but all processes proceed as far as they can to avoid impeding a search through the tree.

Now consider the continuation of search(v) on the tree frozen at time t, with the search starting from wherever it was at time t. Then with respect to the tree frozen at time t, search(v) may or may not terminate, depending upon whether or not the node it has to lock is already locked by another process. We define $\mathrm{TP}(t)$ to be the termination position of search(v) if it terminates; otherwise $\mathrm{TP}(t)$ is undefined. Similarly, with respect to the same tree frozen at time t, we define $\mathrm{TP}_{\mathrm{root}}(t)$ to be the termination position of a new search that starts searching from the root of the tree for the same value v. $\mathrm{TP}_{\mathrm{root}}(t)$ is undefined if the new search does not terminate.

Property P2′ is stated as follows:

P2′. *For any search which starts at* t_0 *and terminates at* t_1,
$\mathrm{TP}(t) = \mathrm{TP}_{\mathrm{root}}(t)$
for any $t \in [t_0, t_1]$ *for which* $TP(t)$ *is well defined.*

In this new terminology, Property P2 can be expressed as

P2. *For any search,*
$\mathrm{TP}(t_1) = \mathrm{TP}_{\mathrm{root}}(t_1)$
where t_1 *is the termination time of the search.*

It is seen that Property P2′ implies Property P2, since by the definition of t_1, $\mathrm{TP}(t_1)$ is well defined.

3.4 The Correctness Proof of the System

We only need be concerned with Properties P1 and P2′; it is trivial that the system satisfies Properties P3, P4, and P5. By Assumption A1, Properties P1 and

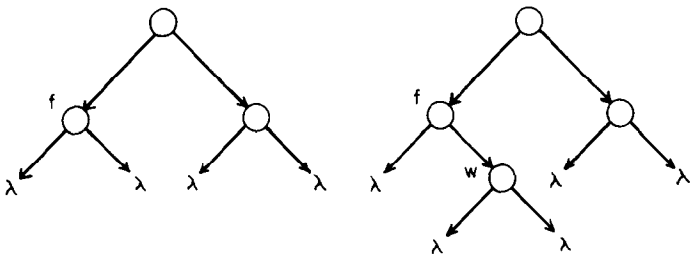

Fig. 3. Trees T^- and T^+ describing the status of the system at times $t - \epsilon$ and $t + \epsilon$, respectively.

P2′ hold initially. We assume (inductively) that Properties P1 and P2′ hold up to time t, when a change to the tree structure, f.dir←w, is made. For proof purposes we may assume that no two operations are done at exactly the same time. Hence we may choose $\epsilon > 0$ so that in the interval $[t - \epsilon, t + \epsilon]$ the change at time t is the only operation done by any process. Let T^- and T^+ be the trees frozen at times $t - \epsilon$ and $t + \epsilon$, respectively. Note that T^+ is the tree resulting from T^- by adding w as the "dir" son of f.[1] This is illustrated in Figure 3, assuming "dir" equal to "right."

We wish to prove the following two assertions (a) and (b).

(a) *Property P1 holds at time* $t + \epsilon$. Consider the insertion responsible for the change at time t. Note that the insertion process is simply a search followed by a pointer change. Since the search satisifies Property P2′ at time $t - \epsilon$, one can view that the change at time t is done by the insertion executed alone on tree T^-. Therefore by Assumption A2 Property P1 holds at time $t + \epsilon$.

(b) *Property P2′ holds at time* $t + \epsilon$. Consider a search process, say, search(v), which starts before time $t - \epsilon$ and terminates after $t + \epsilon$ (i.e., it is in progress when the pointer is changed at time t). By the inductive assumption that Property P2′ holds up to time t, we know that the assertion is true for tree T^-. For the purpose of proving Property P2′ at time $t + \epsilon$, we can assume that $\mathrm{TP}(t + \epsilon)$ is well defined. In the following, we wish to prove that on T^+ the termination position $\mathrm{TP}(t + \epsilon)$ of search(v) coincides with that $\mathrm{TP}_{\mathrm{root}}(t + \epsilon)$ of another search process, namely, find(ROOT, v).

Case 1. $\mathrm{TP}(t - \epsilon)$ is well defined. Then $\mathrm{TP}(t - \epsilon)$ $(=\mathrm{TP}_{\mathrm{root}}(t - \epsilon))$ must not be node f, since f is locked at time $t - \epsilon$ by the insertion process responsible for the change f.dir←w, and $\mathrm{TP}(t + \epsilon) = f$ would make $\mathrm{TP}(t + \epsilon)$ undefined. This implies that $\mathrm{TP}(t)$ and $\mathrm{TP}_{\mathrm{root}}(t)$ are constants over $[t - \epsilon, t + \epsilon]$, since the change f.dir←w has no effect on them. (We rely, of course, on Assumption A2 of the correctness of the pointer change involved.) Therefore $\mathrm{TP}(t + \epsilon) = \mathrm{TP}_{\mathrm{root}}(t + \epsilon)$.

[1] In this paper we use the notation "f.dir" to refer to the node pointed to by node f in the direction specified by "dir" (which is usually a variable), or to refer to a pointer to that node. The notation "dir′" refers to the direction complementary to "dir". Hence if dir = left then dir′ = right, and vice versa.

Case 2. TP($t - \epsilon$) is not well defined. Since TP($t + \epsilon$) is well defined, in defining TP($t - \epsilon$) the continuation of the search, search(v), on T^- must be blocked at node f. There are two cases, depending on the state of the process search(v) at time $t - \epsilon$:

(i) The process search(v) has not yet examined f.dir at time $t - \epsilon$. Then on T^+ search(v) will correctly reach either f.dir′ or f.dir as find(ROOT, v) does, since the search uses the updated f.dir.

(ii) The process search(v) has read f.dir as λ at time $t - \epsilon$. Then on T^+, search(v) will find f.dir ($= w$) $\neq \lambda$ and thus start searching from f. This implies that search(v) will again correctly reach w as find(ROOT, v) does.

We have shown that the pointer change done by an insertion process preserves Properties P1 and P2′. Therefore, by induction, Properties P1 and P2′ always hold.

3.5 Comments on Locking

Notice that the *find* procedure locks the father of the node whose key has the value for which *find* is searching. (Consequently, the *search* procedure also locks that node.) However, for purposes of simply reporting the "instantaneous" existence of a key in the database the *find* and *search* procedures can be modified by deleting the lock/unlock calls to provide the ability to search without locking. While we have omitted those versions of *find* and *search* from the present paper for purposes of clarity, we would certainly include such procedures in a full system, where simultaneous examination of nodes by several processes was likely to occur.

Notice, however, that locking the associated record is often necessary. For example, locking would be used in the case that some modification will be made to the associated record, once the key is found (see Section 2.5). In this case we would lock the record (or possibly some segment of the record) to prevent change by another process.

Similarly, record locking may be necessary during prolonged examination of a node and its record. For example, if we wish to guarantee that a key continues to exist while we examine its record, then we must lock the node containing that key to prevent deletion by another process. Again, we might instead wish to lock some segment of the record to prevent modification to that segment.

4. A SEARCH-INSERTION-ROTATION SYSTEM

In this section we extend the system of Section 3 to include rotation processes. Important ideas of this paper such as the use of back pointers, copying, and blue nodes are introduced in this section.

4.1 An Example

The following example illustrates the kind of problems we might encounter when rotations are executed concurrently with search. Consider Example 4.1 for rotating and searching the tree shown in Figure 4a.

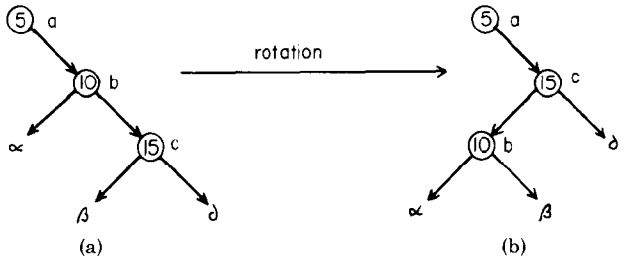

Fig. 4. Rotating (b, c) to the left; α, β, and δ are subtrees.

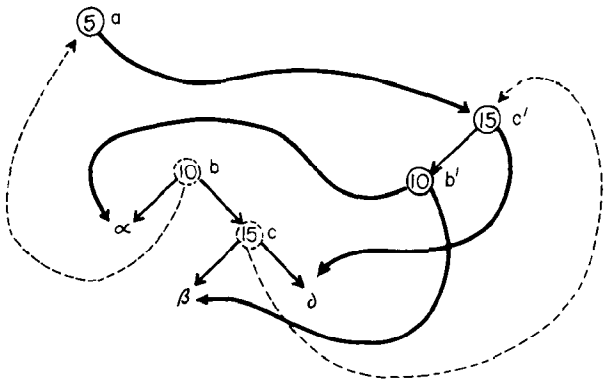

Fig. 5. Results of rotation (of the tree in Figure 4a) using the idea of copying. In the diagram, blue (or garbage) nodes are indicated by dotted circles, and back pointers by dashed lines.

Example 4.1

	search(20)	rotation
1.	$s \leftarrow a$	
2.	$20 > s$.value (= 5)	
3.	$s \leftarrow s$.right(= b)	
4.	$20 > s$.value (= 10)	
5.		a.right$\leftarrow c$
6.		c.left$\leftarrow b$
7.		b.right$\leftarrow \beta$
8.	$s \leftarrow s$.right(= b.right = β)	

Note that at step 8 the search starts to search subtree β for value 20 in the rotated tree (i.e., the tree in Figure 4b), while at this time a search from the root for the same value 20 (in the same tree) will terminate in subtree δ. Property P2 is therefore violated. Our solution to this problem is to first establish a rotated version of the structure in a copy outside the tree. (In particular, we create copies b' and c' of nodes b and c in Figure 5.) We then connect the copy into the tree by changing just one pointer from node a, *which is an indivisible operation.* The nodes in the old structure are changed to blue nodes and inserted into GC-queue, and are to be collected by garbage collectors. (The garbage collection process is described in Section 6.) By providing "back pointers," we ensure that those search processes which are at blue nodes can still come out to reach their "correct"

termination positions. The result of rotating the tree shown in Figure 4a using this new method is illustrated in Figure 5.

4.2 The System

The *rotation* process, which follows, performs a rotation by building a copy of the section of the tree to be altered and then replacing the old section with the modified new section. In this version of the *rotation* procedure we include some code that will only be useful when used (in Section 5) with the deletion procedure. For example, locking the new nodes (b' and c') is unnecessary for the rotation procedure itself, since once the procedure switches from the old version to the new version by a pointer change, it no longer uses b' and c'.

Also of interest is the use of a "back pointer," which was mentioned earlier. This pointer has no meaning for ("white") nodes that are part of the tree. However, for "blue" nodes, the back pointer is used to continue the search when the father of the node for which *find* is searching has been deleted (made blue) while *find* was deciding whether the node was, in fact, the father of the desired node.

The Rotation Process

Rotation. Suppose that a, a.dir1, and [a.dir1].dir2 are three consecutive white nodes on a path. The following procedure moves a.dir1 away from the path by performing a rotation.[2] It is assumed that a and a.dir1 are locked when this procedure is called. The procedure returns (a, c', b') where $c' = a$.dir1 and $b' = c'$.dir2′, with c', b' locked.

```
procedure rotation(a, dir1, dir2)
b←a.dir1;
c←b.dir2;
create(b'); create(c');                                  /*Set up new nodes*/
lock(c'); lock(b'); lock(c);
c'.dir2←c.dir2; c'.dir2'←b'; c'.value←c.value;
b'.dir2←c.dir2'; b'.dir2'←b.dir2'; b'.value←b.value;
a.dir1←c';                                               /*Change the tree*/
b.back←a;                                                  /*Back pointers*/
color b blue;                                              /*And blue nodes*/
c.back←c';
color c blue;
enqueue nodes b and c in GC-queue;                       /*For garbage collection*/
unlock(a); unlock(b); unlock(c);                           /*Unlock a, b, c*/
return (a, c', b')
```

The search and insertion processes are the same as those defined in Section 3. But the procedure find(n, v) must be redefined (as follows) to handle the presence of blue nodes. In particular, it must consider the possibility that f became blue between the decision to lock it and the actual locking of the node.

Find. The modified find procedure. This version of *find* uses deleted ("blue") nodes and back pointers in order to continue searching from a deleted node.

[2] In this paper we do not wish to restrict our result to any specific type of balanced tree such as the AVL tree. Therefore, for the balancing purpose, schemes of deciding where rotations should take place will not be specified. Recent schemes by Guibas and Sedgewick [9] on detecting rotations to be performed based on local information seem to be particularly suitable to our systems.

```
procedure find(n, v)
f←n;
if v < f.value then dir←left else dir←right fi;
s←f.dir;                                              /*Find son*/
if s ≠ λ and s.value ≠ v then
  return find(s, v)                                   /*Next level*/
else
  lock(f);
  if f is blue then                                   /*Just missed getting node*/
    unlock(f);
    return find(f.back, v)                            /*Follow back pointer from blue node*/
  else
    if s ≠ f.dir then
      unlock(f);                                      /*Some process changed it*/
      return find(f, v)
    else return(f, dir)                               /*Found it*/
    fi
  fi
fi
```

4.3 The Correctness Proof of the System

We only need be concerned with Properties P1, P2′, and P3; it is trivial that the system satisfies Properties P4 and P5. Since a rotation process always locks nodes on a path in top-to-bottom order, there is no danger of deadlock. Hence Property P3 is satisfied. We now prove that Properties P1 and P2′ hold. This proof uses the framework and terminologies established in Section 3.4.

By Assumption A1, Properties P1 and P2′ hold initially. We also assume (inductively) that they hold up to time t, when a change performed by a rotation process is made to the tree structure. We need not be concerned with changes due to insertions, since, by the results of Section 3, we know that insertions will preserve Properties P1 and P2′. Define ϵ, T^-, and T^+ as in Section 3.4.

(a) *Property P1 holds at time* $t + \epsilon$. Note that the rotation process locks all the nodes it reads and writes. Hence the proof follows directly from Assumption A2.

(b) *Property P2′ holds at time* $t + \epsilon$. As before, consider a search process, say, search(v), which starts before time $t - \epsilon$ and terminates after $t + \epsilon$. By the inductive assumption that Property P2′ holds up to time t, we know that the assertion is true for tree T^-. Again, we assume that $\mathrm{TP}(t + \epsilon)$ is well defined and want to prove that on T^+ the termination position, $\mathrm{TP}(t + \epsilon)$, of search($v$) coincides with the termination position, $\mathrm{TP}_{\mathrm{root}}(t + \epsilon)$, of find(root, v).

Case 1. The change at time t is a.dir1←c' or b.back←a (cf. Figure 6).

(i) $\mathrm{TP}(t - \epsilon)$ is well defined. Then $\mathrm{TP}(t - \epsilon)$ $(= \mathrm{TP}_{\mathrm{root}}(t - \epsilon))$ must not be a, b, or c, since they are all locked at time $t - \epsilon$. This implies that $\mathrm{TP}(t)$ and $\mathrm{TP}_{\mathrm{root}}(t)$ are constant over $[t - \epsilon, t + \epsilon]$, since the change at time t has no effect on them. That is, search(v) or find(root, v) will terminate at the same node on either T^- or T^+. Therefore, $\mathrm{TP}(t + \epsilon) = \mathrm{TP}_{\mathrm{root}}(t + \epsilon)$.

(ii) $\mathrm{TP}(t - \epsilon)$ is not well defined. Then search(v) on T^- must be blocked at some node. It is still blocked on T^+, since no lock will be released

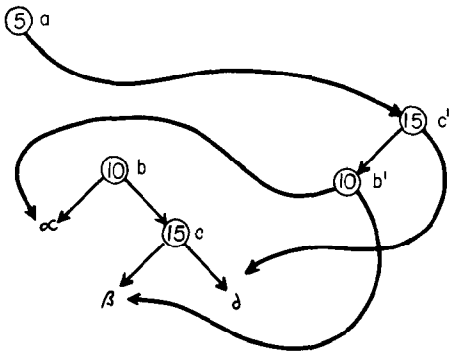

Fig. 6. The new tree formed after the operation a.dir1←c'.

after the change at time t and before the next pointer change. This contradicts the assumption that TP($t + \epsilon$) is well defined..

Case 2. The change at time t is c.back←c'.

(i) TP($t - \epsilon$) is well defined. The proof that TP($t + \epsilon$) = $\text{TP}_{\text{root}}(t + \epsilon)$ is the same as that in part (i) of Case 1.

(ii) TP($t - \epsilon$) is not well defined. Then we can conclude that search(v) on T^- must be blocked at a, b, or c, since (1) these are the only nodes whose locks will be released on T^+ by the action at time t and (2) search(v) is unblocked at time t (recall that TP($t + \epsilon$) is well defined). Further, on T^+ search(v) will always come out from the garbage nodes to reach correct white nodes. The procedure find(root, v) does this, by utilizing the back pointers to resume the search through the tree.

5. A SEARCH-INSERTION-ROTATION-DELETION SYSTEM

In this section, we further extend our system to include concurrent deletion processes. Unlike other operations considered so far, deletion is not a "local" operation in the sense that it may have to make changes in two sections of the tree that are arbitrarily distant from each other. That is, the node to be deleted and the node with which it is to be replaced can be arbitrarily far apart. This makes the deletion operation difficult to deal with in a concurrent system where only a constant number of nodes may be locked by a process at any time. In this section we demonstrate how "nonlocal" operations such as the deletion operation can still be correctly incorporated into a concurrent system using only "local" locks. This is achieved through the repeated use of the rotation process introduced in Section 4.

5.1 An Example

Example 5.1 and Figure 7 illustrate that an existing searching process may become incorrect when another deletion process is executing concurrently.

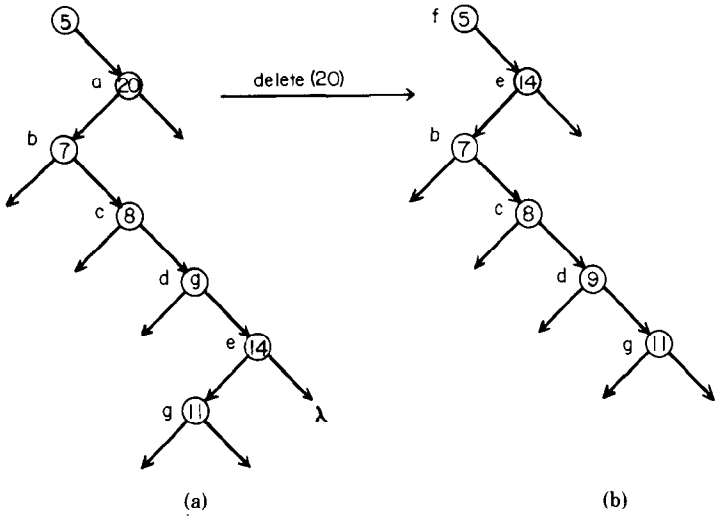

Fig. 7. Deletion of node a with value 20.

Example 5.1

	search(15)	delete(20)
	⟨*previous steps*⟩	
1.	15 > *f*.value (= 5)	
2.	*s*←*f.right* (= *a*)	
3.	15 < *s.value* (= 20)	
4.	*s*←*s*.left (= *b*)	
5.		⟨search(20); obtain node *a*⟩
6.		⟨search for the node in the left subtree of *a* which has the largest value (node *e*, in this case)⟩
7.		replace *a* with *e*
8.	15 > *s*.value (= 7)	

After step 7, the searching process searches for value 15 in the left subtree of node e (cf. the tree in Figure 7b). Property P2 is not satisfied because find(root, 15) would search the right rather than the left subtree of node e.

In general, suppose that node a is the node to be deleted and node a' is the node with which node a is to be replaced. Then any active search process that has passed node a while searching for a value between a'.value and a.value will become inconsistent after the deletion. In the following we propose a method for dealing with this problem.

5.2 The System

The search, insertion, and rotation processes are the same as those defined in Section 4. The deletion process is described as follows.

Note that if at least one son of the node to be deleted is λ (which should occur with more than 0.5 probability), then the deletion is very simple. This is illustrated in Figure 8. The procedure *remove* defined below performs this simple deletion. Briefly, the procedure works by changing the pointer from the father (a) of the node (b) to be deleted to point "around" that node. (This only works when b has

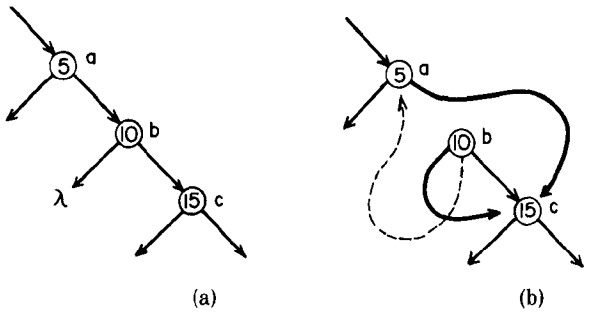

Fig. 8. The simple deletion case.

only one son, since node a cannot use the same pointer to point to two nodes simultaneously.) This operation removes b from the tree. A back pointer is provided (which points from b to a) for any process that was searching at b when a.dir was changed, so that the search can continue correctly.

The Deletion Process

Remove. This procedure removes a node (a.dir1) when it is known that one son of that node ((a.dir1).dir2′) is λ. Nodes a and a.dir1 are locked when the procedure is called, and are unlocked when the procedure ends.

```
procedure remove(a, dir1, dir2)
b←a.dir1;
c←b.dir2;
a.dir1←c;                                   /*Point around b*/
b.dir2'←c;                  /*Redirect search from b: b.left = b.right = c*/
b.back←a;                                   /*Provide back pointer*/
color b blue;                               /*And blue node*/
enqueue node b in GC-queue                  /*For garbage collection*/
unlock(a);
unlock(b);
```

The deletion process described below is formulated as two steps: (1) find the correct node and (2) delete it (handled by the procedure *deletion-by-rotation*).

Delete. This procedure deletes a node with value v from the tree, if such a node exists.

```
procedure delete(v)
(f, dir)←find(root, v);
if f.dir = λ then
   print "Value v is not in the tree";
   unlock(f);
else
   s←f.dir;
   lock(s);
   deletion-by-rotation(f, dir)                 /*Do the dirty work*/
fi
```

The procedure deletion-by-rotation(f, dir) is defined below. Since simple deletion by the *remove* procedure is sometimes not possible, this procedure moves the node to be deleted down in the tree to a place where that action *is* possible.

It does this by repeatedly rotating the node to a lower position in the tree (using the *rotation* procedure defined in Section 4), until it is possible to call *remove* to actually delete the node. After this has been accomplished, the procedure works its way back up the tree in an attempt to rebalance the tree. In particular, the procedure moves the node down the tree by recursive calls on itself. After deletion, it rebalances by going back up the tree (again using *rotation*), after each recursive call returns.

In the version of the *deletion-by-rotation* procedure that we give here, all operations are biased in one direction for purposes of clarity and simplification of the algorithm. This directional bias is not necessarily unreasonable if the deletion starts on a balanced tree or if the information about the structure of the tree is not available. If one were striving for efficiency, one could add additional code to optimize the direction in which rotations and removals were to be done, using information about the structure of the tree.

Note that in the call to *rotation*, in the returned triple (f, g, h), h is the new copy of the node to be deleted, and f is identical to the procedure parameter f.

Deletion-by-Rotation. The following procedure deletes node f.dir by (recursively) performing a sequence of rotations that serve to move f.dir to a position lower in the tree where it can be removed by the procedure *remove* given above (ending the recursion). Nodes f and f.dir are locked when the procedure is called. The procedure also rebalances the tree after deletion when such rebalancing is still possible. The procedure ends with no locks set.

```
procedure deletion-by-rotation(f, dir)
s←f.dir;
if s.left = λ then remove(f, dir, right)                    /*End recursion*/
                                          /*(Note. Example of directional bias)*/
else
  (f, g, h)←rotation(f, dir, left);                         /*Move f.dir down*/
  if h.left = λ then
                                  /*Do not need to rebalance on last recursive call*/
    deletion-by-rotation(g, right);
  else                                                      /*Do recursion and rebalance*/
    deletion-by-rotation(g, right);                         /*Recursive call*/
                                        /*N.B. at this point, no nodes are locked*/
    lock(f);                                                /*Begin rebalance*/
    if g ≠ f.dir or f is blue then
                                   /* Cannot rebalance, since things have changed*/
                                     unlock(f)
    else
       lock(g);
      (f, g', h')←rotation(f, dir, right);
      unlock(g');
      unlock(h')
    fi
  fi
fi
```

It is relatively easy to check that the inclusion of the procedure *remove into the system of Section 4* preserves its correctness. If the system in the current section is deadlock-free, then we can conclude that it is correct, since it is built from the procedure *remove* and the system in Section 4. To show that the system

is deadlock-free, we note that at any level of recursion, when the *deletion-by-rotation* call returns, no nodes are locked. For each level at which rebalancing is attempted, therefore, only new locks are used. Furthermore, they are applied using the top-down discipline. Thus deadlocks cannot occur in the system.

An alternative solution to the problem concerning deletions would be to simply leave locked all nodes locked by the deleter, and then unlock them "on the way back up," after rebalancing. However, this would violate our constraint of never locking more than a constant number of nodes at one time.

6. GARBAGE COLLECTION

In this section we consider the problem of correctly appending garbage nodes to the free list.

6.1 An Example

The following example illustrates that for garbage collection one cannot simply append blue nodes to the free list. Refer to Figure 8.

Example 6.1

search(20)	delete(10)
1. $r \leftarrow a$	
2. $20 > r$.value($= 5$)	
3. $r \leftarrow r$.right ($= b$)	
4.	$\langle delete(b) \rangle$
5.	*garbage-collect*(b)
6. compare r.value($= b$.value) with 20	

Note that the comparison in step 6 is erroneous, since node b no longer exists in the tree. It should have been left (blue) and not garbage collected so the search could have recovered from the deletion. This is why—in the procedures given above—we only enqueue blue nodes to be garbage collected. The garbage collector must be careful not to collect a node to which another process might still have access.

6.2 Remarks

Rules for a garbage collector are simply that it not collect garbage too soon, but that it also not have to wait "too long." These can be stated more formally as

(1) Let f be a node that is detached from the tree. If f is referenced—or *can* be referenced—by any process, then f is not yet garbage ready to be collected.
(2) When the garbage collector prepares to collect node f, it only has to wait for that particular node (not for later copies of the node) to no longer be referenceable.

6.3 A Solution

In this section we offer a simple solution with a single garbage collector. In Appendix A we sketch modifications to the concurrent tree manipulation processes that allow a number of garbage collectors to operate concurrently with the tree mutators.

Perhaps the simplest way to solve the problem is as follows. Periodically, the garbage collector freezes the garbage collection queue (GC-queue) in the following sense. It locks the queue (against any insertion to it by the tree mutators), copies it (to, say, a queue GC′-queue), resets the original queue (GC-queue) to its emply state, and unlocks it. (Copying GC-queue and resetting it can be done in constant time for arbitrarily long queues if the queue is stored as a linked list.) Then, it waits until all of the currently running processes have terminated. These are the processes that started running before GC-queue was locked by the garbage collector, i.e., the processes which might access the garbage (blue) nodes in GC′-queue. (Such a wait might be implemented, for example, by having each process enter in a log the time when it starts and terminates. The GC process would then wait until "OUT" entries appeared in the log for each process that had an "IN" entry, but no "OUT" entry, at the time the GC-queue was locked.) After this wait, the garbage collector returns each of the garbage nodes in GC′-queue to the free list by using the append procedure defined below.

Append. This procedure returns a node to the free list.

```
procedure append(n)
n.color←white;
n.right←λ;
lock(FREE);
[FREE.right].right←n;
FREE.right←n;
unlock(FREE)
```

With this solution, blue (or garbage) nodes may not be appended to the free list for some long period of time after they become garbage. This is undesirable for situations where space utilization is crucial. Note, however, that, because white nodes never point to blue nodes in our systems, the existence of blue nodes has no effect on the speeds of those searches through the tree which started after these nodes had become garbage nodes. Also, the execution of the *append* procedure is carried out in parallel with other processes. Thus it appears that this simple solution is quite acceptable as far as search speeds are concerned.

7. SUMMARY AND CONCLUDING REMARKS

In this paper, we have examined some of the details of a particular problem in concurrent database manipulation. To the authors' knowledge, many of the properties of the systems presented are not achievable on the basis of any existing general theory on concurrency control. For example, in the two-phase locking scheme offered by Eswaran et al. [7], a search process would be required to lock all nodes in the search path, and would not release any of these locks until the end of the search. The special structure of binary search trees enables us to design concurrent systems enjoying a high degree of concurrency. For further discussion and results on how special information about a problem can help the design of efficient concurrent database systems, see [11, 12].

We summarize some of the important contributions embodied in the concurrent binary search tree systems presented in this paper:

(1) The algorithms use neither reader locks, nor exclusive locks. Only writer-exclusion locks are used, simply to prevent the obvious problems engendered

by simultaneous update of a node by more than one process. The locking scheme used to apply these locks is simple. In particular, it is implementable without the overhead incurred by a queue manager or a system supervisor.

(2) The size of the region of the tree which is locked by a process at any time is bounded by a (small!) constant.

(3) The idea of *copying*—doing large amounts of work outside the data structure and then indivisibly introducing all of the changes simultaneously—is a useful technique for removing some of the inherent complexity of concurrent operations.

(4) The *back pointers* and *blue nodes* are a specific instance of the idea of a general mechanism for recovery from some of the "confusion" caused by concurrency. Such a mechanism is provided for use by processes whose earlier actions have become invalid as a result of the actions of another process.

(5) In order to take full advantage of the power of multiprocessing, we introduce the idea of *postponement*. This is embodied by the rule: "A process should only do what it has to do." Often, nothing is lost by allowing a second process to continue the work begun by a first process. In fact, waiting time may sometimes be avoided by postponing work (e.g., collection of the garbage nodes produced by a process is postponed and is eventually performed by a garbage collector process).

(6) We present a fairly rigorous proof of the correctness of our concurrent systems. In doing so we demonstrate that such correctness can be proved, and we develop techniques for use in these proofs.

(7) Two garbage collection mechanisms are offered. These are auxiliary to the main tree system. They allow us to further exploit the concurrency available by using some of the techniques mentioned above (copying, postponement, etc.), and decoupling the necessary garbage collection from the main tree operations (insertion, deletion, reorganization).

Binary search trees represent a very simple structure for storing data. Further work should try to extend some of the ideas presented in this paper to more general database systems.

APPENDIX A. CONCURRENT GARBAGE COLLECTION

A1. The Problem

In Section 6 we presented a simple approach to the garbage collection problem. Here we sketch modifications to the set of concurrent processes given in the paper which will allow a set of concurrent garbage collectors to operate correctly in parallel with the tree mutators. The garbage collectors will never incorrectly collect a blue node that might still be used by some process (as was described in Example 6.1). This scheme should be used when space utilization is important, since garbage is collected and returned to the free list very quickly (relative to the batched collection suggested in Section 6).

Concurrent garbage collection in a list processing environment has recently received much attention (see, for example, [5] and [13]). The problem considered in this appendix is different in that the safety of collecting a garbage node depends

on whether or not it is reachable by any existing or future search; knowing that the node is not reachable from the root does not ensure the safety of collecting the node. The idea of the method described in this appendix is to use reference counts to guarantee that only blue nodes that are no longer reachable by any existing or future search will be garbage collected. Note that in a tree there are no cycles; so we do not encounter some of the problems of using reference count schemes in general list processing. We assume that in addition to the usual fields, each node also has two reference counters: node.ref[0] and node.ref[1], and one index (or indicator) field, node.index, to designate which counter is currently in use for that node. The field node.index can take either 0 or 1 as a value to designate one of the two reference counts. We further assume that interchanging between these two values (denoted "comp" for complement) is an indivisible operation and that incrementing and decrementing reference counts (denoted "inc" and "dec") can also be done indivisibly.

A2. The System

In this section, we demonstrate the modifications to the procedures given above that will allow concurrent garbage collection as described.

The search process used in previous sections must be redefined to handle updates for reference counts. For any node n, n.ref[0], n.ref[1], and n.index are initially set to zero by the *create* procedure.

The Search Process

Search. This procedure searches for a node in the tree with a given value v.

```
procedure search(v)
j←ROOT.index;
inc ROOT.ref[j];                                   /*Reference ROOT*/
(f,dir)←find(ROOT, j, v);
s←f.dir;
if s ≠ λ then print "Value v is at node s"
else print "Value v is not in the tree" fi;
unlock(f)
```

The procedure find(n, i, v) is redefined as follows. When the procedure is called, f.ref[i] has already been incremented by the search process which calls the procedure.

Find. This procedure recursively performs the search, modifying reference counts as appropriate.

```
procedure find(n, i, v)
f←n;
if v < f.value then dir←left else dir←right fi;
s←f.dir;                                           /*Find son*/
if s ≠ λ and s.value ≠ v then
    {j←s.index; inc s.ref[j]};                     /*Reference son*/
                          /*(Operations inside {...} assumed indivisible.)*/
    dec f.ref[i];                                  /*Then dereference father*/
    return find(s, j, v)                           /*Next level*/
```

```
else
  lock(f);
  if f is blue then
    unlock(f);
    t←f.back;                                   /*Get pointer to back son*/
    {j←t.index; inc t.ref[j]};                  /*Reference it*/
    dec f.ref[i];                               /*Then dereference this one*/
    return find(f.back, j, v)                   /*Follow back pointer*/
  else
    if s ≠ f.dir then
       unlock(f);                               /*Lost it*/
       return find(f, i, v)
    else                                        /*Found it*/
      dec f.ref[i];
      return(f,dir)
    fi
  fi
fi
```

The new version of the *insert* procedure follows:

The Insertion Process

Insertion. This procedure inserts a node with value v into the tree (at one of the leaves), if no such node already exists in the tree.

```
procedure insert(v)
j←ROOT.index;
inc ROOT.ref[j];                                /*Reference ROOT*/
(f, dir)←find(ROOT, j, v);
if f.dir ≠ λ then
  print "Value v is already in the tree";
  unlock(f)
else
  create(w);                                    /*Build a node*/
  w.left←λ;
  w.right←λ;
  w.value←v;
  f.dir←w;                                      /*Point to it*/
  unlock(f)
fi
```

The new *delete* procedure follows:

Delete. This procedure deletes a node with value v from the tree, if such a node exists.

```
procedure delete(v)
(f, dir)←find(ROOT, v);
if f.dir = λ then
  print "value v not in tree";
  unlock(f);
else
  s←f.dir;
  lock(s);
  deletion-by-rotation(f, dir)                  /*Do the dirty work*/
fi
```

The procedure *deletion-by-rotation* is modified as follows:

Deletion-by-Rotation

```
procedure deletion-by-rotation(f, dir)
i←f.index;
inc f.ref[i];
s←f.dir;
if s.left = λ then remove(f, dir, right)                    /*End recursion*/
                                          /*(Note. Example of directional bias)*/
else
  (f, g, h)←rotation(f, dir, left);                         /*Move f.dir down*/
  if h.left = λ then
                                  /*Do not need to rebalance on last recursive call*/
    deletion-by-rotation(g, right);
  else                                                      /*Do recursion and rebalance*/
    deletion-by-rotation(g, right);                         /*Recursive call*/
                                        /*N.B. at this point, no nodes are locked*/
    lock(f);                                                /*Begin rebalance*/
    if g ≠ f.dir or f is blue then      /*Cannot rebalance, since things have changed*/
      unlock(f)
    else
      lock(g);
      (f, g', h')←rotation(f, dir, right);
      unlock(g');
      unlock(h')
    fi
  fi
fi
dec f.ref[i]
```

Create. This procedure creates a new node named w, by removing it from the free list. It also sets the reference counts and index for w to zero.

```
procedure create(w)
lock(FREE);
if FREE.left = FREE.right then
  Abort the process which calls create and inform
  the system that the free list is empty
else
  w←FREE.left;
  FREE.left←[FREE.left].right
fi
unlock(FREE);
w.ref[0]←0; w.ref[1]←0; w.index←0
```

The procedure *append*, *remove*, and *rotation* are the same as that given above in the main part of the paper.

We include in the algorithm below steps to handle the multiple garbage collectors case. For this purpose, we require the use of the additional field for each node mentioned above—the GC-lock field. Garbage collectors use this lock to prevent confusion caused by switching a reference count field while another garbage collector is still using it. There is also a single lock on the entire GC-queue; this lock is also used by the enqueue operation in the *rotation* and *remove* procedures. Moreover, for technical reasons, in the *rotation* procedure we en-

queue the triple (a, b, c) rather than the nodes b and c, and in the *remove* procedure we enqueue the pair (a, b) rather than node b.

The Garbage Collection Process

Garbage-Collector. This process appends garbage nodes to the free list.

```
lock(GC-queue);                                   /*Single lock for GC-queue*/
get (a, b, c) (or (a, b) for which the algorithm is
  similar) from GC-queue;
lock a for GC;                                    /*Set GC-lock field of a*/
unlock(GC-queue);
a.index←comp a.index;                             /*Switch a*/
i←comp a.index;                                   /*Old counter*/
while a.ref[i] > 0 wait;                          /*Let old processes drain*/
unlock a for GC;                                  /*We are done with it*/
lock(b);                                          /*Make sure no GC is*/
unlock(b);                                        /*Using b*/
i←b.index;
while b.ref[i] > 0 wait;                          /*Be sure everyone done with b*/
append(b);                                        /*Append b to free list*/
if we got (a, b) instead of (a, b, c) then return;  /*Done*/
lock(c);                                          /Make sure no GC is using it*/
unlock(c);
i←c.index;
while c.ref[i] > 0 wait;                          /*Be sure everyone done with c*/
append(c)                                         /*Append c to free lis*/
```

The procedure append(n) was defined in Section 6.

A3. Comments and Justification

Note the simplicity of the GC operation when taken sequentially for collection of nodes b and c. It consists of switching the counter on node a (thus searches arriving at a after the switch will increment the reference count in the new counter), letting the old processes "drain" from a, letting b drain, freeing b, and if we have a triple (a, b, c), then letting c drain and freeing c.

The concurrent garbage collection works simply because we do the switch on a after b becomes garbage. Then we let all old processes drain from a. After this step, any process which can access a must access it at some time after a enters the GC-queue, which guarantees that it accesses a after a no longer points to b. Then we simply have to wait for all old processes to drain from b. This means that b is safe to free.

Further, suppose we are running concurrent garbage collectors that might interfere with each other. We first observe that only one tuple in the GC-queue can have any node, b, as the nonfirst node. Otherwise that node would have been deleted from the tree twice before being returned to the free list. This is clearly impossible based on the operation of the deletion and rotation (and search) algorithms.

Now suppose that two garbage collectors are working, say, on the tuples (a, b) and (b, c). (The case of ordered triples instead of ordered pairs is an easy generalization.) Then we can show that it is impossible that the tuples were removed from the GC-queue in the order: (a, b), (b, c). This is, of course,

equivalent to being placed in the queue in that order. If this were so, then consider placing (b, c) in the queue. Placing this tuple in the queue implies that the node c was removed from the tree, and that node b was the father of c, but was still in the tree. However, at the time that (b, c) was placed in the queue, (a, b) was already in the queue, implying that node b had previously been removed from the tree. For (b, c) to be placed in the queue, both b and c must be locked; similarly for (a, b). But then after (a, b) is placed in the queue, b must be unlocked before it can be locked by the process that locks b and c and places (b, c) in the queue. Therefore, this latter process locks a node that has been deleted from the tree. However, such a lock (on a blue node) is checked for, and immediately released if detected. Therefore, a node that had been deleted from the tree would not be the first element in an ordered pair (triple) placed in the GC-queue. This contradicts the placement of (b, c) in the queue after (a, b).

Therefore, we know that the tuples occur in the order: (b, c), (a, b).

Lastly, we observe that a garbage collector, say g, locks node b (from tuple (a, b) or (a, b, c)) to guarantee that other garbage collectors (those using it in tuples of the form (b, c) or (b, c, x)), say g', are done with it. Any such g' would lock node b while in the critical section of the garbage collector (protected by lock/unlock(GC-queue)). This means that g cannot lock b until all tuples (b, x)—that occur in the queue before (a, b)—have been processed to the extent that they have locked (and then unlocked!) b. Therefore, node b will not be garbage collected by g until it is safe to do so.

APPENDIX B. RELATED WORK

In this appendix, we discuss alternative solutions to the problem presented in this paper, and related work that has been done. In discussing alternative solutions, we point out some of the advantages and disadvantages of each.

For examples of the design and construction of multiprocessors see Wulf and Bell [24], and Swan, Fuller, and Siewiorek [22]. For examples of verification methodology, see Dijkstra's book [4], and the comprehensive survey by Manna and Waldinger [17] (and its references). For extensions of verification ideas to parallel programs, see the work by Owicki [19] and Lamport [14]. In the database systems area, research in concurrency and integrity control has been done, for example, by Eswaran et al. [7], Gray [8], and Ries and Stonebreaker [20].

Since B-trees (see Bayer and McCreight [2] or Knuth [10]) have been found convenient for storing large amounts of data in the sequential case, many database systems have been constructed using B-trees (or often B^*-trees; see Wedekind [23]) as the main data structure (e.g., Astrahan et al. [1]). These structures have the advantage that they are balanced by definition (although this does not preclude the necessity of other forms of reorganization). While we chose to examine the structure of binary search trees, much similar work on the question of concurrent operations on B-trees and B^*-trees has been done. We note, however, that the branching factor in most practical B-trees is such that the number of levels required to store large amounts of data rarely exceeds four. This raises the question of just how much concurrency we can squeeze into such a flat structure.

(a) The first solution to the concurrent B-tree problem was advanced by Samadi [21]. His approach is to use semaphores to lock exclusively the path along which modifications may take place, effectively locking the entire subtree of the highest node locked.
(b) Bayer and Schkolnick [3] improve upon this by proposing a parameterized algorithm for concurrent B^*-tree manipulation. This algorithm locks upper sections of the tree with writer-exclusion locks (which *do not* lock out readers), until the actual modifications need to be done (when exclusive locks are finally applied), thus increasing the concurrency of the algorithm.
(c) Miller and Snyder [18] are working on a solution which locks a region of the tree of bounded size (which is close to our notion of locking a region of constant size). This locked region propagates up the tree, performing appropriate modifications to the tree structure.
(d) Ellis [6] presents a solution for 2–3 trees (generalizable to B-trees) which uses several methods to enhance concurrency. These methods include an application of Lamport's idea for correctly reading and writing simultaneously [15]. The algorithms Ellis presents allow temporary departures from the tree structure in order to minimize the cost of maintaining consistency during concurrent operations. Also, "relaxing a process's responsibility to do its own work" is a specific case of our idea of *postponement*; the structural degradation caused by one process may be fixed later by another.
(e) A paper by Lehman and Yao [16] will contain a more extensive survey of these ideas, along with a concurrent B^*-tree algorithm that uses some of the ideas in the present paper to achieve minimal (constant size) locking and high concurrency.

Guibas and Sedgewick's [9] scheme for representing many types of tree structures as "dichromatic" binary trees suggests that the problems of concurrently maintaining more general trees may be reducible to the set of problems studied in the present paper.

APPENDIX C. A CORRECTNESS CRITERION FOR CONCURRENT SEARCH SYSTEMS

The binary search tree or *any* physical database storage structure can be viewed as an implementation of the abstract notion of some data storage mechanism. The abstract notion specifies properties of various operations that we wish to perform on the database.

In this paper, we have adopted a natural abstract notion. That is, the responses given by the search processes must correctly reflect the results of the modifying operations on the database. For example, if the following operations take place (shown in order of termination time) then the responses given by the search processes must be the ones shown on the right-hand side.

```
insert(1)
insert(2)
search(2)     response: "Yes."
delete(2)
search(1)     response: "Yes."
search(2)     response: "No."
  ⋮
rotate
  ⋮
```

That is, *search*(v) returns the answer "Yes" if and only if the number of successful *insert*(v) operations which have terminated so far is strictly larger than the number of successful *delete*(v) operations which have terminated. We use the same abstract notion for *concurrent* database systems. We say a concurrent system is *correct* if it implements the abstract notion.

It is necessary to define more precisely what we mean by "termination time of a process" in a concurrent environment. We define this as the instant at which an updating process makes its last modification to the database link structure or the instant at which a query process reports the result of its search. We can easily argue that Properties P1–P5 are sufficient for the correctness criterion stated here. (Actually, they also guarantee no deadlocks and completion of all processes.) Since by Property P5 a value v will not be added to or deleted from the tree without using *insert*(v) or *delete*(v), respectively, we only need to check that a *search*(v) returns "Yes" (i.e., finds v) if and only if v is in the tree. This is guaranteed by Properties P1 and P2; Property P1 ensures that the tree is always consistent, and thus by Property P2 the search will find v if and only if it is in the tree (since the *search* process on the "frozen" tree is correct in the sense of Assumption A2).

REFERENCES

1. ASTRAHAN, M.M., ET AL. System R: Relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.
2. BAYER, R., AND MCCREIGHT, E. Organization and maintenance of large ordered indexes. *Acta Inf. 1*, 3 (1972), 173–189.
3. BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on B-trees. *Acta Inf. 9*, 1 (1977), 1–21.
4. DIJKSTRA, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
5. DIJKSTRA, E.W., LAMPORT, L., MARTIN, A.J. SCHOLTEN, C.S., AND STEFFENS, E.F.M. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM 21*, 11 (Nov. 1978), 966–976.
6. ELLIS, C.S. Concurrent search and insertion in 2-3 trees. Tech. Rep. 78-05-01, Dep. Computer Science, Univ. Washington, Seattle, 1978.
7. ESWARAN, K.P., GRAY, J.N., LORIE, R.A., AND TRAIGER, I.L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
8. GRAY, J. Notes on database operating systems. *Lecture Notes in Computer Science* 60: Operating Systems, Berlin, Germany, Feb. 1978, pp. 393–481.
9. GUIBAS, L.J., AND SEDGEWICK, R. A dichromatic framework for balanced trees. *Proc. 19th Ann. Symp. Foundations of Computer Science, IEEE*, 1978.
10. KNUTH, D.E. *The Art of Computer Programming*, vol. 3, Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.
11. KUNG, H.T. *The Structure of Parallel Algorithms*, vol. 19, Advances in Computers. Academic Press, New York, 1980. (Also available as a CMU Computer Science Dep. Tech. Rep., Aug. 1979.)
12. KUNG, H.T., AND PAPADIMITRIOU, C.H. An optimality theory of concurrency control for databases. *Proc. ACM SIGMOD 1979 Int. Conf. Management of Data*, 1979, pp. 116–126.
13. KUNG, H.T., AND SONG, S.W. A parallel garbage collection algorithm and its correctness proof. *Proc. 18th Ann. Symp. Foundations of Computer Science*, 1977, pp. 120–131.
14. LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng. SE-3*, 2 (March 1977), 125–143.
15. LAMPORT, L. Concurrent reading and writing. *Commun. ACM 20*, 11 (Nov. 1977), 806–811.
16. LEHMAN, P.L., AND YAO, S.B. Efficient locking for concurrent operations on B-trees. To appear in *ACM Trans. Database Syst.*
17. MANNA, Z., AND WALDINGER, R. The logic of computer programming. *IEEE Trans. Softw. Eng. SE-4*, 3 (May 1978), 199–229.
18. MILLER, R.E., AND SNYDER, L. Multiple access to B-trees. Proc. Conf. Information and Systems, Johns Hopkins Univ., Baltimore, Md., March 1978, preliminary version.

19. OWICKI, S. Axiomatic proof techniques for parallel program. Ph.D. Dissertation, Dep. Computer Science, Cornell Univ., Ithaca, N.Y., 1975.
20. RIES, D.R., AND STONEBREAKER, M. Effects of locking granularity in a database management system. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 233–246.
21. SAMADI, B. B-trees in a system with multiple users. *Inf. Process. Lett. 5*, 4 (Oct. 1976), 107–112.
22. SWAN, R.J., FULLER, S.H., AND SIEWIOREK, D.P. CM*—A modular multi-microprocessor. *Proc. AFIPS 1977 NCC*, vol. 46, AFIPS Press, Arlington, Va., pp. 637–644.
23. WEDEKIND, H. On the selection of access paths in a data base system. In *Data Base Management*, J. W. Kimbie and K. L. Koffeman, Eds. North-Holland, New York, 1974, pp. 385–397.
24. WULF, W.A., AND BELL, C.G. C.mmp—A multi-mini-processor. *Proc. AFIPS 1972 FJCC*, vol. 41, AFIPS Press, Arlington, Va., pp. 765–777.

Received March 1978; revised September 1979; accepted February 1980