CONCURRENT MAINTENANCE OF A DYNAMIC SEARCH STRUCTURE

Udi Manber

Computer Sciences Technical Report #488

November 1982

CONCURRENT MAINTENANCE OF A DYNAMIC SEARCH STRUCTURE

Udi Manber

Department of Computer Science
University of Wisconsin
1210 W Dayton st´
Madison, Wisconsin 53706

November 1982

This research was supported in part by the National Science Foundation under Grant MCS80-03337.

ABSTRACT

The problem of providing efficient concurrent access for independent processes to a dynamic search structure is the topic of this paper. We develop concurrent algorithms for search, update, insert and delete in a simple variation of binary search trees, called external trees. The algorithm for deletion, which is usually the most difficult operation, is relatively easy in this data structure. The advantages of the data structure and the algorithms are that they are simple, flexible and efficient, so that they can be used as a part in the design of more complicated concurrent algorithms where maintaining a dynamic search structure is necessary. In order to increase the efficiency of the algorithms we introduce maintenance processes that independently reorganize the data structure and relieve the user processes of non-urgent operations. We also discuss questions of transactions in a dynamic environment and replicated copies of the data structure.

1. Introduction

Parallel asynchronous algorithms appear more and more in different applications such as concurrent databases, operating systems and distributed systems. It is therefore important to investigate data structures that support efficient concurrent operations. In this paper we suggest one such basic data structure, called external trees, and present a set of algorithms to manipulate it. The design is intended to be flexible and general so that it can be easily modified and adapted to different applications. The situation we are addressing is as follows. Many independent processes need to maintain a common data structure, for example, in order to update certain global information. The data structure contains items that are ordered according to a unique key. The basic operations a process can perform are find a given item, update an item, insert a new item, and delete an item. Efficient algorithms for these operations are described in section 5. In addition to the basic operations, the processes may wish to perform transactions that consist of sequences of interdependent operations involving several items. We have extended the notion of transactions to include not only read and write, but also insert and delete. In section 8 we discuss the implementation of such extended transactions on external trees.

For simplicity, we assume that the data structure resides in a shared memory. The only requirement for the shared memory is that every process will have access to it. Hence, the algorithms can be easily adapted to applications where physical shared memory is not available. If the processes reside at remote sites then the number of accesses to the shared memory should be minimized. In section 9 we discuss a way to replicate a part of the data structure so that a great part of each operation can be done locally, yet the amount of overhead to maintain the different copies is low.

In order to improve the performance of the algorithms we distinguish between an action that has to be carried out immediately and an action that may be postponed without undermining the integrity of the data structure. We introduce special processes, called maintenance processes, and design the algorithms so that actions that can be postponed are left to the maintenance processes. The maintenance processes operate concurrently with the user processes. The user processes are able to carry out the maintenance jobs but are not required to.

In order to control the concurrency we use locks. The locks are designed in a way that minimizes interference among user processes, and to a second degree, between user and maintenance processes. The maintenance processes are given low priority for acquiring locks. Each basic user operation, update, insert and delete, requires at most one lock. As a result, transactions that involve k items require at most k nodes.

2. Relation to other work

Most other work on concurrent access to dynamic search structures deals with concurrency in balanced multiway trees, in particular B-trees and variations of B-trees [1,9,10,13]. Ellis presented algorithms for concurrent search and insert in AVL trees [3] and 2-3 trees [4]. Kung and Lehman introduced concurrent algorithms to search, insert and delete in binary search trees [8]. These algorithms, and in particular the deletion algorithm, were improved by Manber and Ladner [12], who also introduced the notion of maintenance processes.

The algorithms presented in this paper are simple hybrid of the algorithms in [4] [8] and [12]. They are simpler than the algorithms in [4] since binary search trees (instead of 2-3 trees) are manipulated and no strict balance is maintained. They support more concurrency since only one node is locked by a process per operation. The algorithms in this paper are also simpler than the algorithms in [8,12] since the data is associated with the leaves.

When deletions are allowed to be performed concurrently with searching, a searching process may end up in a deleted part of the tree and, if no precautions are taken, may yield a wrong result. The simple approach to this problem is to lock out searching processes from parts of the data structure that are being modified until the modifications are completed. If the data structure is modified frequently then this approach cannot support high level of concurrency. The approach taken in [8,12] is to allow searching processes to reach deleted nodes, but maintain recovery paths so that they can continue from them correctly. Processes can therefore search down the tree without being blocked even in the presence of concurrent deletions. However, maintaining these recovery paths, which consist of pointers from deleted nodes to their fathers in the tree, is quite complicated.

In this paper we consider both approaches. Since the data is always associated with the leaves, deletions and other reorganizations of the tree occur mostly at the bottom of the tree. As a result, even if processes are locked out during a deletion, they are locked out from only a small part of the tree. In sections 4-6 we follow the first approach. We design locks that minimize the interference between searching and deleting processes. In particular, low priorities are given to the maintenance processes which carry out non-urgent tasks. Algorithms that use the second approach are presented in the Appendix. We simulated algorithms using both approaches [11] and found that under random insertions and deletions there was no significant difference in the average amount of time a process was blocked.

3. The data structure

The model of concurrency we are using is that of a single global memory to which an unbounded number of processes (with their own private memories) have access. For simplicity, we assume that the global memory is divided into units of constant size, which are called _nodes_. Each node can contain several fields. A process can either read or write in one field of one node as an indivisible step. The basic operations of a process are inserting a node, deleting a node, and updating the data associated with a node.

We will attempt to define the algorithms and the data structure in a simple and general manner. For that reason, we avoid dealing with implementation details independent of our algorithms.

The data structure we investigate here is very similar to a regular binary search tree with one notable difference: The data is always associated with the leaves. More precisely, there are two kinds of nodes, _internal_ nodes, and _external_ nodes, which are always leaves in the tree (an internal node may be a leaf temporarily due to an incomplete deletion). Internal nodes are used only for directory purposes, and the data is always associated with the external nodes. We call this data structure an _external tree_.

Each node has a _key_ field, according to which the nodes are ordered, and two pointer fields, pointing to its two sons, and denoted by _left_ and _right_ respectively. An external node also has a _data_ field, which is usually a pointer to another data structure that holds the data associated with the key. An internal node contains an additional flag, called the _garbage_ flag, which indicates that the node has been deleted. Such nodes can eventually be collected by a garbage collector and reused. There are also _lock_ fields which are discussed in detail in the next section.

There is also a list which is used by the maintenance processes. It contains keys of nodes that need a maintenance job. Whenever a maintenance process becomes ready, it takes a job from that list and carries it out. The list can be organized as a queue although it is not necessary since the maintenance jobs are independent and need not be carried out in the same order they were "requested". We discuss the maintenance list in detail in section 6.

An internal node may have the same key as an external node. The rules for searching for a key x are as follows: Branch left if the key of the current (internal) node is > x, otherwise branch right, until an external node or a nil pointer is reached. The search always ends at the bottom of the tree, although it may end in an internal node due to an incomplete deletion (as will be explained shortly), in which case the search is of course unsuccessful.

4. Locking

We use three types of locks: exclusive locks (X-locks), shared locks (S-locks), and edge locks (E-locks). All locks apply only to internal nodes. Access to an external node is controlled by locking its (internal) father. We will return to this point shortly.

Exclusive locks and shared locks are used in the standard way. When a process holds an exclusive lock to a node no other lock can be put on that node. As a result, every other process that needs any access (either read or write) to that node is blocked. Such a lock is needed for deletion to prevent processes from using a deleted node. The shared lock, on the other hand, allows other processes access to that node. Many shared locks can be put on the same node at the same time. The shared lock's purpose is to block processes wishing to acquire an exclusive lock and delete the node. The shared and exclusive locks are similar to RHO-locks and XI-locks [3] (see also [6]).

One way to implement shared and exclusive locks is to associate a counter with every internal node. The value of the counter, if non-negative, equals the number of processes currently holding a shared lock on that node. An exclusive lock is indicated by -1. It turns out to be simpler for the algorithms described in the next section to require that in order to obtain an exclusive lock the corresponding shared lock has to be obtained first. Hence an exclusive lock can be obtained only if the value of the counter equals 1, indicating only one shared lock (its own). The shared and exclusive lock procedures are described below. We assume that they can be carried out as indivisible operations.

```
procedure S-lock(a) ;
while a.lock < 0 do wait ;
a.lock := a.lock + 1 ;

procedure S-unlock(a) ;
a.lock := a.lock -1 ;


procedure X-lock(a) ;
{ We assume that the shared lock
has been obtained first }
while a.lock > 1 do  wait ;
a.lock := -1 ;

procedure X-unlock(a) ;
a.lock := 0 ; {The shared lock is also removed}
```

This is obviously not the only way to implement such locks. Notice that the exclusive lock in this implementation has the lowest priority. Another way to implement the locks could be to disallow any new shared locks once an exclusive lock has been requested. We chose this implementation because the exclusive locks are used only by the maintenance processes and they should have the lowest priority.

The edge lock is used to control access to an external node. It is similar to a binary semaphore. Each possible key value at any time t is associated with a unique internal node; either this node is the father of an external node with the same key or such an external node could be inserted there at time t. Each basic operation involves one key. Before the operation is carried out the (internal) node associated with the key is locked with an edge lock. In order to increase the concurrency we do not lock the whole node but only the appropriate half, hence the name edge lock. Only one edge lock can be put on a given edge at any given time. A process that attempts to lock an already locked edge is blocked until the edge lock is removed. The edge lock is thus associated with the pointer to the external node rather than with the internal node itself. The edge lock is compatible with the shared lock but not with the exclusive lock. For simplicity we also require that the corresponding shared lock be acquired before the edge lock so that any request for an exclusive lock will be blocked and the node will not be deleted while the edge lock is held.

The decision to associate locks with internal nodes rather than external nodes where the data reside proved to be very beneficial. This way all the overhead resulting from the concurrency is restricted to the internal (directory) nodes. As a result, the external nodes are in a sense independent of the data structure. The data associated with the external nodes may also be a part of other data structures at the same time (using additional locking). This independence simplifies the design of the interfaces between the data structures, and possibly between the different concurrency control protocols. It also turns out that the algorithms become significantly simpler.

The procedure for locking a node in a given "direction" with an edge lock is given below. Again, it is assumed to be an indivisible operation. Each internal node has two Boolean variables denoted by left-lock and right-lock.

```
procedure E-lock(a,x) ;
if a.key > x then
  { Checking this condition need not be a part
    of the indivisible step since both a.key and
    x cannot be changed. }
 begin
   while a.left-lock do wait ;
   a.left-lock := true ;
 end else
 begin
   while a.right-lock do wait ;
   a.right-lock := true ;
 end ;

procedure E-unlock(a,x) ;
if a.key > x then a.left-lock := false
 else a.right-lock := false ;
```

5. Basic user operations

In this section we present procedures for the basic dynamic operations. Since the data is associated with the leaves a deletion of an external node is quite simple. However, some reorganization of the tree is still needed, and since it can be postponed, we leave it to the maintenance processes. We assume that the root of the tree is a special purpose node that can never be deleted. An informal proof of correctness is given in section 7.

For simplicity we use the notation <u>b.direction(x)</u> to denote b.left if x < b.key, and b.right otherwise. We assume that an external node can be easily identified as such. Function strong-search(x) returns the (internal) node associated with the key x and locks its appropriate pointer with an edge lock. Nodes encountered during the search are locked with a shared lock on the way down to make sure that they are not being deleted. A node is unlocked only after its son is locked and found to be non-deleted (the garbage flag would indicate if the node had been deleted). In this way, even if a node encountered during the search had been deleted in the middle of the search, there is no need to backtrack since its father is still locked (hence it has not been deleted), and known to the process. In the Appendix we show how to avoid using the exclusive and shared locks altogether by maintaining a father pointer so that a process that finds itself in a garbage node can backtrack. An edge-lock, which does not block searching processes, can be used to avoid conflicts among deleting processes.

```
function strong-search(x) ;
begin
b := root ;
S-lock(b) ;
L : a := b.direction(x) ;
```

```
while a ≠ nil and a is not external do
 begin
  {at this point b is guaranteed not to be garbage}
     S-lock(a) ;
     if not a.garbage then
      begin
        S-unlock(b) ;
        b := a ;
      end ;
   {If a is garbage we find the new son of b.
    A lock on a garbage node is ignored. }
     a := b.direction(x) ;
 end ;
E-lock(b,x) ;
a := b.direction(x) ;
if a ≠ nil and a is not external
       { in case another internal node has been
         inserted before the edge lock was put on }
   then [ E-unlock(b,x) ; go to L ] ;
 else return b ;  { b remains locked }
end ;
```

Procedure update is straightforward.

```
procedure update(x,f) ;
begin
b := strong-search(x) ;
a := b.direction(x) ;
if a = nil or a.key ≠ x
  then report "node(x) is not in the tree"
   else begin
          b.direction(x).data := f(b.direction(x).data) ;
          {update using the function f}
        end ;
E-unlock(b,x) ; S-unlock(b) ;
end ;
```

Procedure insert first finds the right place to insert using strong-search which locks the appropriate pointer (if the key is not in the tree strong-search will return the internal node to which this key should be inserted). Then, if the pointer turns out to be nil, which is the result of a previous deletion of an external node, a new external node is created and a pointer to it simply replaces the nil pointer. (A deletion of an external node is followed by a deletion of the corresponding internal node. This last deletion is carried out by the maintenance processes and thus can be delayed as will be shown in section 6.) If the node associated with the key points to an existing external node then a new internal node is created with the new and existing external nodes as its sons.

```
procedure insert(x) ;
begin
b := strong-search(x) ;
a := b.direction(x) ;
if a ≠ nil and a.key = x then
  report "node(x) is already in the tree" ;
else
 begin
  create a new external node newex with key x ;
  if a = nil then b.direction(x) := newex
  else
  begin
    create a new internal node newin with key (a.key+x)/2 ;
    if x < a.key then
     begin
       newin.left := newex ;
       newin.right := a ;
     end else
     begin
       newin.right := newex ;
       newin.left := a ;
     end ;
    b.direction(x) := newin ;
  end ;
 end ;
E-unlock(b,x) ; S-unlock(b) ;
end ;
```

A deletion is quite simple since only leaves are deleted. After an external node has been deleted its (internal) father is left with only one son, which implies that it is not needed any more and should be deleted. We leave the task of deleting internal nodes to the maintenance processes. The key of the internal node is put in a maintenance list called a <u>delete-list</u>. When a maintenance process becomes ready it takes a key from that list and carries out the deletion (see section 6).

```
procedure delete-external(x) ;
begin
b := strong-search(x) ;
a := b.direction(x) ;
if a = nil or a.key ≠ x then
 report "node(x) is not in the tree"
  else begin
         b.direction(x) := nil ;
         put b.key in the delete-list ;
       end ;
E-unlock(b,x) ; S-unlock(b) ;
end ;
```

6. Algorithms for the maintenance processes

The only task left for the maintenance processes is deletion of internal nodes. First, the internal node with the given key is found using a procedure similar to strong-search. The keys are taken from the delete-list. An alternative way is to let the maintenance processes constantly traverse the tree and look for internal nodes that can be deleted. Once a candidate node is found, it is locked with an exclusive lock, and since it has at most one son it can be easily deleted. The reason an exclusive lock is needed is to prevent processes from being "stuck" in a garbage node. One can avoid using exclusive locks by maintaining "father" pointers so that a process that finds itself in a garbage node can backtrack (see Appendix). Notice that such a node may have no sons, in which case its father can also be deleted. A maintenance process cannot delete the father without having a shared lock on the "grandfather", hence it puts the father key in the delete-list.

```
procedure delete-internal(y) ;
begin
b := root ;
S-lock(b) ;
L : a := b.direction(x) ;
while a ≠ nil and a.key ≠ y and a is not external do
 begin
     S-lock(a) ;
     if not a.garbage then
      begin
        S-unlock(b) ;
        b := a ;
      end ;
     a := b.direction(x) ;
 end ;
if a = nil or a is external then [S-unlock(b) ; terminate ] ;
S-lock(a) ;
if a.garbage then go to L ;
X-lock(a) ;
if b.left = nil or b.right = nil then
 begin
    if a.left = nil then b.direction(y) := a.right
     else b.direction(y) := a.left ;
    a.garbage := true ;
    if b.direction(y) = nil then
      put b.key in the delete-list ;
 end ;
X-unlock(a) ;
S-unlock(b) ;
end ;
```

7. Informal proof of correctness

We do not restrict in any way the order in which the processes access the shared memory. As a result, we cannot guarantee termination of a process. A process may be blocked forever from an external node by other processes that always get the corresponding edge lock before it. One can overcome this problem in various ways, for example, by maintaining queues for acquiring locks. The integrity of our system does not depend on any such queuing mechanism. Notice that since the locks are acquired on the way down the tree, no deadlock can occur so some progress is always being made. In this section we prove the integrity of the concurrent search structure. We show that at any moment each process sees the data structure as if it was its own consistent search structure, and its only interaction with other processes is when it is blocked and needs to wait for a lock. The notion of integrity has to be precisely defined. We define a set of correctness assertions that imply the integrity and prove them correct simultaneously by induction on time.

The data structure consists, at any moment t, of a binary tree; this fact is the main assertion.

A1. Every node v, at any moment t, is a root of two (possibly empty) subtrees. The left subtree contains only nodes with keys less than v.key, and the right subtree contains only nodes with keys greater than or equal to v.key. An external node has no sons.

Assertion A1 implies that the data structure is consistent at any given moment. We also have to show that the algorithms are correct. First we have to define what it means for a strong-search to return the "correct" (internal) node.

Each possible key, at any moment t, is associated with a unique (internal) node and there is a unique path from the root to this node. By assertion A1, the data structure is a valid search tree at any moment, hence the definition of this association is simple. The reason we can rely on the assertions for the definitions is because we prove the assertions simultaneously by induction. Thus, if the definitions are meaningful, by induction, at time t, and we prove that the assertions are true for t+1, then the definitions are meaningful at t+1. For each key x, we define $path_t(x)$ to be the path, starting from the root and ending at a leaf, in which the next node is chosen according to the (regular) search at time t. In other words, we freeze the tree at time t, and let $path_t(x)$ be the path a strong-search will follow from the root to a leaf without being blocked. The last internal node in $path_t(x)$ is the node associated with x at time t; this node is denoted by $node_t(x)$.

A2. A non-garbage node a, which is locked with an S-lock in function strong-search(x) at time t, is contained in

$\text{path}_t(x)$.

Function strong-search is designed so that a pointer to at least one such non-garbage node is always maintained, hence it is always on the "right track". In addition, we have to argue that when strong-search finally returns a node, it is the right one.

A3. If strong-search(x) returns node b at time t, then b = $\text{node}_t(x)$.

We prove assertions A1-A3 by considering every step of every procedure that modifies the tree. We assume that initially the data structure is consistent, hence the assertions are satisfied. Assuming that A1-A3 are true at time t, we will show that for any next step, taken by any procedure at time t+1 (we assume that every step, taken by any process, takes one unit of time), A1-A3 remain true at time t+1.

It is easy to verify the effects of the different locks. A shared lock guarantees that the node will not be deleted. An exclusive lock prohibits any access to the node, and an edge lock serves as a binary semaphore on the corresponding edge.

First we show that assertion A3 is implied by A1-A2 at any time t. Let b be the node returned by strong-search(x) at time t. By A1 $\text{path}_t(x)$ is well defined. By A2, b is on $\text{path}_t(x)$. If x < b.key then $\text{path}_t(x)$ should "turn" left at

b. However, b.left-lock had been acquired by the strong-search before b.left (a) was found to be an external node or nil, and as long as b.left-lock is held the status of b.left cannot be changed. Hence b is the last internal node on $path_t(x)$ which implies that b = $node_t(x)$. The case of $x \geq$ b.key is similar.

We have to consider the three procedures, insert, delete-external and delete-internal. The correctness of procedure update follows directly from the correctness of strong-search which is implied by A3.

<u>procedure</u> <u>insert</u>(<u>x</u>). By A3, b = $node_t(x)$ and a is either an external node or nil. Without loss of generality we can assume that x < b.key. Since the left edge lock in b is held by the insert procedure, no other process can change b.left. b.left is changed in procedure insert only in the last step (except for unlocking), hence there is no interference from other processes during the insert. A1 is satisfied after the insertion since b was on $path_t(x)$, only a son of b is modified, b was found at one time (during strong-search) to be an internal node and a node cannot change from internal to external (or vice versa). A2 is satisfied since the only possible change in any $path_{t+1}(y)$ (as compared to $path_t(y)$) is to make it longer and end in newin.

The new internal node, newin, is created only when $x \neq$

a.key. Since b is associated at time t with both x and a.key, newin.key (which equals (x+a.key)/2) is a unique key. Hence, although an internal node's key can be equal to some external node's key, no two internal nodes can have the same key.

Procedure delete-external. By A3, b = $\text{node}_t(x)$, hence if there is an external node with key x it is a son of b. If b.direction(x) = nil then nothing is done. Otherwise, by A1, since b.direction(x) is an external node it has no sons, hence setting it to nil does not change any path or any association. Hence A2-A3 are still satisfied. A1 is clearly satisfied.

Procedure delete-internal. The search used in this procedure is very similar to strong-search. The main difference is that we stop whenever we find an internal node with the given key rather than always go all the way to the bottom. The other difference is that the more powerful exclusive lock is used. We have already shown that the keys of the internal nodes are unique. Hence, we can use the same arguments as in the strong-search case (defining $\text{internal-node}_t(x)$ for example) to show the correctness of this search. Once the node is found, it is locked with an exclusive lock and checked to have only one son. The deletion is then straightforward. The father is locked with a shared lock so that it cannot be deleted. Since the deleted node is locked with an exclusive lock no other process can access it at that time, which implies A1, A2 and A3. The keys associated with the deleted node change their association to either its predecessor or its successor.

8. Generalized transactions

In many applications a user wishes to make a sequence of updates to several nodes such that the whole sequence will be regarded as one atomic operation. Such a sequence is usually called a _transaction_[5,7]. In a concurrent environment a transaction cannot be physically carried out in one indivisible step, hence some rules have to be imposed so that concurrency can be allowed. The problem of designing a concurrency control mechanism to support the concurrent execution of transactions received much attention lately. Such a control usually guarantees _serializability:_ namely, the outcome of the concurrent processing is always the same as an outcome of a serial processing (in some order) where there is no concurrency.

The association of each key in our data structure with a unique node and the fact that the dynamic operations are local to the node with the corresponding key allow us to extend the scope of the transactions to include not only updates, but also deletions and insertions. For example: A transaction may now be of the form "replace a given node with another node", i.e. "delete node x and insert node y". If the tree holds a file directory such that the keys are file names, then the simple example above corresponds to changing the name of a file. Since each key can be controlled by one lock (the edge lock in this case), a transaction consisting of k keys can be carried out with at most k locks. The two-phase locking protocol [5] can be easily extended to support such generalized transactions. A similar protocol which supports generalized transactions on regular binary search trees is described in [11,12].

Since these generalized transactions are powerful and yet conceptually simple they can be used as building blocks in the design of general concurrent algorithms. The design would be divided into two steps. First the transactions (namely the simulated atomic operations) are defined and the problem is solved using them as atomic operations, and second a concurrency control protocol is designed to simulate the transactions efficiently.

9. Replicated copies of the data structure

When distributed computation is involved, namely when the users reside in remote sites and communication with the shared memory is expensive, replicated copies of the shared memory are used. The same shared memory model can apply to this case, even though the memory is not physically shared. The external tree data structure and the algorithms described in the previous sections can be adapted to the case of replicated copies. Moreover, we can replicate only parts of the data structure in the following way. We consider the tree to be divided into two parts. The top part, which consists of a subtree containing the root, and the bottom part, which consists of the rest. The division is across a horizontal (but not necessarily straight) line. We further assume that the top part and the roots of the subtrees of the bottom part contain only internal nodes that are never deleted. In this case the top part can be replicated and distributed among the users with no concurrency problems. Moreover, each user can rebalance the top part to optimize it according to its own use. The main feature of having each key associated with a unique node is still preserved. The decision where to divide the tree depends on the application. The condition that a node in the top part will never be deleted is not necessary. A process that wishes to delete a node in the top part broadcasts this wish to all the other users; if all agree, they acknowledge, and each carries out the deletion in its own copy. A survey of algorithms for updating distributed databases is given in [2]. Such a broadcast algorithm is generaly more expensive than a simple access to the shared memory, a fact that leads to a tradeoff: If the top part is made larger then more local accesses can be used, but also more broadcasts may be required.

Due to the flexibility of the data structure, one can also replicate the bottom part (which is being physically shared) independently of the top part, for example, for reliability purposes. If the number of copies of the bottom part is small (2 or 3 copies are sufficient for reliability purposes), then the following modification to the algorithms can be made. In order to carry out any operation (e.g. insert, delete), a user process will be required to acquire the corresponding edge locks for all the copies. Each external node will have pointers to all its copies, as well as a pointer to its (internal) father. These pointers can be used by the user process, which finds the node in one copy, to find the corresponding nodes in the other copies. If a process wishes only to read the data associated with an external node then it is sufficient to lock (and access) only one copy. The modifications to the algorithms are straightforward. As for the maintenance operations, it is also possible to have the internal nodes of different copies of the bottom part connected through such pointers. However, maintenance processes need not be very efficient, thus they can find the different copies by searching through the tree.

10. Conclusions and further research

We presented a design of a simple data structure and relatively simple concurrent algorithms to manipulate it. The algorithms support high degree of concurrency, yet they are not too complicated. As a result, they can be used as a basic search structure in the design of concurrent algorithms. The data structure and the algorithms can also be used in conjunction with a concurrency control protocol to support transactions containing dynamic operations (insert and delete) as well as static operations (read and update). The data structure can be adapted to a distributed environment in an efficient way.

We have not discussed the question of balancing the tree in this paper. We are currently working on efficient balancing schemes that may be used with external trees. We are also working on implementing other operations such as simple range queries.

Acknowledgement

I would like to thank Richard Ladner for many helpful discussions, and Raphael Finkel for many comments that improved the clarity of the paper.

REFERENCES

[1] R. Bayer and M. Schkolnick, "Concurrency of Operations on B-trees", Acta Informatica Vol 9, pp. 1-22, 1977.

[2] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", ACM Computing Surveys, Vol 13, pp. 185-222, June 1981.

[3] C. Ellis, "Concurrent Search and Insertion in AVL Trees", IEEE Transactions on Computers, Vol C-29, pp. 811-817, September 1980.

[4] C. Ellis, "Concurrent Search and Insertion in 2-3 Trees", Acta Informatica, Vol 14, pp. 63-86, 1980.

[5] K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol 19, pp. 624-633, November 1976.

[6] J.N. Gray, "Notes on Database Operating Systems", in Operating Systems: An Advance Course, Springer-Verlag, pp. 393-481, 1978.

[7] J.N. Gray, "A Transaction Model", in Automata, Languages and Programming, Lecture Notes in Computer Science, Volume 80, Springer Verlag, 1980.

[8] H.T. Kung and P.L. Lehman, "Concurrent Manipulation of Binary Search Trees", ACM Transactions On Database Systems, Vol 5, pp. 354-382, September 1980.

[9] Y.S. Kwong and D. Wood, "A New Method for Concurrency in B-trees", IEEE Transactions on Software Engineering, Vol SE-8, pp. 211-222, May 1982.

[10] P.L. Lehman and S.B. Yao, "Efficient Locking for Concurrent Operations on B-Trees", ACM Transactions On Database Systems, Vol 6, pp. 650-670, December 1981.

[11] U. Manber, "Concurrency Control for Dynamic Data Structures and Fault Tolerance", Ph.D. Thesis, University of Washington, July 1982.

[12] U. Manber and R.E. Ladner, "Concurrency Control in a Dynamic Search Structure", in ACM Symposium on Principles of Database Systems, Los Angeles, pp. 268-282, March 1982.

[13] B. Samadi, "B-Trees in a System With Multiple Users", Information Processing Letters, Vol 5, pp.107-112, October 1976.

Appendix I

In this Appendix we describe algorithms for strong-search and delete-internal that do not use exclusive or shared locks. The algorithms for update, insert and delete-external are unchanged. We assume the existence of an additional pointer field, called father in each internal node. Initially the father pointer is set to nil. When a process, which is looking for x, finds an internal node that seems to be the node associated with x, it locks the node and then checks to see whether the node has been deleted. If the node has been deleted (i.e. the garbage flag is on) then at the time of the deletion its father pointer was set, and the father was on the path to the new node associated with x. Hence, it was correct at that time to continue from the father. The father may be deleted later as well, in which case the same procedure is repeated. When an internal node is deleted both its edge locks are acquired first. As a result, holding one edge lock of a node is sufficient to prevent that node from being deleted.

The garbage collection is more complicated in this case. One cannot reuse a deleted node as long as there may be some processes still looking at it. One possible solution is as follows [8]. When a node is deleted a pointer to it is entered in a queue Q. Such node can be reused after all the processes that were active at the time of deletion terminate (new processes cannot reach the node). The garbage collection is done is two phases. When a garbage collection process becomes ready it copies Q to another queue R (Q has to be locked while it is copied to prevent insertions), and resets Q to nil. (If linked lists are used to represent the queues then this step can be done very efficiently.) The garbage collection process takes note of all the processes active at the time R is created. When all these processes terminate R can be appended to the available list.

```
function strong-search(x) ;
begin
b := root ;
L : a := b.direction(x) ;
while a ≠ nil and a is not external do
 begin
   b := a ;
   a := b.direction(x) ;
 end ;
E-lock(b,x) ;
if b.garbage then
 begin
   E-unlock(b,x) ;
   b := b.father ;
   go to L ;
 end ;
a := b.direction(x) ;
if a ≠ nil and a is not external
       { in case another internal node has been
         inserted before the edge lock was put on }
   then begin
          E-unlock(b,x) ;
          go to L ;
        end
    else return b ;  { b remains locked }
end ;
```

```
procedure delete-internal(y) ;
begin
b := root ;
L : a := b.direction(x) ;
while a ≠ nil and a.key ≠ y and a is not external do
 begin
   b := a ;
   a := b.direction(x) ;
 end ;
E-lock(b,y) ;
if b.garbage then
 begin
   E-unlock(b,y) ;
   b := b.father ;
   go to L ;
 end ;
if a ≠ nil and a is not external then
 begin
   E-lock(a,a.key-1) ; {left-lock}
  { The left edge lock is acquired to make sure a is
   not deleted (the right lock will do as well).}
   if a.garbage then
    begin
      E-unlock(b,y) ; E-unlock(a,a.key-1) ;
      go to L ;
    end ;
   E-lock(a,a.key+1) ; {right-lock}
  { Both edge locks have to be acquired to make sure
   no other process uses a (for example insert to a).}
   if b.left = nil or b.right = nil then
    {b can indeed be deleted}
    begin
      if a.left = nil then b.direction(y) := a.right
       else b.direction(y) := a.left ;
      a.garbage := true ;
      a.father := b ;
      if b.direction(y) = nil then
        put b.key in the delete-list ;
    end ;
   E-unlock(a,a.key-1) ; E-unlock(a,a.key+1) ; E-unlock(b,y) ;
  end ;
end ;
```