# Amortization Results for Chromatic Search Trees, with an Application to Priority Queues

Joan Boyar, Rolf Fagerberg, and Kim S. Larsen\*

Department of Mathematics and Computer Science, Odense University, Campusvej 55, DK-5230 Odense M, Denmark

Received March 10, 1995; revised April 4, 1997

The intention in designing data structures with relaxed balance, such as chromatic search trees, is to facilitate fast updating on shared-memory asynchronous parallel architectures. To obtain this, the updating and rebalancing have been uncoupled, so extensive locking in connection with updates is avoided. In this paper, we prove that only an amortized constant amount of rebalancing is necessary after an update in a chromatic search tree. We also prove that the amount of rebalancing done at any particular level decreases exponentially, going from the leaves toward the root. These results imply that, in principle, a linear number of processes can access the tree simultaneously. We have included one interesting application of chromatic trees. Based on these trees, a priority queue with possibilities for a greater degree of parallelism than previous proposals can be implemented. © 1997 Academic Press

### 1. INTRODUCTION

A chromatic search tree [21, 8] is a binary search tree for shared-memory asynchronous parallel architectures. It was introduced with the aim of allowing processes to lock nodes, in order to avoid inconsistencies from updates and rebalancing operations, without decreasing the degree of parallelism too much. The means for obtaining this was a new balance criteria, referred to as relaxed balance, along with new uncoupled operations for updating and rebalancing.

The rebalancing is taken care of by background processes in small independent steps; the processes do only a constant amount of work before they release locks and move on to another problem. This means that the traditional exclusive locking of whole paths or step-wise exclusive locking down paths, which would limit the amount of parallelism possible to the height of the tree, does not take place. Another advantage of the uncoupling of the rebalancing from the updating is that all or parts of the rebalancing can be postponed until after peak working hours. The disadvantage, of course, is that the tree can become very unbalanced if there are not enough background processes doing the rebalancing.

Since the rebalancing is done in small independent steps, which can be interspersed with other updating and rebalancing operations, an actual proof of complexity is not straightforward, and the original proposal in [21] did not contain

0022-0000/97 \$25.00 Copyright © 1997 by Academic Press All rights of reproduction in any form reserved. any such proof. In [8], the proposal of [21] was analyzed, and it was proven that some updating could give rise to a super-logarithmic number of rebalancing operations. A modified set of rebalancing operations was proposed, and it was proven that the new set of rebalancing operations give rise to at most  $\lfloor \log_2(n+i) \rfloor$  rebalancing operations per insertion and at most  $\lfloor \log_2(n+i) \rfloor - 1$  rebalancing operations per deletion, if *i* insertions are performed on a tree which initially contains *n* leaves. Furthermore, most of these rebalancing operations do no restructuring at all; they simply move weights around. The number of operations which actually change the structure of the tree is at most one per update. Compared to [21], a small constant number of extra locks per rebalancing operation are necessary in [8].

Having obtained logarithmic worst-case bounds, the next result to hope and search for when dealing with trees is an amortized constant number of rebalancing operations. It turns out that the proposal from [8] has these properties, although in order to get the best possible constant, one operation should be modified slightly. In this paper, we prove that, starting with an empty tree, *i* insertions and *d* deletions give rise to at most  $\frac{5}{2}i + \frac{3}{2}d + \frac{1}{2}$  rebalancing operations. We also show that the number of rebalancing operations which can occur at weighted height h is at most  $3i/2^{h-1}$ . The latter result is especially important in a parallel environment, since many of the rebalancing operations require exclusive locks on the nodes they are accessing. The higher up in the tree a lock occurs, the larger the subtree which cannot be accessed by other operations. Our amortization results imply that, in principle,  $\Theta(n)$  processors can simultaneously access the tree, since searching does not require exclusive locking. The results are obtained assuming, as is standard in amortized analyses, that the structure is initially empty, although we also have results for the case where the structure is initially nonempty.

In the last part of the paper, we discuss one particularly interesting application of chromatic trees in greater detail. From the sequential case, it is known that in some cases search tree implementations of priority queues give better performance than heap implementations [12]. In our setting of a shared-memory architecture, it turns out that a variation

<sup>\*</sup> E-mail: {joan,rolf,kslarsen}@imada.ou.dk.

of a chromatic search tree, used as a priority queue, allows for a greater degree of parallelism than in previous proposals for priority queues. The priority queue is suited for branch and bound and similar applications.

Our results, some of which appeared in [7], build upon a long sequence of work on search trees that deals with the idea of uncoupling the updating from the rebalancing. At first, this was only done partially, but later the separation became complete. The more important results introducing the idea of separating updating and rebalancing and partially obtaining this include [9, 14–16, 19, 24–26]. In [21, 22], rebalancing is separated entirely from the updating. States of unbalance are merely registered, and background processes deal with these problems of unbalance in parallel with searches and updates. This type of search tree is referred to as a search tree with relaxed balance.

Several complexity results have been obtained for relaxed search trees. The analysis of the behavior of a relaxed version of AVL trees [1] introduced in [22] was given in [17]. The structure chromatic search tree of interest in this paper is a relaxed version of red-black trees [3, 9]. As already mentioned, this structure was introduced in [21] and analyzed in [8]. A relaxed version of (a, b)-trees [11] was analyzed in [18]. Both of the common B-trees [4], 2-3 trees [2, 10], and 2-3-4 trees [9] have relaxed versions, the properties of which are also discussed in [18]. The first relaxed version of a B-tree is from [22]. Some implementation experience with a modified version of [8] can be found in [20].

## 2. CHROMATIC SEARCH TREES

In this section, we describe chromatic search trees, noting a couple of minor changes from earlier definitions [21, 8]. Chromatic trees are *leaf-oriented* binary search trees, so the keys are stored in the leaves and the internal nodes only contain *routers* which guide the search through the tree. The router stored in a node v is greater than or equal to any key in the left subtree and less than any key in the right subtree. The routers are not necessarily keys which are present in the tree, since the node containing the corresponding key may have been deleted. The tree is a *full* binary tree, so each node has either zero or two children.

Since chromatic trees are a relaxation of red-black trees, the nodes have weights, which are restricted to being only zero or one in a red-black tree, but can be greater than one in a chromatic tree. This relaxation means that the data structure can simply be left as it is after an update; the rebalancing is taken care of by other processes.

In earlier definitions of chromatic trees, the weights were associated with the edges, but here we place them on the nodes to conform with standard usage. The results in this paper and [8] hold in either formulation. Here, each node v in the tree has an associated nonnegative integer weight

w(v). If w(v) = 0, we call the node *red*; if w(v) = 1, we say the node is *black*; and if w(v) > 1, we say the node is *overweighted*. The *weight* of a path is the sum of the weights of its nodes, and the *weighted level* of a node is the weight of the path from the root to that node.

For completeness, we give definitions for both red-black trees and chromatic trees.

DEFINITION 1. A *red–black tree* is a full binary search tree *T* with the following balance conditions:

B1: The leaves of *T* are black.

B2: All leaves of T have the same weighted level.

B3: No path from T's root to a leaf contains two consecutive red nodes.

B4: T has only red and black nodes.

Chromatic trees are defined similarly to red-black trees, except that the balance conditions are relaxed.

DEFINITION 2. A *chromatic tree* is a full binary search tree *T* with the following conditions:

C1: The leaves of T are not red.

C2: All leaves of T have the same weighted level.

The insertion and deletion operations are depicted in Appendix A, along with the rebalancing operations. Squares denote leaves, circles denote general nodes (either internal nodes or leaves), and the labels denote weights. For the sake of intelligibility, the subtrees of internal nodes are not shown. However, note that for all operations there are the same number of these subtrees before and after the operation. The in-order ordering of the tree enforces a one-to-one correspondence between these subtrees. For instance, in the operation labelled (rb2), there are five subtrees not shown. These are simply attached again in the same order (but to new nodes) after the operation. Also, in all rebalancing operations there are the same number of nodes before and after the operation, so updating the routers after a rebalancing operation is simple: the same routers can be used again, and their distribution among the nodes is determined by the in-order ordering. The updating of the routers during updates is as follows: For an insertion, the router in the new internal node is given the value of the key in its left child. For a deletion, the router in the sibling of the node deleted becomes the router of the single node remaining after the operation.

It is easy to verify that the operations alter chromatic trees in a well-defined manner; that is, if the tree is a chromatic tree before an operation, it is still a chromatic tree after the operation.

Note that we do not list symmetric cases. We also omit showing separately the special cases which apply at the root. Whenever an operation changes the weight of the root, as part of the operation, the weight of the root is set to one (thus, the weight of the root is always one).

The order in which operations are applied is unrestricted. All results stated in this paper are independent of the order in which operations are carried out. An operation can be applied in any circumstances that match the lefthand side depicted, except for the blacking operation, which has one more restriction: it can only be applied if at least one of the two lower nodes has a child of weight zero.

Note that the third weight-decreasing operation from [8] has been modified slightly in order to obtain the results in this paper. Instead of the operation given in Fig. 1, we use the similar operation given in Fig. 2. The results of [8] still hold with this modification.

The proper location for an insertion or deletion is found by searching, as in any binary search tree. Thus, an insertion will take logarithmic time if the structure is still reasonably close to being a red-black tree. The actual update, however, only requires constant time.

The rebalancing operations are defined so that if, at any point, no further updates occur, but rebalancing occurs as long as any operation is applicable, then the chromatic tree will eventually become red-black. This was proven in [8], and the same proof holds with the one modified operation.

The rebalancing operations are employed as follows. If a chromatic tree is not a red-black tree, it must have either two consecutive red nodes on some root-to-leaf path (a redred conflict) or a node with weight greater than one (an overweighted node). These problems are easily identified when they are created. When a problem is identified, a pointer to one of the nodes involved is placed in a problem queue for the rebalancing processes. In the case of a red-red conflict, the pointer should be to the lower node involved, and in the case of overweight, the pointer should be to the actual node involved. This ensures that pointers to problems never get separated from these problems, even though other rebalancing operations may change the structure of the tree. (We do not address the problem of maintaining a "problem queue" in this paper, although we anticipate that each rebalancing processor would have its own dequeue, so it could add or remove problems from one end, while the other is used for load balancing. One could possibly use a method similar to that in [6].) If a rebalancing process creates another redred conflict or another overweighted node, it can also easily recognize this and put a pointer to it in the problem queue. Clearly, a rebalancing process handling a red-red conflict



FIG. 1. The old operation.



FIG. 2. The new operation.

will need to find the parent and grandparent of the node it finds through the problem pointer. For this purpose, it is necessary for each node to have a parent pointer, in addition to left and right child pointers.

The red-red conflicts result in blacking operations or redbalancing operations. The latter are referred to as (rb1) and (rb2) in Appendix A. Overweighted nodes result in push operations or weight decreasing operations. The latter are referred to as (w1) through (w7) in Appendix A.

A locking scheme is necessary here in order to prevent inconsistencies which might otherwise occur because asynchronous parallel processors share this common data structure. We present one possible locking scheme in Appendix B to show that this is possible.

#### **3. COMPLEXITY**

In [8], it was proven that if one begins with any redblack tree with *n* leaves and then does *i* insertions, a number of deletions, and some rebalancing operations, there will be at most  $\lfloor \log_2(n+i) \rfloor$  rebalancing operations per insertion and at most  $\lfloor \log_2(n+i) \rfloor - 1$  rebalancing operations per deletion. Furthermore, the number of operations which actually change the structure of the tree is at most one per update.

In this section, we give new bounds on the worst case number of rebalancing operations necessary in order to restore balance after updates. Our main result is that, starting with an empty chromatic tree, a very small constant number of rebalancing operations per update are needed to keep the structure balanced. We also define the weighted height of an operation, which is closely related to the actual height at which it is taking place in the structure, and show that, in chromatic trees, the number of rebalancing operations of weighted height h is an exponentially decreasing function of h. In the last part of the section, we present various extensions to these results. Results similar to the ones presented in this section, but for (2, 4)-trees, which are equivalent to standard red-black trees, can be found in [11], which has given inspiration for the proofs presented here.

In chromatic trees, all paths starting at the same node and ending at a leaf will have the same weight. We define the *weighted height* of a node to be the weight of any path from that node to a leaf. The *weighted height of an operation* is most straightforwardly defined as the weighted height of the top node of the operation. However, for technical reasons, we need the following slightly different definition.

DEFINITION 3. The *weighted height of an operation* in a chromatic tree is the weighted height, before the operation occurs, of the children of the top node of the operation, except for insertions, where it is the weighted height of the single leaf present before the operation.

Thus, the weighted height of a weight decreasing or push operation is at least two, and the rest of the operations have weighted height at least one. Note that if the chromatic tree is close to being red-black, the weighted height is closely related to the actual height at which the operation takes place, since in a red-black tree, the actual height of a node is always between one and two times its weighted height. Therefore, if the necessary rebalancing operations are carried out concurrently with the updates, our definition is a good measure of the actual height of the operation. We can now state our main theorem.

THEOREM 4. If i > 0 insertions and d deletions are performed on an initially empty chromatic tree, then at most 3i + d - 2 rebalancing operations can occur. Furthermore, the number of rebalancing operations of weighted height h that can occur is bounded by  $3i/2^{h-1}$  for  $h \ge 2$  and by i for h = 1.

The theorem will follow from the lemmas below. These all apply to the situation where *i* insertions, *d* deletions, and some rebalancing operations have taken place on an initially empty chromatic tree. This will not be repeated in the statements of the lemmas. We let  $d_h$ ,  $b_h$ ,  $r_h$ ,  $w1_h$ , ...,  $w7_h$ , and  $p_h$  denote the number of deletions, blacking operations, red-balancing operations, weight decreasing operations 1, ..., 7, and push operations, respectively, of weighted height *h* that have occurred. To simplify the statements of the lemmas, we also define  $b_0$ ,  $w1_1$ ,  $w2_1$ ,  $w7_1$ , and  $p_1$  (these would otherwise be undefined) as  $b_0 = i$  and  $w1_1 = w2_1 = w7_1 = p_1 = 0$ .

LEMMA 5. For  $h \ge 1$ ,  $b_h \le i/2^h$ .

*Proof.* Let *T* denote a chromatic tree. Call the edges above red nodes *red edges*. By the *red connected components* in *T*, we mean the connected components of the subgraph induced by the red edges. All red nodes in such a component have the same weighted height. By the *height of a red connected component*, we mean the weighted height of any of its red nodes. We let  $\mathscr{C}_h(T)$  denote the set of red connected components of height *h* in *T* and define a sequence,  $\Phi_h(T)$ ,  $h \in \{1, 2, 3, ...\}$ , of potential functions on *T*, by

$$\Phi_h(T) = \sum_{r \in \mathscr{C}_h(T)} (|r| - 1).$$

Here, |r| means the number of red nodes in *r*. An operation of weighted height *k* can change  $\Phi_h(T)$  for some *h*'s. We denote this change by  $\Delta \Phi_h(T)$  and state some facts concerning this:

• For an *insertion* of any weighted height,  $\Delta \Phi_1(T) \leq 1$ and  $\Delta \Phi_h(T) = 0$ , for  $h \neq 1$ .

• For a *blacking operation* of weighted height k,  $\Delta \Phi_k(T) \leq -2$ ,  $\Delta \Phi_{k+1}(T) \leq 1$ , and otherwise  $\Delta \Phi_h(T) = 0$ , for  $h \notin \{k, k+1\}$ .

• For any other operation of any weighted height,  $\Delta \Phi_h(T) \leq 0$ , for all *h*.

These facts can be verified by a tedious inspection of the operations in Appendix A (for the first fact concerning blacking operations, note that in order for a blacking operation to be applied, we require that at least one of the red nodes must be the parent of another red node). For an empty chromatic tree,  $T_0$ ,  $\Phi_h(T_0) = 0$  for all h, so by the facts above,  $\Phi_1(T) \leq i - 2b_1$  and  $\Phi_h(T) \leq b_{h-1} - 2b_h$  for  $h \geq 2$ . As  $\Phi_h(T)$  is never negative, this implies that  $b_1 \leq i/2$ and  $b_h \leq b_{h-1}/2$  for  $h \geq 2$ , from which the lemma follows.

LEMMA 6. For  $h \ge 1$ ,  $b_h + r_h \le b_{h-1}$ .

*Proof.* The two nodes in a red-red conflict have the same weighted height, which we call the height of the red-red conflict. Denote by  $\mathcal{R}_h(T)$  the number of red-red conflicts of height h in T. The following facts can be verified by inspection of the operations:

• For an *insertion* of any weighted height,  $\Delta \mathcal{R}_1(T) \leq 1$ and  $\Delta \mathcal{R}_h(T) = 0$ , for  $h \neq 1$ .

• For a *blacking operation* of weighted height  $k, \Delta \mathcal{R}_k(T) \leq -1, \Delta \mathcal{R}_{k+1}(T) \leq 1$  and  $\Delta \mathcal{R}_h(T) = 0$ , for  $h \notin \{k, k+1\}$ .

• For a red-balancing operation of weighted height k,  $\Delta \mathscr{R}_k(T) = -1$  and  $\Delta \mathscr{R}_h(T) = 0$ , for  $h \neq k$ .

• For any other operation of any weighted height,  $\Delta \mathscr{R}_h(T) \leq 0$ , for all *h*.

For an empty chromatic tree,  $T_0$ ,  $\mathscr{R}_h(T_0) = 0$  for all h, so by the facts above,  $\mathscr{R}_1(T) \leq i - (r_1 + b_1)$  and  $\mathscr{R}_h(T) \leq b_{h-1} - (r_h + b_h)$ , for  $h \geq 2$ . As  $\mathscr{R}_h(T)$  is never negative, the lemma follows.

For the proofs of the next two lemmas, we need the following definition. For any chromatic tree T, we define an *expanded tree*, T', containing only nodes of weight 0 and 1. T' is constructed from T by replacing each overweighted node by a path, the length of which is equal to the weight of the original node minus one. All nodes on the path have weight 1. We call all nodes on the path, except the one closest to the leaves, *heavy*. Note that these heavy nodes are unary. Red and black nodes in T are left unchanged in T', except for the root, which is replaced by a path extending

infinitely upwards. The nodes on this path have weight 1. The example given in Fig. 3 illustrates the construction of T'. Heavy nodes are marked by an asterisk.

The expanded tree  $T'_0$  for the empty tree  $T_0$  is defined as a path containing nodes of weight 1 extending infinitely upwards—in other words, the expanded tree of an empty tree is identical to the expanded tree of a tree with only one node. As before, the weighted height of a node in T' is defined as the weight of any path from the node to a leaf. One can think of T' as a way of assigning appropriate weighted heights to the overweight in T.

LEMMA 7. For 
$$h \ge 1$$
,  $d_h + w1_h + w2_h + w7_h + p_h \le b_{h-1}$ .

*Proof.* For a chromatic tree, *T*, we define a sequence  $\Psi_h(T)$ ,  $h \in \{1, 2, 3, ...\}$ , of potential functions on *T*, letting  $\Psi_h(T)$  denote the number of nonred (black or heavy) nodes of weighted height *h* in the expanded tree *T'*. An operation of weighted height *k* can change *T* and thereby *T'*, resulting in a change in  $\Psi_h(T)$  for some *h*'s. We denote this change by  $\Delta \Psi_h(T)$  and state some facts concerning this:

• For an *insertion* of any weighted height,  $\Delta \Psi_1(T) \leq 1$ and  $\Delta \Psi_h(T) = 0$  for  $h \neq 1$ .

• For a *blacking operation* of weighted height k,  $\Delta \Psi_{k+1}(T) = 1$  and  $\Delta \Psi_h(T) = 0$ , for  $h \neq k+1$ .

• For any other operation of any weighted height,  $\Delta \Psi_h(T) \leq 0$  for all *h*. In particular, for a *deletion* not resulting in an empty tree, a *weight decreasing operation* 1, 2, or 7 or a *push operation* of weighted height k,  $\Delta \Psi_k(T) = -1$ .

These facts can be verified by inspecting the operations on T and drawing the expanded tree T' before, as well as after, the operation occurs. Note that deletion of one overweighted node in T implies the removal of several nodes in T'. For sequences of operations where there is always at least one node in the tree, except initially, the lemma is proven as follows. For an empty chromatic tree  $T_0$ ,  $\Psi_h(T_0) = 1$  for all *h*, so by the facts above,  $\Psi_1(T) \leq 1 + i - d_1$  and  $\Psi_h(T) \leq i + i - d_1$  $1 + b_{h-1} - (d_h + w1_h + w2_h + w7_h + p_h)$  for  $h \ge 2$ . As  $\Psi_h(T) \ge 1$  for all h and T, the desired inequality is proven. For more general sequences during which empty trees are encountered, the inequality will hold between any two consecutive empty trees. As no rebalancing occurs on an empty structure, the inequality holds for the entire sequence.

LEMMA 8. For  $h \ge 2$ ,  $z_h \le d_{h-1} + w7_{h-1} + p_{h-1}$ , where  $z_h = d_h + w1_h + w2_h + w3_h + w4_h + w5_h + w6_h + 2w7_h + p_h$ .

*Proof.* In a chromatic tree, T, denote by  $\mathscr{H}_h(T)$  the number of heavy nodes of weighted height h in the expanded tree T'. Note that  $h \ge 2$ , as there are no heavy nodes of weighted height 1 in T'. The following facts can be verified by inspection of the operations:

• For a *deletion* of weighted height 1,  $\Delta \mathscr{H}_2(T) \leq 1$ . For a deletion of weighted height k > 1,  $\Delta \mathscr{H}_{k+1}(T) \leq 1$  and  $\Delta \mathscr{H}_k(T) = -1$ . For any deletion and any  $\Delta \mathscr{H}_h(T)$  not mentioned above,  $\Delta \mathscr{H}_h(T) \leq 0$ .

• For a weight decreasing operation 1 to 6 of weighted height k,  $\Delta \mathscr{H}_k(T) \leq -1$  and  $\Delta \mathscr{H}_h(T) = 0$  for  $h \neq k$ .

• For a weight decreasing operation 7 of weighted height k,  $\Delta \mathscr{H}_k(T) = -2$ ,  $\Delta \mathscr{H}_{k+1}(T) \leq 1$ , and  $\Delta \mathscr{H}_h(T) \leq 0$  for  $h \notin \{k, k+1\}$ .

• For a *push* operation of weighted height k,  $\Delta \mathscr{H}_k(T) = -1$ ,  $\Delta \mathscr{H}_{k+1}(T) \leq 1$ , and  $\Delta \mathscr{H}_h(T) = 0$  for  $h \notin \{k, k+1\}$ .

• For any other operation of any weighted height,  $\Delta \mathscr{H}_h(T) \leq 0$ , for all *h*.

By the facts above,  $\mathscr{H}_2(T) \leq d_1 - z_2$  and  $\mathscr{H}_h(T) \leq d_{h-1} + w_{h-1} + p_{h-1} - z_h$  for  $h \geq 3$ . The result follows.

We include the following lemma from [8] for completeness.

LEMMA 9. The total number of red-balancing operations cannot exceed i, and the total number of weight decreasing operations cannot exceed d.

*Proof.* Red-red conflicts can only be introduced by insertions, each of which increases the number of red-red conflicts by at most one. Since each red-balancing operation removes at least one red-red conflict, the total number of red-balancing operations is bounded by *i*. Similarly, overweight can only be introduced by deletions, each of which increases the total amount of overweight in the structure by at most one. Since each weight decreasing operation removes at least one unit of overweight, the number of weight decreasing operations is bounded by *d*.

**Proof of Theorem 4.** The last part of Theorem 4 follows by adding the results of Lemmas 6 and 8 and using the two remaining lemmas (in the case h = 1, only Lemma 6 is needed, as blacking and red-balancing operations are the only rebalancing operations that can have weighted height one). The first part of the theorem is proven as follows. Lemmas 5 and 6 imply that the total number of blacking and redbalancing operations cannot exceed  $\lfloor \sum_{h=1}^{t} (i/2^{h-1}) \rfloor$ , which is bounded by 2i - 1 for all finite weighted heights t. In the same way, Lemmas 5 and 7 imply that the total number of push operations cannot exceed i-1 (remember that a push operations has weighted height at least two). By Lemma 9, the total number of weight-decreasing operations cannot exceed d. Adding these three bounds gives the first part of Theorem 4.

In the rest of this section, we give various extensions to Theorem 4. A version of Theorem 4 for the situation starting with a chromatic tree which is *nonempty* and red-black (hence, in balance) can also be proven, giving bounds on the number of rebalancing operations that can occur before the



**FIG. 3.** Construction of T'.

tree is again in balance. To do this, we need some bounds on the values of the potential functions, occurring in the proofs of Lemmas 5 to 8, for nonempty red-back trees.

LEMMA 10. If T denotes a red-black tree containing n > 0 elements and  $\Phi_h(T)$ ,  $\mathcal{R}_h(T)$ ,  $\Psi_h(T)$ , and  $\mathcal{H}_h(T)$  denote the potential functions from the proofs of Lemmas 5, 6, 7, and 8, respectively, then

(i)  $0 \leq \Phi_h(T) \leq (n/2^{h+1})$ 

(ii) 
$$0 = \mathcal{R}_h(T)$$

- (iii)  $(n/4^{h-1}) \leq \Psi_h(T) \leq \max\{1, (n/2^{h-1})\}$
- (iv)  $0 = \mathscr{H}_h(T).$

These bounds hold for any h for which these potential functions are defined.

*Proof.* By the weighted height of an edge, we mean the weighted height of its lower endpoint. By the *connected* components of weighted height h in T, we mean the connected components of the subgraph induced by the edges having weighted height h. When T is red-black, such a connected component must be symmetric to one given in Fig. 4.

Note that the root of such a component of weighted height h is a leaf of a component of weighted height h + 1. Let T' denote the expanded tree corresponding to T. Recall that  $\Psi_h(T)$  denotes the number of nonred nodes in T' of



FIG. 4. Types of connected components of a given weighted height.

weighted height *h*. When *T* is red-black, any nonred node is black, and *T'* is equal to *T*, except for the infinite path added above the root. Any black node, except the root and nodes on the infinite path above it, must be a leaf node in one of the components above. As all of the components have at least two and at most four leaves, and as  $\Psi_1(T) = n$ , the third inequality follows (the expression max  $\{1, (n/2^{h-1})\}$  is needed because  $\Psi_h(T) \ge 1$  for all *h*, due to the added infinite path). The first inequality follows from the third by noting that any red connected component *r* of height *h* (as defined in the proof of Lemma 5) for which |r| > 1 is part of a component of type *C* containing 4 nonred nodes of weighted height *h*. Thus,  $\Phi_h(T) \le \frac{1}{4}\Psi_h(T)$  for all *h*. The second and the last assertions are direct consequences of the definition of a red-black tree.

We can now prove the following versions of Lemmas 5, 6, 7, and 8. They all apply to the situation where *i* insertions, *d* deletions, and some rebalancing operations have taken place on an initially red-black tree  $T_1$  of *n* elements, resulting in a chromatic tree  $T_2$ . This will not be repeated in the statements of the lemmas. Without loss of generality, we may assume that  $T_2$  is also red-black, since, as noted in Section 2, the rebalancing operations are defined so that if no further updates occur, but rebalancing occurs as long as any operation is applicable, the tree will eventually become red-black. The notation used in the following lemmas is defined before Lemma 5.

LEMMA 11. For  $h \ge 1$ , we have  $b_h \le (1/2^h)i + (h/2^{h+2})n$ .

*Proof.* The proof is the same as the proof of Lemma 5, except that in the equalities at the end of the proof, the value of the potential function of the initial tree should be taken into account. By the facts stated in the proof,  $\Phi_1(T_2) \leq \Phi_1(T_1) + i-2b_1$  and  $\Phi_h(T_2) \leq \Phi_h(T_1) + b_{h-1} - 2b_h$  for  $h \geq 2$ . By Lemma 10,  $b_h \leq \frac{1}{2}(\Phi_h(T_1) - \Phi_h(T_2) + b_{h-1}) \leq \frac{1}{2}(n/(2^{h+1}))$ 

 $-0 + b_{h-1}$ ), which holds for  $h \ge 1$ , since  $b_0$  is defined to be *i*. Using this inequality recursively,

$$b_h \leqslant \frac{1}{2^h} b_0 + \sum_{j=0}^{h-1} \frac{1}{2^{j+1}} \frac{n}{2^{(h-j)+1}} = \frac{1}{2^h} i + \frac{h}{2^{h+2}} n.$$

LEMMA 12. For  $h \ge 1$ ,  $b_h + r_h \le b_{h-1}$ .

*Proof.* In the proof of Lemma 6, the bounds  $\mathscr{R}_h(T_1) = 0$  and  $\mathscr{R}_h(T_2) \ge 0$  for all *h* were used. As noted in Lemma 10, these bounds still hold when  $T_1$  is nonempty, so the proof goes through.

*Proof.* The proof is the same as the proof of Lemma 7, except that we obtain the inequality  $d_h + w1_h + w2_h + w7_h + p_h \leq \Psi_h(T_1) - \Psi_h(T_2) + b_{h-1}$ , which holds for  $h \geq 1$ , since  $w1_1$ ,  $w2_1$ ,  $w7_1$ , and  $p_1$  are defined to be 0, and  $b_0$  is defined to be *i*. The number of elements in  $T_2$  is n + i - d, so the inequality follows from Lemma 10.

LEMMA 14. For  $h \ge 2$ ,  $z_h \le d_{h-1} + w7_{h-1} + p_{h-1}$ , where  $z_h = d_h + w1_h + w2_h + w3_h + w4_h + w5_h + w6_h + 2w7_h + p_h$ .

*Proof.* The proof is the same as the proof of Lemma 8.

From these lemmas we obtain the following theorem.

THEOREM 15. If i > 0 insertions and d deletions are performed on an initially red-black tree, containing n > 0 elements, then the number of rebalancing operations of weighted height h that can occur is bounded by

$$\frac{3}{2^{h-1}}i + \frac{3h-5}{2^{h+1}}n + \max\left\{1, \frac{n}{2^{h-2}}\right\} - \frac{1}{4^{h-2}}(n+i-d)$$

for  $h \ge 2$  and by i for h = 1.

*Proof.* This follows by adding the results of Lemmas 12 and 14, and then using Lemmas 11 and 13. For the case h = 1, only Lemma 12 is needed, as blacking and red-balancing operations are the only rebalancing operations that can have weighted height one.

For Theorem 4, results about the *total* number of rebalancing operations were proven by adding up the bounds for each weighted height. Using the same technique in the case where the initial tree is nonempty gives suboptimal results, due to the extra term in the bound in Lemma 11, compared to Lemma 5. We can improve on this by using a different potential function.

**THEOREM 16.** If i > 0 insertions and d deletions are made on an initially red-black tree, containing n > 0 elements, then at most  $\frac{5}{2}i + \frac{3}{2}d + \frac{1}{2}(n-1)$  rebalancing operations can occur. *Proof.* For a chromatic tree T, denote by  $\mathscr{C}(T)$  the set of red connected components (as defined in the proof of Lemma 5) in T, and by  $\mathscr{N}(T)$  the set of nodes in T. Define a potential function  $\Gamma(T)$  by

$$\Gamma(T) = 2\left(\sum_{r \in \mathscr{C}(T)} \left(|r|-1\right)\right) + \sum_{v \in \mathscr{N}(T)} w(v),$$

where w(v) is the weight of the node v. Operations on the tree can change  $\Gamma(T)$  in the following way:

- For an *insertion*,  $\Delta \Gamma(T) \leq 3$
- For a *deletion*,  $\Delta \Gamma(T) \leq -1$
- For a blacking operation,  $\Delta \Gamma(T) \leq -1$
- For a *push operation*,  $\Delta \Gamma(T) \leq -1$
- For any other operation,  $\Delta\Gamma(T) \leq 0$ .

These facts can be verified by inspection of the operations (for the fact concerning blacking operations, recall that in order for a blacking operation to be applied, we require that at least one of the red nodes must be the parent is of another red node). If  $T_1$  denotes the initial tree and  $T_2$  is the final tree, it follows from the above that  $\Gamma(T_2) \leq \Gamma(T_1) + 3i - 3i$ d-b-p, where b and p denote the total number of blacking operations and push operations, respectively, that have occurred. We claim that for any red-black tree T containing m > 0 elements,  $\frac{3}{2}m - \frac{1}{2} \leq \Gamma(T) \leq 2m - 1$ . This claim is proven below. Hence,  $\Gamma(T_1) \leq 2n-1$  and  $\frac{3}{2}(n+i-d) - \frac{1}{2} \leq \Gamma(T_2)$ , as we may assume, without loss of generality, that  $T_2$  is redblack too. Thus, by the inequality above,  $b + p \leq \frac{3}{2}i + \frac{1}{2}d + \frac{1$  $\frac{1}{2}(n-1)$ . By Lemma 9, the number of red-balancing operations is bounded by *i*, and the number of weight decreasing operations is bounded by d. Thus, the total number of rebalancing operations that can occur is bounded by  $\frac{5}{2}i$  +  $\frac{3}{2}d + \frac{1}{2}(n-1).$ 

We now prove the claim made above. Let T denote a redblack tree containing m > 0 elements. Define the connected components of weighted height h as in the proof of Lemma 10. If the root of T has weighted height k, we can consider T as consisting of a root, which is the top node in the connected component of weighted height k - 1, the leaves of which are the top nodes of the connected components of weighted height k - 2, and so forth. In the process of building T in this way from a single node (the root), each component added will increase  $\Gamma$  as well as the number of leaves l by amounts  $\Delta\Gamma$  and  $\Delta l$ , depending on the type of the component. From Fig. 4 in the proof of Lemma 10, we obtain the amounts given in Fig. 5.

It appears that for each increase in l,  $\Gamma$  is increased by at least  $\frac{3}{2}$  times as much. At the beginning of the process,  $l = \Gamma = 1$ , as there is just one black node. Since l is increased from 1 to m,  $\Gamma$  is increased by at least  $\frac{3}{2}(m-1)$ , so at the end of the process,  $\Gamma(T) \ge \frac{3}{2}(m-1) + 1 = \frac{3}{2}m - \frac{1}{2}$ .

Туре	A	В	C
$\Delta\Gamma$	2	3	6
$\Delta l$	1	2	3

FIG. 5. Increases for the different components types.

On the other hand, when adding a component of type A, B, or C during this process,  $\Gamma$  is increased by at most twice as much as l. Thus,  $\Gamma(T) \leq 2(m-1) + 1 = 2m - 1$ , and the claim is proven.

Basically, Theorems 15 and 16 can be paraphrased by saying that when the number of updates is  $\Omega(n)$ , there are still only O(1) rebalancing operations per update and the number of rebalancing operations is still an (essentially) exponentially decreasing function of their weighted height, even when the initial tree is nonempty. However, if the number of updates is much smaller than n, then the bound in Theorem 16 is not tight, as the results from [8] show that no more than  $\lfloor \log_2(n+i) \rfloor$  rebalancing operations per update are ever needed.

The proof of Theorem 16 can also be applied when the initial tree is empty and, hence,  $\Gamma(T_1) = 0$ . This gives the following result.

**THEOREM 17.** If i > 0 insertions and d deletions are made on an initially empty chromatic tree, then at most  $\frac{5}{2}i + \frac{3}{2}d + \frac{1}{2}$ rebalancing operations can occur.

This is a better bound than that of Theorem 4, as  $d \le i$  if the initial tree is empty.

We close this section by noting that Theorems 16 and 17 can be tightened slightly. This is done as follows. We consider a conceptual set of operations, which is slightly changed compared to the actual operations used. On the conceptual level, we introduce two more rebalancing operations, namely root blacking and root weight decreasing. The first can be applied any time the root is red, and its effect is to make the root black. The second can be applied any time the root is overweighted, and its effect is also to make the root black. On the conceptual level, the rest of the operations are as in the Appendix, without the addition of leaving the root black all the time, as described in Section 2. Thus, an actual blacking operation or insertion at the root corresponds, on the conceptual level, to a blacking operation *plus* a root blacking operation; and an actual push operation, w7 operation, or deletion at the root corresponds, on the conceptual level, to the same operation *plus* a root weight decreasing operation.

If we also add a (conceptual) unary red node above the root of our trees, it can be verified that a root blacking operation or a root weight decreasing operation changes the potential function  $\Gamma$  in the same way as ordinary blacking operations or weight decreasing operations, respectively. Thus, the bounds obtained using these functions hold for *any* sequence of operations on the conceptual level, in particular for sequences of actual operations, where root blackings can be considered to occur immediately after insertions and (ordinary) blackings, and where the root weight decreasing operations occur immediately after pushes, w7 operations, and deletions. As the two new operations are actually part of the other operations as defined in this paper, their number can be subtracted from the bounds mentioned above when counting how many operations actually occur.

On the conceptual level, the only operations that can change the weighted height of the root are the two new operations, plus insertions into an empty tree, and deletions of the last element of the tree. If  $W_{\rm max}$  denotes the maximum weighted height of the tree during the sequence of operations, and  $W_{\rm start}$  and  $W_{\rm final}$  denote the initial and the final weighted height of the tree, respectively, then at least  $W_{\rm max} - \max\{W_{\rm start}, 1\}$  root blackings must have occurred and at least  $W_{\rm max} - \max\{W_{\rm final}, 1\}$  root weight decreasing operations must have occurred. Accordingly, the bounds can be lowered by these amounts. For example, if the initial tree is empty, then at most  $\frac{5}{2}i + \frac{3}{2}d + \frac{1}{2} - 2W_{\rm max} + \max\{W_{\rm start}, 1\} + \max\{W_{\rm final}, 1\}$  rebalancing operations can occur.

Notice that if the tree remains reasonably close to a redblack tree, then  $W_{\text{max}}$  will be logarithmic in the maximum number of elements in the tree.

#### 4. PARALLEL PRIORITY QUEUES

The problems with using priority queues in a parallel environment have been investigated extensively. Two different, largely incomparable, models for parallelism are used. One model deals with synchronous parallel processors, and the other model, which we use here, deals with asynchronous parallel processors which have access to a shared memory. In designing pointer-based priority queues to be used in such an environment, the problem of allowing processes to lock nodes, in order to avoid inconsistencies from updates and rebalancing operations, without decreasing the degree of parallelism too much is even more serious than it is in the design of search trees, since small values are accessed frequently. Clearly, using a standard heap organization and locking the root before all updates creates a congestion problem at that node and forces sequentiality in the access to the data structure.

Previous work on priority queues in an environment with asynchronous parallel processes [5, 13, 23] all use some form of a heap [27]. Thus, the root has to be locked at the beginning of each deletemin operation, thereby preventing other processes from accessing the data structure during that time. In fact, [5] also locks the last node at the beginning of a deletemin. In [13, 23], the root is also locked at the beginning of an insertion, and step-wise locking down the update path is always used, thus further restricting the possibilities for parallelism. Additionally, since they all work on heap structures, insert as well as deletemin are  $\Omega(\log n)$  in the worst case and this many locks are necessary, although not necessarily at the same time. In fact, even if data is drawn from a uniform distribution, the expected complexities for insert and deletemin are still  $\Theta(\log n)$ .

As already stated, binary search trees can sometimes be successfully used as priority queues. In this section, we present a quite competitive priority queue for a parallel asynchronous environment. It is based on chromatic search trees, with some modifications. Using chromatic trees to implement priority queues gives the advantage that the updating is uncoupled from the rebalancing. The idea of uncoupling updating and rebalancing in a priority queue has also been considered in [5], but much more rebalancing is necessary there than in our proposal.

Our priority queue is tailored to problems where serializability is not a requirement, i.e., where it is not necessary that there always exists a way of sequencing all the operations which have been performed on the parallel priority queue, such that, if they were performed on a sequential priority queue, all the deletemin operations would return the same values as they did in the parallel version. This means that when serializability is necessary for the correctness of an algorithm using a priority queue, our structure cannot be used. However, there are many applications (including the large class of problems which can be solved using branch and bound techniques), where fast access is the key issue and where the correctness does not depend on serializability.

Our structure differs from previous work in that the processes doing updates need not have an exclusive lock on the root, so the number of processes which can work on the structure at any one time is not limited by the height of the tree; parallelism of the order the size of the tree is possible. The congestion problem is further reduced by the fact that our deletemin is constant time and the inserts (with the exception of inserting a minimal element) proceed independently from deletemins.

Of course, the complexity results for chromatic trees carry over, sometimes with minor changes. Improved results can, however, be proven if a general delete operation is not wanted. We obtain that if the initial tree is empty, *i* insertions and any number of deletemins give rise to at most 2i - 1 rebalancing operations. As for chromatic search trees, this means that the necessary exclusive locks will be very localized in space, as well as in time, allowing for a high degree of parallelism. Additionally, we obtain that the number of rebalancing operations which can occur at weighted height *h* is at most  $i/2^{h-1}$ . We also have results for the case in which the structure is initially nonempty.

#### 4.1. Chromatic Priority Queues

Modifications of red–black and chromatic trees give us red–black and chromatic priority queues. These structures are also full binary search trees which are leaf-oriented, so the minimum element will be the left-most leaf. We allow overweight on the left-most path, to decrease the amount of rebalancing necessary there and thus the probability of contention there. The operations used are the same as those described in Section 2 for chromatic trees, so the updating is still uncoupled from the rebalancing.

DEFINITION 18. A *red-black priority queue* is a red-black tree with a pointer to the element with minimum key and all of the leaves kept in a doubly linked list, except that conditions B1 and B4 from Definition 1 are replaced with:

B1: The leaves of T are black, except that the left-most leaf may be overweighted.

B4: T has only red and black nodes, except on the left-most path.

DEFINITION 19. A *chromatic priority queue* is a chromatic tree with a pointer to the element with minimum key and all of the leaves kept in a doubly linked list.

The pointer to the element with minimum key is used for the findmin and the deletemin operations. The doubly linked list of leaves facilitates updating this pointer after a deletemin, since the new minimum is in the next leaf. This makes the deletemin operation constant time. The list of leaves is doubly linked, rather than singly linked, so that insertions can be done more efficiently. After the proper location for an insertion is found by searching, as in a binary search tree, the actual update is constant time. Clearly, the findmin operation is also constant time, and so is the creation of an empty queue.

Notice that allowing overweight on the left-most path does not change the fact that the length of any path in a redblack priority queue is  $O(\log n)$ , where *n* is the number of nodes in the structure, since the right subtree of the root is still a red-black tree. It does, however, decrease the amount of rebalancing necessary near the minimum element. If the only deletions allowed are deletemins, then the weight decreasing and push operations are unnecessary, as overweight can only move upwards and thus would remain on the left-most path.

In some applications of priority queues, it is useful to allow arbitrary deletions, in addition to the deletemin operation. Since chromatic priority queues are based on chromatic trees, which have an arbitrary deletion operation and the necessary rebalancing operations, we can also allow these deletions in chromatic priority queues. Note that, as opposed to most other implementations of priority queues, searches for elements are supported and it is therefore not necessary to have a pointer to the element to be deleted.

The deletion operation is essentially the same as the deletemin, except that, in order to locate the element to be deleted, the updater performs a search in the tree, starting at the root, rather than using the pointer to the minimum element. Introducing this deletion operation, however, introduces the possibility of overweight nodes off the leftmost path. The weight decreasing and push rebalancing operations can be used here, although we still never use them for the purpose of reducing overweight on the leftmost path if that overweight is solely due to a deletemin operation. If, however, applying one of these rebalancing operations to decrease the overweight off the left-most path incidentally also decreases overweight on the left-most path, that is fine. In addition, if any process doing a deletion (other than a deletemin) or a rebalancing operation incidentally introduces some overweight onto a node on the left-most path, it will simply put a pointer to the overweight node into the problem queue for the rebalancing processes, which will be allowed to handle it. Through careful design of the problem queue, it is possible to avoid removing overweight created by deletemins. However, a simpler approach might be to mark overweight created by deletemins. This can be done using only one bit per node (see Section 4.3.1).

It is clear that the chromatic priority queue can be made double-ended, simply by keeping a pointer to the maximum element, also, and allowing overweight on the right-most path. Results similar to those presented in the the next section hold for double-ended priority queues.

#### 4.2. Complexity

The complexity of chromatic priority queues is essentially the same as the complexity of the chromatic trees on which they are built, except that if deletemin operations are the only deletions made, then we can improve on the constants in the bounds, as we do not need to rebalance any overweight. In addition, some results change slightly because of the overweight allowed on the left-most path of a red-black priority queue.

The results from [8] also apply to chromatic priority queues, although in a slightly modified version. This is summarized in the theorem below. We give a brief explanation of the necessary changes here. Since we now assume that we are starting from a red-black priority queue, instead of a red-black tree, Lemma 4.4 and Theorem 4.5 from [8] only hold for edges not on the left-most path, and the *M* in Theorem 4.5 should be  $\lfloor \log_2(N-1) \rfloor$  because the left-most path in the original red-black priority queue could have consisted of only one edge, which had a considerable amount of weight. Thus, we have:

THEOREM 20. If *i* insertions and any number of deletemins are performed on a chromatic priority queue which is initially a red-black priority queue with *n* items (leaves), then at most  $i(\lfloor \log_2(n+i-1) \rfloor + 1)$  rebalancing operations can occur. In addition, the number of rebalancing operations which actually change the structure of the tree is at most i. If i insertions and d arbitrary deletions (including any deletemins) are performed, the result is  $(i+d)(\lfloor \log_2(n+i-1) \rfloor + 1) - d$ , with at most i+d rebalancing operations changing the structure of the tree.

In the proofs of Lemmas 5 through 8, there is no assumption that the final tree is red–black. Thus, these lemmas still apply to chromatic priority queues, and we get the following results.

THEOREM 21. If i > 0 insertions and any number of deletemins are performed on an initially empty chromatic priority queue, then at most 2i - 1 rebalancing operations can occur. Furthermore, the number of rebalancing operations of weighted height h that can occur is bounded by  $i/2^{h-1}$ . If i > 0 insertions and d deletions (including deletemins) are performed on an initially empty chromatic priority queue with arbitrary deletions, then at most 3i + d - 2 rebalancing operations of veighted height h that can occur is bounded by  $3i/2^{h-1}$  for  $h \ge 2$  and by i for h = 1.

*Proof.* The last part is just a restatement of Theorem 4. The first part follows the proof of the same theorem, except that only Lemmas 5 and 6 are used, since blacking and red-balancing are the only rebalancing operations used.

The theorems from Section 3, for the case where the initial tree is nonempty, do not directly apply to chromatic priority queues, as the bounds on the various potential functions are affected by the presence of an overweighted left-most path. We proceed to make the necessary changes. For chromatic priority queues, Lemma 10 appears as follows.

LEMMA 22. If T denotes a red-black priority queue containing n > 0 elements and  $\Phi_h(T)$ ,  $\mathcal{R}_h(T)$ ,  $\Psi_h(T)$ , and  $\mathcal{H}_h(T)$  denote the potential functions from the proofs of Lemmas 5, 6, 7, and 8, respectively, then

- (i)  $0 \leq \Phi_h(T) \leq (1/2^{h+1})(n-1) + \frac{1}{4}$
- (ii)  $0 = \mathscr{R}_h(T)$
- (iii)  $(1/4^{h-1}) n \leq \Psi_h(T) \leq (1/2^{h-1})(n-1) + 1$
- (iv)  $0 \leq \mathscr{H}_h(T) \leq 1.$

These bounds hold for any h for which these potential functions are defined.

*Proof.* The proof follows the proof of Lemma 10, except that we now consider T' instead of T and the possible connected components are given in Fig. 6.

Components of type *D* can only occur on the left-most path, and there is at most one for each weighted height *h* in the tree. By the argument from the proof of Lemma 10, it follows that  $\Psi_{h+1}(T) \leq \frac{1}{2}(\Psi_h(T)-1)+1 = \frac{1}{2}\Psi_h(T) + \frac{1}{2}$  for all *h*. As all leaves in *T'* have weight 1,  $\Psi_1(T) = n$ . Using the



**FIG. 6.** Types of connected components for T'.

inequality recursively, we obtain  $\Psi_h(T) \leq (1/2^{h-1}) n + \sum_{i=1}^{h-1} (1/2^i) = (1/2^{h-1})n + (1 - (1/2^{h-1})) = (1/2^{h-1})(n-1) + 1$ . This gives the upper bound in the third inequality. The lower bound is unchanged from Lemma 10. The first inequality follows from the third, and the remaining two are trivial consequences of the definition of chromatic priority queues.

Repeating the proofs of Lemmas 11 through 14 with the above bounds gives us the following version of Theorem 15. We omit the details of the proof.

THEOREM 23. If i > 0 insertions and any number of deletemins are performed on an initially red-black priority queue containing n > 0 elements, then the number of rebalancing operations of weighted height h that can occur is bounded by

$$\frac{1}{2^{h-1}}i + \frac{h-1}{2^{h+1}}(n-1) + \frac{1}{4}\left(1 - \frac{1}{2^{h-1}}\right).$$

If i > 0 insertions and d deletions (including deletemins) are performed on an initially red-black priority queue with arbitrary deletions, containing n > 0 elements, then the number of rebalancing operations of weighted height h that can occur is bounded by

$$\frac{3}{2^{h-1}}i + \frac{3(h+1)}{2^{h+1}}(n-1) - \frac{1}{4^{h-2}}(n+i-d) - \frac{3}{2^{h+1}} + \frac{5}{2}$$

for  $h \ge 2$ , and by i for h = 1.

As in Section 3, the lemmas leading to this theorem are not well suited to proving results about the total number of rebalancing operations. Using potential functions pertaining to the whole tree improves on this. In the case where deletemins are the only deletions occurring, we have the following.

THEOREM 24. If i > 0 insertions and any number of deletemin operations are performed on an initially red-black priority queue containing n > 0 elements, then at most  $2i + \frac{2}{3}(n-1)$  rebalancing operations can occur.

*Proof.* For a chromatic priority queue T, denote by  $\Pi(T)$  the total number of red nodes in T. Operations on the priority queue can change  $\Pi(T)$  in the following way:

- For an *insertion*,  $\Delta \Pi(T) \leq 1$ .
- For a blacking operation,  $\Delta \Pi(T) \leq -1$ .

• For a deletemin operation or a red-balancing operation,  $\Delta \Pi(T) \leq 0$ .

These facts can be verified by inspection of the operations. No other operations are employed on a chromatic priority queue without general deletions. If  $T_1$  denotes the initial priority queue and  $T_2$  the final priority queue, it follows from the above that  $\Pi(T_2) \leq \Pi(T_1) + i - b$ , where b is the total number of blacking operations that have occurred. To bound  $\Pi(T_1)$  and  $\Pi(T_2)$ , we note the following. In a redblack priority queue, any red node must, by definition, have two nonred children, so at most one third of the nodes (not counting the root, as this is always black) can be red. As the priority queue is a full binary tree, the total number of nodes is twice the number of leaves minus one. This implies that  $\Pi(T_1) \leq \frac{1}{3}((2n-1)-1) = \frac{2}{3}(n-1)$ . Trivially,  $\Pi(T_2) \geq 0$ . Thus,  $b \leq \frac{2}{3}(n-1) + i$ . By Lemma 9, the number of redbalancing operations never exceeds i. As blacking operations and red-balancing operations are the only rebalancing operations employed in a chromatic priority queue without general deletions, the theorem is proven.

For the case where general deletions are allowed, the following version of Theorem 16 applies.

THEOREM 25. If i > 0 insertions and d deletions (including deletemins) are made on an initially red-black priority queue with arbitrary deletions, containing n > 1 elements, then at most  $\frac{5}{2}i + \frac{3}{2}d + \frac{1}{2}(n-1) + 2\log_2(n-1)$  rebalancing operations can occur. If n = 1, then at most  $\frac{5}{2}i + \frac{3}{2}d$  rebalancing operations can occur.

*Proof.* Following the proof of Theorem 16, we arrive at the inequality  $\Gamma(T_2) \leq \Gamma(T_1) + 3i - d - b - p$ , where b and p denote the total number of blacking operations and push operations, respectively, that have occurred. We claim that for any red-black priority queue T, containing m > 1elements and having a root which is not overweighted,  $\frac{3}{2}m - \frac{1}{2} \leq \Gamma(T) \leq 2m - 1 + \log_2(m - 1)$ . This claim is proven below. If m = 1 or m = 0,  $\Gamma(T) = 1$  or  $\Gamma(T) = 0$ , respectively. Hence, if n > 1,  $\Gamma(T_1) \leq 2n - 1 + \log_2(n-1)$  and  $\frac{3}{2}(n+i-d)$  $-\frac{1}{2} \leq \Gamma(T_2)$ , as we may assume, without loss of generality, that  $T_2$  is red-black too. Thus, by the inequality above,  $b + p \leq d$  $\frac{3}{2}i + \frac{1}{2}d + \frac{1}{2}(n-1) + \log_2(n-1)$ . If n = 1, we obtain  $b + p \le 1$  $\frac{3}{2}i + \frac{1}{2}d$ . Lemma 9 can be directly applied to give that the number of red-balancing operations never exceeds *i*, and its proof can be applied to give that the number of weight decreasing operations never exceeds d plus the amount of overweight in  $T_1$ , which we claim is bounded by  $\log_2(n-1)$ when n > 1. Thus, if n > 1, the total number of rebalancing operations that have occurred is bounded by  $\frac{5}{2}i + \frac{3}{2}d + \frac{3}{2}d$  $\frac{1}{2}(n-1)+2\log_2(n-1)$ . If n=1, there is no overweight in  $T_1$ , and we obtain the bound  $\frac{5}{2}i + \frac{3}{2}d$ .

We now prove the claims made above. Let T denote a red-black priority queue containing m > 1 elements and

having a root which is not overweighted, and let T' denote the expanded tree, as defined in Section 3, except that no infinite path is added above the root. Then, obviously,  $\Gamma(T) = \Gamma(T')$ . Define the connected components of weighted height h as in the proof of Lemma 10. If the root of T has weighted height k, we can consider T' as consisting of a root, which is the top node in the connected component of weighted height k-1, whose leaves are the top nodes of the connected components of weighted height k-2, and so forth. In the process of building T' in this way from a single node (the root), each component added will increase  $\Gamma$  as well as the number of leaves l by amounts  $\Delta\Gamma$  and  $\Delta l$ , depending on the type of the component. From Fig. 6 in the proof of Lemma 22, we obtain the amounts given in Fig. 7:

It appears that for each increase in l,  $\Gamma$  is increased by at least  $\frac{3}{2}$  times as much. At the beginning of the process,  $l = \Gamma = 1$ , as there is just one black node. Since l is increased from 1 to m,  $\Gamma$  is increased by at least  $\frac{3}{2}(m-1)$ , so at the end of the process,  $\Gamma(T') \ge \frac{3}{2}(m-1) + 1 = \frac{3}{2}m - \frac{1}{2}$ .

On the other hand, when adding a component of type A, B, or C during this process,  $\Gamma$  is increased by at most twice as much as *l*. Adding a component of type *D* increases  $\Gamma$  by one, while leaving l unchanged. Thus,  $\Gamma(T') \leq 2(m-1) + 1$ 1 + a = 2m - 1 + a, where a denotes the number of components of type D. Components of type D are due to overweight, so if T is red-black, they can only appear as part of the left-most path of T'. Also, by the way T' is constructed from T, the top node of such a component cannot be the root. Thus, if the root of T' has weighted height k, there can at most be k-2 components of type D. Though nodes on the left-most path are allowed to be overweighted, the root will never be overweighted because the operations which occur at the root always set the root's weight to one. Hence the right subtree of the root of T' is a red-black tree of weighted height k-1, and thus contains at least  $2^{k-2}$ leaves (in the building process above, no components of type D appear, so each new weighted level at least doubles *l*). As the left subtree of the root of T' contains at least one leaf,  $2^{k-2} \leq m-1$ . Therefore,  $a \leq k-2 \leq \log_2(m-1)$ . Also, for each unit of overweight in T, there is a component of type D in T', so the amount of overweight in T is bounded by  $\log_2(m-1)$ . Thus, the claims are proven.

Туре	A	В	С	D
$\Delta\Gamma$	2	3	6	1
$\Delta l$	1	2	3	0

**FIG. 7.** Increases for the different components types of T'.

The same techniques discussed at the end of Section 3 for tightening the results concerning the total number of rebalancing operations on a chromatic search tree can also be applied to the results for chromatic priority queues. For example, it can be verified that a root blacking operation or a root weight-decreasing operation changes the potential function  $\Pi$  in the same way as ordinary blacking operations or weight-decreasing operations, respectively. Let  $W_{\text{max}}$ denote the maximum weighted height of the tree during the sequence of operations, let  $W_{\text{start}}$  denote the weighted height of the initial tree, and let  $W_{\text{final}}$  denote the weighted height of the final tree. If i > 0 insertions and any number of deletemin operations are performed on an initially redblack priority queue containing n elements, then at most  $2i + \frac{2}{3}(n-1) - 2W_{\text{max}} + \max\{W_{\text{start}}, 1\} + \max\{W_{\text{final}}, 1\}$ rebalancing operations can occur.

#### 4.3. Possible Improvements and Heuristics

The results proven in Subsection 4.2 show that congestion is unlikely to be a problem with a chromatic priority queue. If one wishes to even further reduce the probability of any problem due to congestion, there are some small improvements and heuristics which could be used.

One obvious change to make, to decrease the probability of congestion near the left-most leaf, is to modify the meaning of a router, so that searching proceeds to the right, instead of the left in case of equality. Thus, the router stored in a node v is greater than any key in the left subtree and less than or equal to any key in the right subtree, and for an insertion, the router in the new internal node is given the value of the key in its right child, instead of its left.

Since deletemin is one of the basic operations on a priority queue, it seems likely that the most "active" part of the priority queue will be around the smallest element. Depending on the application, it may be important that insertions of small elements are not delayed significantly; one may be concerned about the possibility that the deletemin operations will be deleting elements while many smaller elements are waiting to be inserted. In this section, we discuss some possible improvements to the data structure and heuristics for dealing with these concerns.

Of course, if any of the heuristics are implemented, the locking scheme of Appendix B should be elaborated on, in order to take the actions of the heuristics into consideration.

4.3.1. Eliminating rebalancing due to deletemins. In Theorems 21, 23, and 25, d is the number of deletions, *including* deletemins. It is easy to change *including* to *excluding* by adding just one bit of information at each node of a chromatic priority queue. Nodes should be marked by the deletemin operations that create overweight on them, and also by the push and w7 operations that move overweight onto them from other marked nodes. In this way, marked

nodes can only appear on the left-most path, so the overweight on marked nodes does not have to be rebalanced. Also, any overweight originating from a deletemin operation will reside on marked nodes, so no such overweight needs to be considered in the proof of the theorem.

4.3.2. Completely eliminating rebalancing on the left-most *path.* One possible heuristic for decreasing the probability of congestion is to avoid any rebalancing at all on the leftmost path of the priority queue, at least if the rebalancing would actually change the structure of the tree, rather than just moving weights around. This would eliminate the need for exclusive locks on this path, which could prevent processes from inserting new elements. It should include cases in which the red nodes or overweighted nodes involved are not actually on the left-most path, if the rebalancing would involve locking nodes on the left-most path. This could, however, theoretically, create a problem with a long string of red nodes off the left-most path, with no possible operation which could be applied, due to the requirement that the top node in the red-red conflict being handled must not be red. That rule would have to be changed so that either the top node should not be red or its parent should be on the left-most path. The effect of this would essentially be to consider each right subtree of a node on the left-most path as a distinct chromatic tree which should be made red-black. It is possible that this decision to avoid rebalancing on the left-most path would lead to an extremely long left-most path-heuristics which address this problem are discussed later in this section.

Another concern with this heuristic is how to implement it: how does one determine if a rebalancing operation should not be applied because it involves the left-most path? One obvious possibility here is to keep track of which nodes are on the left-most path, keeping an extra flag with each node to indicate this. These flags could then be updated as part of the deletemin and insertion operations. The problem with this (other than the additional space required) is that this could occasionally take a lot of time. When a deletemin occurs, the left-most path, of the right subtree involved in the deletion, becomes part of the left-most path of the priority queue. Then, all of the nodes on that path would have to be marked, as part of the deletemin operation. Since one of our main concerns is that this operation be fast, this is definitely a drawback. Although this would occasionally lead to considerable extra work, the total amount of this work for each operation is amortized constant time, so this could be a viable possibility.

Another possibility is for the processor which is about to do a rebalancing operation to check the priority queue to make sure that none of the nodes it is working on are on the left-most path. Clearly, this would only involve checking one of the nodes involved. To determine this, the process could follow parent pointers until it discovers that the node it just came from was a right child (indicating that the original node was not on the left-most path) or it discovers the root (indicating that it was on the left-most path).

Assuming a uniform distribution on the operations, from a random location in the tree, the expected number of parent pointers followed, before finding that the node one just came from was a right child, is two. Thus, under the proper distribution of operations, this could be expected to take average constant time, and it would not entail keeping extra flags. More importantly, it would not slow down the processes doing the actual insertions and deletemins. Of course, under the wrong distribution, this could take time proportional to the longest path in the tree.

4.3.3. Partially eliminating rebalancing on the left-most path. Instead of avoiding rebalancing on the entire leftmost path, one might decide to avoid it on the last k nodes on that path, for some relatively small value k. This would generally be sufficient: it would avoid interference between the processes doing the deletemins and those doing the rebalancing, and rebalancing operations become less and less frequent, closer to the root. The advantage is that one could limit how much extra time is needed. Either of the above-mentioned possibilities, for determining which nodes are on the left-most path, can be chosen, and the worst case extra time per operation will be O(k). (In the case of marking, it would only be necessary to follow k-1 parent pointers from the minimum. In the case of checking if one of the nodes is among the last k nodes on the left-most path, one only needs to follow k-1 left child pointers to see if one reaches the minimum element.) The amortized amount of time in the case of markers is still constant. The expected amount of time would still be constant in the case where there are no markers, if one assumes that the algorithm is simultaneously looking for the minimum element and following parent pointers to see if the one it just came from was a right child.

One could also decide to only avoid rebalancing if any of the nodes involved is actually known to be on the left-most path. Again, there could be a flag for each node, and processes which use that node and know it is on the leftmost path could set the flag. For example, when a deletemin occurs, the processor performing it knows that the new minimum and its parent are on the leftmost path. A process searching for the location for an insertion knows that it is on the left-most path as long as it only follows left child pointers. In addition, when a rebalancing process discovers that a node it uses is marked as being on the left-most path, it knows that that node's parent and left child are on the leftmost path, and it can mark these if it has locked them for the rebalancing operation. There is some overhead involved, but it would only amount to a small constant factor.

4.3.4. What if the left-most path is likely to become extremely long? Now, we return to the possible problem of an

extremely long left-most path. If this is likely to be a problem in the desired application, it would lead to slow insertion times for small elements. A heuristic for getting around this problem would be to keep track of the k smallest elements and always check if an element to be inserted was one of them or not. Since the leaves are kept in a doubly linked list, one can keep track of the k smallest by keeping a pointer to the kth smallest. This pointer can easily be updated when an insertion or deletemin occurs, in constant time. If an insertion process discovers that an element to be inserted will be one of the k smallest, that process can search for the proper location for insertion by starting with the smallest element and following the pointers linking the leaves. If the element is equal to the kth smallest, it can be inserted immediately after the kth smallest. In all other cases, a standard search from the root will locate the appropriate location for the insertion. The value k should probably be close to the logarithm of the expected number of elements in the priority queue to balance the amount of time it takes when the search is from the root with the amount of time it takes when it is from the smallest element.

#### **APPENDIX A: OPERATIONS**

The update and rebalancing operations used in this paper are shown below. Note that the blacking operation has one more restriction, which is not shown: it can only be applied if at least one of the two lower nodes has a child of weight zero.

#### Update operations

(insert)

(deletemin/delete)

# Rebalancing operations

(blacking) 
$$0 \stackrel{v_1 \ge 1}{\longrightarrow} 0 \stackrel{w_1 \ge 1}{\longrightarrow} 1 \stackrel{w_1 - 1}{\longrightarrow} 1$$

 $w_1 \ge 1$ 

 $0 w_1 + w_2$ 

(rb1) 
$$0 \xrightarrow{w_1 \ge 1} w_2 \ge 1 \longrightarrow 0 \xrightarrow{w_1} 0 \xrightarrow{w_2} w_2$$

(rb2) 
$$0 \diamond w_1 \ge 1 \rightarrow 0 \diamond w_2 \ge 1 \rightarrow 0 \diamond w_2 \ge 0$$

$$(\text{push}) \quad \begin{array}{c} w_2 > 1 & & & \\ w_3 > 0 & & & \\ \end{array} \quad \begin{array}{c} w_1 & & \rightarrow & \\ w_2 - 1 & & \\ w_3 & & & \\ \end{array} \quad \begin{array}{c} w_1 + 1 \\ 0 \\ w_3 & & \\ \end{array}$$

(w1) 
$$w_1 > 1$$
  $0$   $w_0 \rightarrow 0$   $w_0$   
 $w_2 > 1$   $0$   $w_1 - 1$   $0$   $w_2 - 1$ 

(w3) 
$$w_1 > 1$$
  $0$   $w_2 > 0$ 





#### APPENDIX B: A LOCKING SCHEME

Clearly, a locking scheme is necessary in order to prevent inconsistencies in chromatic trees and priority queues. We present one possible scheme here to show that such a scheme can be designed. In designing the following locking system, we attempted to allow as many processes as possible to access the same node at the same time, to maximize the amount of parallelism. There are, however, still possibilities for improvement, by making the scheme slightly more complicated. This scheme can also be simplified somewhat if it is applied to chromatic search trees, rather than chromatic priority queues, since the leaves of a chromatic search tree are not kept in a doubly linked list. Our system has some similarities to the system described in [21], but also major differences.

Each node may be locked separately from all others. A test-and-set variable, a semaphore, or whatever the system offers, exists for each node to protect the locking mechanism associated with that node. This locking mechanism consists of four types of locks, r-locks, w-locks, p-locks, and x-locks, plus two additional variables, TimeStamp and FailTime. The r-locks are used for reading. Any number of processes may r-lock a node simultaneously. The variable Readers is a count of how many readers currently hold an r-lock on the node. The w-locks are used for changing weights, and the p-locks are used prior to holding an x-lock. Only one process at a time may w-lock or p-lock a node, but w-locking or p-locking a node does not exclude readers. However, when a node is p-locked, no new process will be granted an r-lock. The x-lock is used for making changes to a node. Only one process at a time may x-lock a node, and if a node is x-locked, it cannot simultaneously be r-locked, w-locked, or p-locked. A process attempting to obtain an x-lock must already hold a p-lock, and a process attempting to obtain a w-lock or a p-lock must already hold an r-lock. There are boolean variables W, P, and X, to indicate if there is a w-lock, p-lock, or x-lock on a node.

Figure 8 shows how locks are obtained and released, and Fig. 9 contains two variants of the well-known macros for await instructions. Note that when a process is attempting to obtain a w-lock or a p-lock, it is unnecessary to check if the node is already x-locked, since the process must already have an r-lock on the node and x-locks exclude r-locks. Similarly, when a process is attempting to obtain an x-lock, it is unnecessary to check if the node is already w-locked, p-locked, or x-locked, since the process must already have a p-lock on the node, and this excludes other w-locks, p-locks, and x-locks. The existence of the p-lock allows one of the processes interested in obtaining an x-lock on a node to get a semi-exclusive access. Any other readers then trying to get an x-lock will be unable to obtain a p-lock; they will fail and release any r-locks they have which could interfere with the process which was successful in obtaining the p-lock. Because of the order in which p-locks must be obtained (described below), the locks released will be r-locks on the node in question, on nodes further down, and on nodes to the right but on the same level. (Processes which attempt to obtain a w-lock and discover that a node is currently p-locked, also fail in this manner.) This intermediate type of lock is necessary to prevent deadlocks. In order to ensure that processes attempting to obtain an x-lock eventually obtain it, readers, attempting to obtain r-locks, must wait if the node is p-locked or x-locked. (They need not wait if the node is w-locked.) Notice that the p-lock described here corresponds somewhat to the w-lock in [21]. Our w-lock does not exist in their system. The most significant difference between their system and ours is that updaters use r-locks while searching; exclusive locks are only used at the actual location of an update or rebalancing operation. When possible, locking always proceeds in a top-to-bottom, left-to-right order, since this is an easy way to prevent deadlocks.

The top-to-bottom ordering is defined independently of the weights on the nodes, so the root is at the top-most level, its children are on the next level, etc. The only exception to this is that all of the leaves are defined to be on the same level, the lowest level. When an updater is searching for the correct location for an insertion or deletion, it will r-lock nodes in a top-to-bottom order, in a step-wise fashion. If the updater is attempting to insert, when it has r-locks on a parent node and its child, if the child is not a leaf, it can give up the r-lock on the parent node before making the next r-lock. If the updater is attempting to delete, it should hold r-locks on a parent, a child, and a grandchild, before giving up the r-lock on the top-most of these. After finding the appropriate leaf, the updater should first r-lock all necessary nodes, then convert all the r-locks to p-locks, and finally convert all p-locks to x-locks. The conversion from one set of locks to the next should occur in top-to-bottom, left-toright order. Thus, the locks on one level are all obtained before the locks on the next lower level, and within a level, the locks are obtained in a left-to-right order. The r-locks cannot all be obtained in this order since the predecessor leaf to the leaf in question must be r-locked. If the updater is unable to obtain this lock because it is currently p-locked or x-locked, it must give up all r-locks it has to the right of this node. A similar problem occurs with the deletemin operation. The updater first r-locks the pointer to the minimum element, then p-locks it, and finally x-locks it. Then, it r-locks the minimum element, then the parent of the minimum element, and then the grandparent of the minimum element. If any of these attempts are blocked by a p-lock or x-lock, the updater must give up all previous locks it has obtained. After obtaining these r-locks, the updater gets locks on the parent's right child and the minimum element's successor leaf (if that is not the parent's right child). These locks can then be converted to p-locks and finally to x-locks in a top-to-bottom, right-to left order.

A process doing a rebalancing operation gets a pointer to one of the nodes involved. It should first r-lock all of the nodes involved, and the first of these locks (and in the case of an operation dealing with a red-red conflict, also the second) must be obtained while coming from below. If the operation just involves moving weights around, the process should then w-lock all of the nodes, check that the operation is still appropriate and perform it if it is. If the operation involves changing the structure of the tree, the r-locks should be converted to p-locks, and the process should check that the operation is still appropriate. If so, the p-locks should be

#### AMORTIZATION RESULTS FOR CHROMATIC SEARCH TREES

	LOCK	RELEASE
<u>r-lock</u>	<pre>enter inc TimeStamp; time := TimeStamp if coming from below or from the left then     if P ∨ X ∨ (time ≤ FailTime) then     status := fail     else     status := success; inc Readers else     await (¬P ∧ ¬X) ∨ (time ≤ FailTime)     if time ≤ FailTime then     status := fail     else     status := success; inc Readers exit</pre>	enter dec Readers exit
<u>w-lock</u>	enter await ¬W fail on P set W; dec Readers status := <u>success</u> exit	enter unset W exit
<u>p-lock</u>	enter await ¬W fail on P set P; dec Readers status := <u>success</u> exit	enter unset P exit
<u>x-lock</u>	enter await Readers = 0 set X; unset P exit	enter if appropriate then FailTime := TimeStamp unset X exit

FIG. 8. Obtaining and releasing locks.

converted to x-locks and the operation should be performed. Other than the first one or two r-locks which must be obtained, there are only two possible exceptions to the rebalancing processors being able to obtain the necessary locks in a top-to-bottom, left-to-right order. The first exception involves r-locking leaves. If a process attempting to perform a rebalancing operation discovers that it has r-locked a leaf, it should consider itself as coming from below if it must r-lock more nodes. If it fails to r-lock one of these later nodes, it must give up all r-locks it has on leaves. The second exception is with the blacking operation. After performing a blacking operation, a rebalancing process must check to see if it has created a new red-red conflict. Thus, while obtaining the r-locks for the blacking operation, it must also lock the parent of the top node involved in the operation; this r-lock must be obtained by coming from below that node. After all of the necessary r-locks are obtained, this parent of the topnode must be w-locked so that its weight can be read, and the nodes actually involved in the operation must be p-locked and then x-locked, as with all other operations.

After a process has finished changing a node, if it has deleted the node or changed pointers to it, any processors waiting for the node should give up any locks they have and begin again. Otherwise, inconsistencies could occur in the data structure. If a process makes such changes, it will also change the variable FailTime in the node, to tell all processes which found the node before the change was completed to start over. This works as follows: There is a variable TimeStamp associated with the locking mechanism for each node. Initially this variable is zero. When a reader enters the critical region for that node for the first time, it increments TimeStamp and sets its local variable time to TimeStamp. When a process needs to set FailTime, it sets it to the current value of TimeStamp. All processes waiting for an r-lock on the node check if their value for time is less than or equal to FailTime, and, if it is, they release all locks and begin again. When a process which is rebalancing must release all locks and begin again, it also checks to see if there are other problems it could be working on instead.

In discussing deadlocks, we first argue, under the assumption that locks currently held will at some point be released, that deadlocks cannot occur at any one node. Afterwards, we return to the question of deadlocks involving processes waiting for conditions to be met at different nodes. Of

Implementation of <b>await</b> constructs		
await cond	await cond1 fail on cond2	
done := false	done := false	
repeat	enter	
$\mathbf{exit}$	repeat	
repeat until cond	exit	
enter	<b>repeat until</b> cond1 $\lor$ cond2	
if cond then done := true	enter	
until done	if cond2 then	
	status := <u>fail</u>	
	done := true	
	else if cond1 then	
	status := $success$	
	done := true	
	until done	
	$\mathbf{if} \operatorname{status} = \underline{\operatorname{fail}} \mathbf{then}$	
	exit	
	return	

FIG. 9. Macros used in the procedures for obtaining and releasing locks.

course, we assume that the protocol for obtaining and converting locks is followed.

There are four types of locks, each of which has one **await** statement. The rest of the code does not involve loops, so deadlocks, if any, must be due to **await** statements.

First, consider "**await** Readers = 0." Since a process which is trying to obtain an x-lock must already have a p-lock, processes trying to obtain an r-lock, after this p-lock has been obtained, have to wait. This means that the variable *Readers* cannot be increased. As we are assuming that locks are eventually released, *Readers* will eventually be decreased to zero. This means that no deadlock can involve the statement under consideration here.

Now, consider "**await**  $\neg W$  fail on P" (this statement appears both in the code for w-lock and for p-lock). If a process is waiting due to  $\neg W$  currently being false, then a w-lock is currently held by another process. This lock will eventually be released, so this statement cannot be involved in a deadlock.

The only deadlock situation not yet considered is the situation where two or more processes are trying to obtain an r-lock; i.e., they are waiting in "**await**  $(\neg P \land \neg X) \lor$  (time  $\leq$  FailTime)." However, such processes cannot mutually prevent each other from continuing since the locking variables appearing in the statement (*P*, *X*, and *FailTime*) are never set in the r-lock code. So, such processes cannot affect each other.

Defining a total order on the nodes in the data structure and requiring that locks are obtained successively according to this order is an effective method for avoiding deadlocks. The total order is defined implicitly since we require that all locks at one level must be obtained before locks at another level are requested. Similarly, locks at the same level must be requested and obtained from left to right. It is well known that such a scheme is deadlock-free (provided, of course, that deadlocks cannot occur at the individual nodes). It is possible that one process is waiting for another to release its locks, and that this process is waiting for a third process to release its locks, etc. However, since there are a finite number of processes working on the total order, the last process in this sequence must be able to proceed, and the eventual release of its locks will enable the process waiting for it to proceed, etc.

When some processes are allowed to obtain locks going against the ordering, deadlocks can easily occur. Therefore, we are restrictive in such cases. First, only r-locks can be requested in the "wrong" order. Second, if a process which requests r-locks in the wrong order discovers a node which is p-locked or x-locked, it gives up all locks which might interfere with the process having the p-lock or x-lock.

Since w-locks, p-locks, and x-locks are mutually exclusive, and since only x-locks require that no one reads, the only potential deadlock situation is the following: Assume that the processes A and B both have r-locks on nodes u and v. Now, A p-locks u and B p-locks v. At this point, they are both waiting for each other to give up the r-lock on the node they have p-locked such that the p-lock can be converted into an x-lock.

This scenario is not possible due to the protocol. Assume that u precedes v in the total ordering. Since both processes have a p-lock, they both want to make a change which requires exclusive access, so they have to convert all their r-locks to x-locks (via p-locks). This has to be done according to the total order, which means that process B is not working according to the protocol.

#### REFERENCES

- G. M. Adel'son-Vel'skiĭ and E. M. Landis, An algorithm for the organisation of information, *Dokl. Akad. Nauk SSSR* 146 (1962), 263–266, Russian; English translation, *Soviet Math. Dokl.* 3 (1962), 1259–1263.
- 2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "Data Structures and Algorithms," Addison–Wesley, Reading, MA, 1983.
- R. Bayer, Symmetric binary B-trees: data structure and maintenance algorithms, *Acta Inform.* 1 (1972), 290–306.
- R. Bayer and E. McCreight, Organization and maintenance of large ordered indexes, *Acta Inform.* 1 (1972), 173–189.
- J. Biswas and J. C. Browne, Simultaneous update of priority structures, in "Proceedings of the 1987 International Conference on Parallel Processing" (S. K. Sahni, Ed), pp. 124–131, Pennsylvania State Univ. Press, University Park, PA, 1987.
- R. D. Blumofe and C. E. Leiserson, Scheduling multithreaded computations by work stealing, *in* "Proceedings, Thirty-fifth Annual Symposium on Foundations of Computer Science," pp. 356–368, IEEE Computer Society Press, Los Alamitos, CA, 1994.
- J. Boyar, R. Fagerberg, and K. S. Larsen, Amortization results for chromatic search trees, with an application to priority queues, *in* "Algorithms and Data Structures—Fourth International Workshop, WADS'95" (S. Akl, F. Dehne, J.-R. Sack, and N. Santoro, Eds.), pp. 270–281, Lecture Notes in Computer Science, Vol. 955, Springer-Verlag, Berlin, 1995.

- J. Boyar and K. S. Larsen, Efficient rebalancing of chromatic search trees, J. Comput. System Sci. 49 (1994), 667–682.
- L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, *in* "Proceedings, Nineteenth Annual Symposium on Foundations of Computer Science," pp. 8–21, IEEE Comput. Soc. Press, Long Beach, CA, 1978.
- 10. J. E. Hopcroft, unpublished work on 2-3 trees, 1970.
- S. Huddleston and K. Mehlhorn, A new data structure for representing sorted lists, *Acta Inform.* 17 (1982), 157–184.
- D. W. Jones, An empirical comparison of priority-queue and event-set implementations, *Comm. ACM* 29 (1986), 300–311.
- D. W. Jones, Concurrent operations on priority queues, *Comm. ACM* 32 (1989), 132–137.
- J. L. W. Kessels, On-the-fly optimization of data structures, Comm. ACM 26 (1983), 895–901.
- H. T. Kung and P. L. Lehman, Concurrent manipulation of binary search trees, ACM Trans. Database Systems 5 (1980), 354–382.
- Y.-S Kwong and D. Wood, A new method for concurrency in *B*-trees, *IEEE Trans. Software Engrg.* 8 (1982), 211–222.
- K. S. Larsen, AVL trees with relaxed balance, *in* "Proc. 8th Intl. Parallel Processing Symposium," pp. 888–893, IEEE Comput. Soc. Press, Los Alamitos, CA, 1994.
- K. S. Larsen and R. Fagerberg, Efficient rebalancing of B-trees with relaxed balance, *Internat. J. Found. Comput. Sci.* 7 (1996), 169– 186.

- P. L. Lehman and S. B. Yao, Efficient locking for concurrent operations on B-trees, ACM Trans. Database Systems 6 (1981), 650–670.
- L. Malmi, "An Efficient Algorithm for Balancing Binary Search Trees", Technical Report, TKO-B84, Department of Computer Science, Helsinki University of Technology, 1992.
- O. Nurmi and E. Soisalon-Soininen, Uncoupling updating and rebalancing in chromatic binary search trees, *in* "Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems," pp. 192–198, ACM Press, New York, 1991.
- 22. O. Nurmi, E. Soisalon-Soininen, and D. Wood, Concurrency control in database structures with relaxed balance, *in* "Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems," pp. 170–176, ACM Press, New York, 1987.
- V. N. Rao and V. Kumar, Concurrent access of priority queues, *IEEE Trans. Comput.* 37 (1988), 1657–1665.
- Y. Sagiv, Concurrent operations on B\*-trees with overtaking, J. Comput. System Sci. 33 (1986), 275–296.
- B. Samadi, B-trees in a system with multiple users, *Inform. Process.* Lett. 5 (1976), 107–112.
- 26. H. Wedekind, On the selection of access paths in a data base system, in "IFIP Working Conference on Data Base Management" (J. W. Klimbie and K. L. Koffeman, Eds.), pp. 385–397, North-Holland, Amsterdam/New York, 1974.
- J. W. J. Williams, Algorithm 232: Heapsort, Comm. ACM 7 (1964), 347–348.