J. Parallel Distrib. Comput. 73 (2013) 434–449

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A concurrent red–black tree

Juan Besa *, Yadran Eterovic

Pontificia Universidad Católica de Chile, Chile

ARTICLE INFO

Article history:
Received 20 December 2011
Received in revised form
17 November 2012
Accepted 17 December 2012
Available online 26 December 2012

Keywords:
Concurrent data structure
Red–black tree
Algorithm design

ABSTRACT

With the proliferation of multiprocessor computers, data structures capable of supporting several processes are a growing need. Concurrent data structures seek to provide similar performance to sequential data structures while being accessible concurrently by several processes and providing synchronization mechanisms transparently to those processes.

Red–black trees are an important data structure used in many systems. Unfortunately, it is difficult to implement an efficient concurrent red–black tree for shared memory processes; so most research efforts have been directed towards other dictionary data structures, mainly skip-lists and AVL trees.

In this paper we present a new type of concurrent red–black tree that uses optimistic concurrency techniques and new balancing operations to scale well and support contention.

Our tree performs favorably compared to other similar dictionaries; in particular, in high contention scenarios it performs up to 14% better than the best-known concurrent dictionary solutions.

1. Introduction

Concurrent programming has transitioned from being a technique used in a few specialized areas, such as large computational problems or operating systems, to being a tool employed in all types of applications [12]. Many programmers are beginning to confront the difficulties associated with concurrent programming and to search for ways to simplify its use without impacting the performance gain of parallelizing an application. Concurrent data structures are one solution.

Concurrent data structures offer several advantages to application programmers. First, similarly to their sequential counterparts, they are easy to use. Second, they can manage all communication between threads – dealing with synchronization and solving contention – transparently to the rest of the program. The end user can solve many, if not all, concurrent issues by simply sharing the data structure. Third, because they are not developed for a specific program, the effort of developing efficient data structures is passed from the application programmer to the library developer, and is usually outweighed by the potential gain of using it. In fact, the increasing use of concurrent data structures shows how effective they are in solving concurrency issues.

However, there has been varied success in parallelizing data structures. For some – e.g., linked lists and queues – there are natural ways to achieve parallelism. But dictionaries (or ordered maps) are harder to parallelize. In a sequential environment they are typically implemented as balanced binary search trees of logarithmic depth such as AVL trees and red–black trees; but the concurrent versions of these trees were not considered efficient solutions until recently.

The most popular concurrent dictionary is the concurrent skip list, a type of augmented linked list that has expected logarithmic complexity for searches, insertions, and removals. Concurrent skip lists are preferred over balanced binary search trees because they do not require periodic rebalances. Rebalances affect concurrency negatively because they change the structure of the tree. In a concurrent environment a change to the structure of the tree affects all other threads operating in the vicinity of the change. Periodic rebalances thus cause bottlenecks and contention. Recent research into new techniques for traversing binary search trees has resulted in a concurrent AVL tree [2] that has better throughput than concurrent skip lists in low contention scenarios. But in high contention scenarios concurrent skip lists are still the most efficient solution.

We propose that concurrent red–black trees can improve the performance of concurrent dictionaries. Red–black trees are similar to AVL trees but have a more relaxed balance condition. Because of this they have a longer expected path length for traversals, but require fewer rotations than AVL trees. In particular red–black trees require $O(1)$ rotations to correct an imbalance, while AVL trees require $O(\log N)$ rotations, where N is the number of nodes in the tree. Thus, red–black trees have the potential to perform more efficiently than AVL trees in high contention scenarios.

We have developed a partially external concurrent red–black tree that scales well and supports contention; in particular, it outperforms concurrent skip lists in high contention scenarios.

* Correspondence to: Comandante Malbec 13334, Lo Barnechea, Santiago, 7690138, Chile.
E-mail address: juanbesa@gmail.com (J. Besa).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.12.010

By using hand-over-hand optimistic validation we created a strict concurrent red–black tree with fixed size region rotations and simple update operations: while it is easily recognizable from its sequential version, it supports a high level of concurrency. For single threaded access it behaves exactly as a red–black tree and at every time when no thread is inserting or removing an element it is guaranteed to fulfill the rules of a red–black tree.

The rest of the paper is organized as follows. Section 2 presents definitions and properties of red–black trees in general. Section 3 reviews related work. Section 4 describes our algorithm and details the new rebalancing operations introduced. Section 5 demonstrates the correctness of our algorithm. Section 6 presents experimental results: we compare the performance of our tree to those of a concurrent AVL tree and of the concurrent skip list from Java's concurrent library. Finally, Section 7 concludes with a discussion of the findings and presents possible future research.

2. Definitions

A red–black tree is a self-balancing binary search tree [3]. It maintains its balance by using rebalancing operations that rotate and/or recolor nodes to always fulfill the following five red–black rules:

1. Every node has a color, red or black. Depending on their color they are called red nodes or black nodes.
2. The root node is a black node.
3. Every simple path from the root to a leaf has the same number of black nodes; this number is called the *black depth* of the tree.
4. No red node has a red child.
5. Every null node is a black node.

The black depth of a tree is computed by adding the number of *black units* on any path from the root to a leaf. Each black node contributes one black unit, while red nodes do not contribute. A red node is transformed into a black node by adding a black unit to it; and a black node is transformed into a red node by removing its black unit.

A node n with two black units is called a *double-black* node (see Fig. 1) and is the owner of a *double-black imbalance*. Double black nodes occur when unlinking black nodes from the tree and are called *double-black imbalances*. Double black nodes also appear in sequential red–black trees: they are implicit in the recursive (traveling up the tree) rebalancing that may occur when removing a black node. Because of the locking scheme used in our tree the recursive upwards rebalancing must be done in several steps; this implies that the implicit double black node is now visible to other threads and thus requires a clearer definition. In sequential trees this is not required so they are generally not mentioned explicitly.

A pair of consecutive nodes, n and its child node c, is in a *red–red imbalance* if both n and c are red. The node nearest to the root, n, owns the imbalance. A node may own at most two red–red imbalances, one with each of its children.

The red–black rules guarantee $O(\log N)$ time complexity for *get*'s, *insertion*'s and *removal*'s, where N is the number of nodes in the tree. Insertions and removals are *update* operations, and the thread performing the update is an *updater*. An *imbalance* is any node, or parent–child pair of nodes, that breaks a red–black rule; when the tree has an imbalance we say it is *imbalanced*. To correct an imbalance an updater uses rebalancing operations – rotations and recoloring – to reestablish the rules. When rebalancing after removing a node we sometimes generate a double black node that contributes two black units to the black depth.

Red–black trees are extensively used as dictionaries and are the default symbol table for Java, C++, Python, BSD Linux and other systems [11]. As multithreaded applications become more widespread, concurrent dictionaries are becoming an important part of many programs. However, red–black trees, and binary search trees in general, have not had a relevant role as concurrent dictionaries because of the inherent difficulties and complexities associated with them. Instead, skip lists – a type of augmented linked lists with $O(\log N)$ expected performance – are the prevalent concurrent dictionaries [7]. By correcting the inefficiencies of concurrent red–black trees, we think they can also become practical concurrent dictionaries.

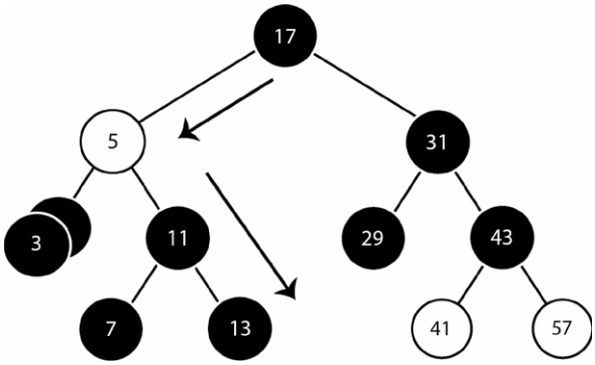

Fig. 1. Access patterns and nodes on a red–black tree. To search for node 13 we must follow the paths shown by the arrows. Node 3 is a double black node. Node 17 is the root and a black node. Node 41 is a leaf and a red node.

3. Related work

The most successful concurrent red–black tree is the chromatic tree proposed by Nurmi and Soisalon-Soininen [10] and also presented at PODS'91. In that proposal, the red–black rules are weakened so that insertions and removals are uncoupled from rebalances; this allows for fast updates but does not guarantee logarithmic depth.

Rebalances can be done gradually by a shadow process or when no urgent operations are present. Several rebalance processes can be working simultaneously on the tree.

To allow for removals they use an external tree, i.e., a tree where all the information resides in the leaves, while inner nodes only contain routing information. Thus, removals only affect leaf nodes and are greatly simplified. Removals may generate double black nodes (in their paper, edges), which contain two or more units of black. As we will explain in Section 4, our tree also has overweight nodes, but only allows a maximum of two units of blackness.

The rebalancing operations are simple and fast, but Nurmi and Soisalon-Soininen give no explicit upper bound on the complexity of their algorithm. These issues were addressed by Boyar and Larsen [1] who introduced new rebalancing operations which guarantee at most $O(\log n)$ rebalancing operations update, where n is the maximum size the tree could ever have, given its initial size and the number of insertions performed. Although the operations proposed by Boyer and Larsen significantly reduce the number of operations, they are costlier because they lock a larger neighborhood of nodes.

For concurrency control both chromatic trees and the new operations by Boyer and Larsen are lock-based techniques. Nurmi and Soisalon-Soininen use three kinds of locks: r-locks, w-locks – which function similarly to read–write locks – and x-locks, to maintain consistency throughout rotations and other updates. Threads lock only a small constant number of nodes by using *lock coupling*: a thread traversing the tree locks the child to be visited next, before releasing the lock on the node it is visiting now. Thus a thread holds at most two locks at the same time while traversing the tree. Updaters and the rebalancing thread lock only a few extra nodes when updating or rebalancing respectively.

```
 1 /*
 2  * attemptUpdateNode Updates or removes already found node If the new value
 3  * is null first transform the node and then attempt to remove it. If the
 4  * new value isn't null just change the stored value Object returned is the
 5  * old value
 6  */
 7 Value attemptUpdateNode(Value newValue, Node n, Node parent) {
 8   if (newValue == null) {
 9     // removal
10     if (n.value == null) {
11       // This node is already removed, nothing to do.
12       return null;
13     }
14     Value oldValue;
15     synchronized (n) {
16       if (isUnlinked(n.changeOVL)) // Something changed the tree
17         return SpecialRetry;
18       // Make route node then try to remove
19       oldValue = n.value;
20       n.value = null;
21     }
22     // Only remove nodes with at most one child
23     if (n.left == null || n.right == null) {
24       removeNode(n);
25     }
26     return oldValue;
27   } else {
28     synchronized (n) {
29       if (isUnlinked(n.changeOVL))
30         return SpecialRetry;
31       Value oldValue = n.value;
32       n.value = newValue;
33       return oldValue;
34     }
35   }
36 }
```

Fig. 2. Update and remove key. Removal of the actual node is left to a separate method, `removeNode`, due to its complexity (line 24). By setting the node's value to null, the node is transformed into a route node (line 20).

Hanke et al. [5] and Larsen [9] also present relaxed balanced red–black trees. These are similar to chromatic trees but differ in the rebalancing operations allowed on the tree. Hanke et al.'s updating threads do not remove nodes, but only mark them to be removed; the rebalancing thread does the actual removal. This technique also appears in Bronson et al.'s AVL tree (2010) (see below) and in our tree, although extended to internal trees.

Hanke et al. [4] compared the performance of these three trees and a standard red–black tree in a concurrent environment. They found that relaxed balanced red–black trees outperform standard trees and that the three relaxed trees perform similarly. The most important difference was in the quality of the rebalancing where the chromatic tree, performed best. In all cases the number of rotations per update is $O(1)$.

Recently, Bronson et al. [2] introduced a type of concurrent AVL tree, a type of binary search tree with $O(\log N)$ updates, but which may have to apply $O(\log N)$ rotations to rebalance the tree following an update. They use a new technique for traversing the tree, called *hand-over-hand optimistic validation*, which does not require acquiring locks while traversing the tree. This reduces the number of locks that must be held, but introduces the possibility of backtracks if a rebalance interferes. Readers are thus invisible, i.e. never delay, to updaters and, because updaters also use hand-over-hand optimistic validation, they can rebalance the tree *on the fly* after doing an update and must only hold locks in a small critical region.

Hand-over-hand optimistic validation is adapted from software transactional memory (STM) but uses knowledge of the algorithm to reduce overheads and avoid unnecessary retries. The algorithm adds a version number to the nodes of the tree. When a node participates in a rotation its version number is increased; this is analogous to normal STM. A thread traversing the tree proceeds in the same manner as in lock coupling but uses the version numbers to validate its movements. If a version number changes, the last traversed link may be invalid, so the thread returns to the previous node. It then checks if that node is still valid: if it is, then it retries to advance down the tree; if not, it returns to the previous node in its path. A thread may have to back up several links – possibly returning to the root – if several rotations have invalidated nodes higher up in the tree.

Bronson et al. improved this by setting aside three bits to mark if the node is growing (moving nearer to the root), shrinking (moving further from the root), or is unlinked (the node has been removed from the tree).

When searching, a thread begins with a possible range of $(-\infty, +\infty)$ and each move to a child node reduces the range. Moving to the left child increases the lower bound while moving to a right child reduces the upper bound; i.e. each node is the root of a subtree whose range is defined by the path from the tree root to it. A search is still valid if the current node grows. When a node grows the range of its subtree also grows and the node looked for will still be inside that range, so the search is still valid. If a node shrinks its range also decreases so the node looked for may be incorrectly removed from the range and the traversal is invalid.

The third bit serves to implement partially external trees. In an internal binary search tree data are located in the inner nodes of the tree. Removing a node with two children requires first finding the node's successor. This is difficult to do in a concurrent tree because the successor may be many links away, so relaxed trees either do not support removals or are external trees. Bronson et al. presented *partially external trees*, which simplify removing nodes with two children by changing them to *route nodes* – nodes that have no value but contain routing information – and keeping them in the tree. Similarly to Hanke's tree, any thread that rebalances the

tree later on unlinks the nodes thus marked for removal. The nodes are unlinked in the simpler cases if they become a leaf or have only one child. Bronson et al. showed that using these route nodes do not increase the tree size significantly.

The main contributions of our tree are the following:

- We use both hand-over-hand optimistic validation and partially external trees and apply them to a concurrent red–black tree.
- We define new rebalancing operations and define the concept of double black nodes. We limit double black nodes to hold a maximum of two black units to improve the efficiency of rebalancing operations.
- Our tree is the first partially external red–black tree.
- It is also the first concurrent red–black tree to not use relaxed balance, instead the updating thread corrects any imbalance it introduces in the tree.
- Our tree cannot guarantee $O(\log N)$ updates and searches when there are several updates occurring concurrently due to scheduling issues and potential backtracks. It is guaranteed to fulfill $O(\log N)$ searches if there are no updates being applied to the tree. In the presence of updates it will generally have $O(\log N)$ updates and searches if they do not interfere with each other.

4. A new algorithm to rebalance concurrent red–black trees

Updates – insertions and removals – affect the tree structure, possibly causing imbalances and subsequent rebalancing operations. An update causes at most one imbalance associated to a specific node in the tree; we say that the updater *owns* that imbalance.

The updater thread must solve the imbalance it owns by applying rebalancing operations. If at any time, due to its own rebalancing operations or to those of a different updater, its node's imbalance is fixed, then the update concludes.

At any given moment, if there are updaters working on the tree, one of the five rules may be broken. Our tree requires that if there is an imbalance, it must be owned by a particular updater. So if there are no updaters working on the tree, then the tree fulfills all the red–black rules.

Insertions are handled in the same way as in any binary search tree. The node inserted is always red, so if the node's parent is red it produces a red–red imbalance.

Removing a key k is accomplished in two steps, as shown in Fig. 2. First, the node with the key is transformed by setting its value to *null* (line 20). This effectively removes k from the tree: the node still holds the key but its value is marked as *null*, so the node only contains routing information; therefore, it becomes a *route node*. Second, the updater attempts to remove, or *unlink*, the node from the tree (line 24).

To remove a node with two children we must first swap it with its successor. Unfortunately the successor may be many links away so it is very costly to do in a concurrent tree; therefore, we do not support this removal. Thus, if the new route node has two children, then the node is not unlinked and the removal is concluded. If it has only one child, then the child is connected to the route node's parent, thus unlinking the route node. If the route node is a black leaf, then the updater first rebalances any imbalance that would be caused by unlinking the node, and then unlinks it.

4.1. Rebalancing operations

This section presents the operations that an updater must execute to rebalance the tree. We first describe the operations to solve red–red imbalances (due to insertions); then in Section 4.2 the operations to solve double black imbalances (due to removals), and the operations for when red–red imbalances collide with double black imbalances in Section 4.3. Finally in Section 4.4 we describe the two root balancing operations.

A rebalance is always as follows:

1. Determine that node n has an imbalance.
2. Acquire the locks to n's grandparent, first, and n's parent, next.
3. Assess the vicinity of the node to determine which operation to use, acquiring more locks if necessary.
4. Apply the operation, which may include performing a rotation; the updater already holds all necessary locks (from step 3).
5. Release all locks.
6. The applied operation may have transferred the imbalance to another node; evaluate it.

Acquiring locks only affects other threads that are concurrently applying rebalances in the same area. Traversals are not affected; these must only worry about rotations. As imbalances propagate closer to the root the contention for locks increases, as does the possibility of resolving more than one imbalance with a single operation. Because of this, delays due to waiting for locks are longer, but actual rebalancing time is reduced for many updates.

To change the color of a node we require that its parent be locked. This means that at any moment only one updater can change a node's color.

In the next sections we use the following labels (as seen in Fig. 3):

n: the node that owns the imbalance
p: n's parent
s: n's sibling; the other child of p
g: n's grandparent; the parent of p
c: n's red child in a red–red imbalance.

There are two types of rotations, single rotations and double rotations. Double rotations can be broken down into two sequential single rotations (as in Fig. 3) but, it is more efficient to do them in one atomic action. Double rotations are also costlier than single rotations because they require that one extra node, n's child c, be locked. In our design we prefer executing a single rotation to a double rotation (see line 90 in Fig. 7).

Note for all operations and rotations we only present those where n is the left child of p (Figs. 5 to 13); all operations – except for the root operations – have a symmetrical version for when n is the right child of p.

4.2. Fixing red–red imbalances

To insert a new node we attach a red node to a leaf. If the (old) leaf was also red this generates a red–red imbalance. The thread performing the insertion must then apply rebalancing operations until the imbalance is solved. In a red–red imbalance the node that is specific to this imbalance, or n, is the red parent.

We have 6 different cases to rebalance the tree. In cases 1–3 we show the rebalancing operations to correct red–red imbalances. We also include new operations for cases where red–red imbalances and double black imbalances collide because the owner of the imbalance, i.e., n is the red–red imbalance. In cases 4–6 we present the operations for correcting double black imbalances.

The following three cases are for when n is the owner of a red–red imbalance. In the figures of this and the following section node n is the node with key 5; black nodes are black, red nodes are white and double black nodes are represented by two overlapping black nodes.

Case 1: *n is red and s is also red.*

If the sibling s of n is red, then we can push the redness up as in Fig. 5.

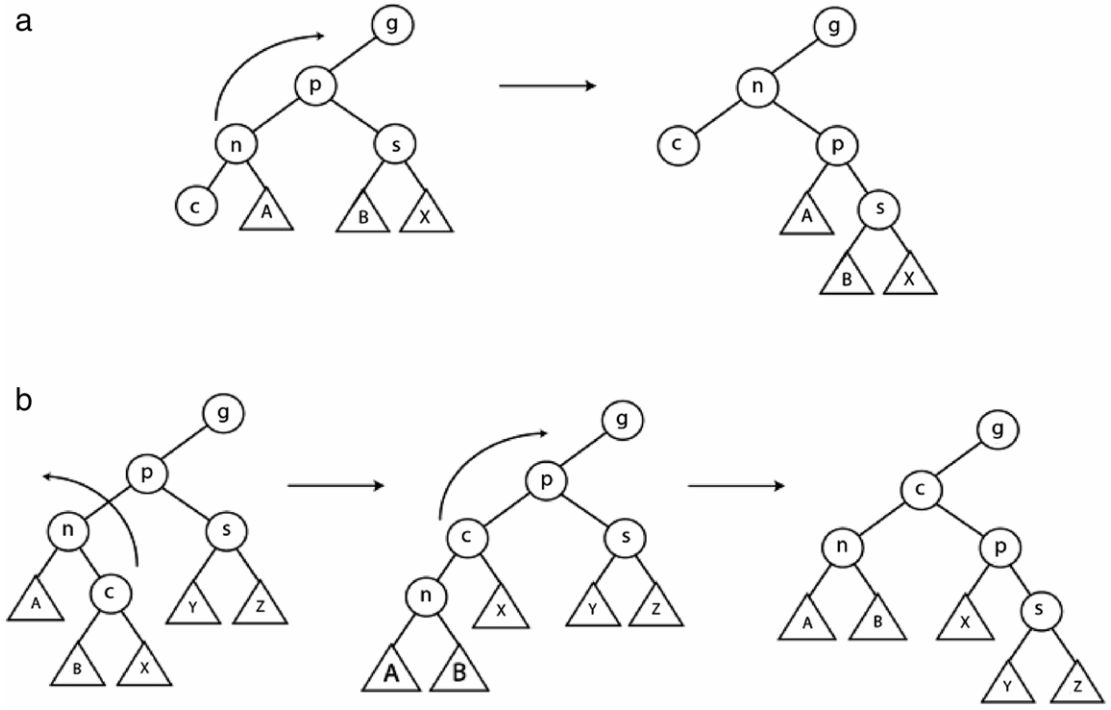

Fig. 3. Rotation in a red–black tree: (a) Single rotation, (b) Double rotation. Nodes g, p, and n must always be locked. For a double rotation c must also be locked.

If the grandparent g of n is also red, then by pushing red we may generate a new red–red imbalance higher up in the tree. The updater then resolves the new imbalance similarly (Line 47 in Fig. 4).

Because the operation is symmetric it also resolves any red–red imbalance centered on s. Thus, if any other thread was trying to rebalance n or s, its imbalance is resolved and its update concludes. So, a push red may resolve up to four red–red imbalances, although it is sufficient for one red–red imbalance to be present for the operation to be executed.

Case 2: *n is red and s is black and the outer child of n is red.*

In Fig. 6, operation (b) is a new operation. It extends case (a) to consider what happens if both of n's children are red. In that case applying operation (a) would resolve the imbalance at n but would generate a new red–red imbalance at p. To solve this case the updater first rotates and then pushes red as one operation. Because the updater already holds the necessary locks, this solves an extra imbalance at no extra cost.

Finally, because the red–red imbalance is located at n, and not at any of its children, both updaters will try to use operation (b). The updater that does not execute the operation has concluded its update.

Operation (c) is an example of a red–red imbalance that collides with a double black imbalance: both imbalances can be solved with one rotation.

Case 3: *n is red and s is black and the inner child of n is red.*

A double rotation is similar to a single rotation but occurs because c is the inner node of n and the outer child of n is black (Fig. 8). A double rotation involves more nodes than a single rotation, so it affects a greater number of traversing threads and is costlier than a single rotation.

Red clusters

In a red–red imbalance if p is also red then n cannot be rebalanced until the red–red imbalance between p and n is corrected. This situation occurs when many threads insert similar keys at the same time. If more nodes are inserted in the same area, say at n or c, they will also be in a red–red imbalance and the updaters will not be able to advance until the upper imbalances are resolved. This generates a zone of only red nodes, which we call a red cluster, in which many threads cannot advance.

To reduce the effect of red clusters, if in any red–red imbalance p is also red then the updater will first try to rebalance p and subsequently return to rebalance n. This prevents updaters from retrying to balance n, a node that cannot be rebalanced yet.

When a red cluster occurs the updaters that are working on lower nodes travel up the cluster until they reach the uppermost red–red imbalance. The first updater to acquire the locks of the uppermost imbalance resolves that imbalance. The other updaters wait until that updater resolves the imbalance, and then begin traveling back to their original node, once again trying to rebalance all the nodes in the path back. When an updater returns to the node it inserted, it is sure that it can rebalance the node (if necessary). During these operations locks serve as concurrent barriers so only updaters holding a lock will be running.

We now present the cases to remove double black imbalances. As before the owner of the imbalance is always node 5 and we present only the cases where n is the left child of p.

4.3. Fixing double black imbalances

This Double black imbalances occur when it is necessary to remove a black unit from a node and pass it to a different node. If that node is black it now has an extra black. There are two situations where this is necessary.

First, before unlinking a black route leaf, it must be transformed into a red node by removing a black unit from it. To do this, the updater first rebalances, and then atomically unlinks the node. To other updaters this is an atomic action, so at no moment rule 3 is broken (Fig. 11).

Second, a double black rebalancing operation may produce a double black node. These are nodes that contain an extra unit of blackness and as such break rule 1. They represent imbalances due to removals and correspond to node n, the node associated with the imbalance.

```
37 /** Pushes red (case 1-a and 1-b)
39  */
40 Node pushRed(Node g, Node p,Node n) {
41   p.left.color = p.right.color = NodeColor.Black;
42   if (p.color == NodeColor.DoubleBlack) {
43     p.color = NodeColor.Black;
44     return n;
45   } else { // p is black
46     p.color = NodeColor.Red;
47     return g != rootHolder ? g : p;
48   }
49 }
50
51 /*
52  * For a red-red imbalance determine if we must rotate or push red
53  */
54 Node rotateOrPushRed(Node n) {
55   Node p = n.parent;
56   Node g = p.parent;
57   if (g == null) {
58     return blackenRoot(n);
59   } else if (color(p) == NodeColor.Red) {
60     fixColorAndRebalance(p); // For red clusters
61     return n;
62   } else {
63     synchronized (g) {
64       if (isUnlinked(g.changeOVL) || p.parent != g)
65         return n;
66       synchronized (p) {
67         // Several sanity checks
68         if (isUnlinked(p.changeOVL) || n.parent != p)
69           return n;
70         if (n.color != NodeColor.Red || p.color == NodeColor.Red)
71           return n;
72         // Determine between case 1 or case 2 and 3
73         if (color(p.left) == NodeColor.Red
74             && color(p.right) == NodeColor.Red) {
75           return pushRed(g, p, n); // Case 1
76         } else if (p.left == n)
77           return rebalance_to_Right(g, p, n); // Case 2 and 3
78         else
79           return rebalance_to_Left(g, p, n); // Case 2 and 3
80       }
81     }
82   }
```

Fig. 4. The methods `rebalance_to_left` and `rebalance_to_right` (line 77 and 79) are symmetrical and consider both single and double rotations. Method `fixColorAndRebalance` in line 49 forces the updater to travel up red clusters and resolve higher red–red imbalances. Because the updater already holds locks to g and p, in line 70 it does not try to rebalance p and simply backtracks. In line 57 if g is null then p is the root holder and n is the root line and must be painted black.

Figs. 11–16 only talk of double black nodes, but in all cases n may be considered the route node that we want to remove.

Nodes with more than two units of black – for example triple black nodes – are not allowed because they are extreme cases and very difficult to rebalance. It is preferable forbid the removal of double black nodes or the addition of a black unit to a double black node. Fortunately triple (or higher) black nodes are very uncommon because they require a very specific pattern; therefore they are better handled by not allowing them to exist.

Case 4: *n is double black and s is black with black children.*

This operation is very similar to push red. It is important because it is the most common method for a double black node to enter the tree. When eliminating a black route leaf, if s is also a black leaf then, due to rule 5, both of its null children are also black. In this case the updater pushes black, removing n and painting s red (Fig. 9).

Case 5: *n is double black and s is red.*

When s is a red node there is no operation to immediately fix the imbalance (Fig. 10). Instead, it must be done in two operations. If p and the inner child of s are black, then rotating s over p ensures that the next rebalancing operation will correct n. The inner child of s must be locked before the rotation so it can participate in the next

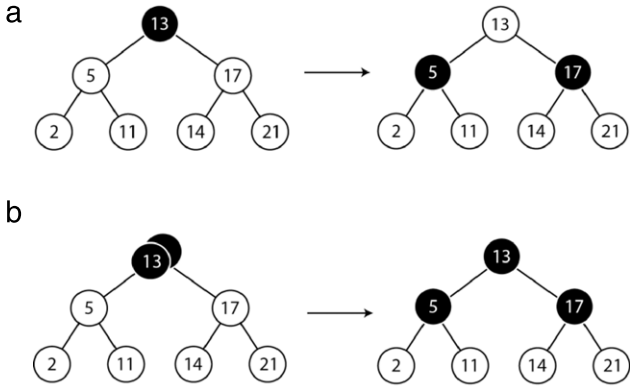

Fig. 5. (a) Case 1-a: Push red, p must be red. (b) Case 1-b Push red and p is a double black node. In both cases only one of the four bottom nodes must be red to do the operation.

operation. Both operations are "concatenated", which corrects the imbalance.

If the inner child of s is also red, then n's updater cannot correct it, but must wait for the red–red imbalance to be corrected first. In this case when eliminating a node it is preferable to cancel the unlink instead of waiting for a more favorable state.

Case 6: *n is double black and s is black with a red child.*

When s is a black node and has a red child, the updater can use the red node to correct the double black node. In both single and double rotations special care must be taken if p is double black. In those cases the imbalance at p is corrected, but the imbalance is transferred to s and must be corrected by the updater. The thread that was updating p will have finished its rebalance because the imbalance is no longer associated to p but to s.

Rebalancing operations involving red–red imbalances and double black imbalances

When red–red imbalances collide with double black imbalances it helps that the double black node is higher up in the tree than the red–red imbalance. If the double black is higher up in the tree – more specifically it is at p – it means that it has removed black units, creating red nodes lower in the tree. To correct both issues it is only necessary to do a red–red operation (as explained in Section 4.2) and distribute the extra black unit to both branches.

It is a different case when the two imbalances are siblings. In this case the double black node has not generated any of the red nodes that participate in the red–red imbalance, but it has left a "wake" of red nodes on its own branch. Rebalancing the tree is much harder because both branches are very different. One lacks nodes while the other has double black nodes so more than one rotation must be done to balance both branches. This is why an updater correcting a double black node with a red sibling that is participating in a red–red imbalance waits for the red–red imbalance to be corrected before proceeding.

4.4. Root operations

Root operations are the simplest of all rebalancing operations and there are only two cases: the root is red, and the root is double black. In both cases it is only necessary to change the root to black. Because the root should always be black it can be painted black at any moment without breaking any of the red–black rules.

5. Correctness

In this section we prove that our algorithm finishes and correctly rebalances a red–black tree. Our proof is adapted from the proof given by Nurmi and Soisalon-Soininen [10] for chromatic trees. We show every imbalance is owned by some updater. We say that an updater is working on a CRBT if it is applying an update to the tree, i.e., inserting or unlinking a node, or if it is rebalancing the tree after applying an update. In Section 5.1 we show that our tree fulfills the necessary conditions to correctly operate in a concurrent environment. Then Section 5.2 proves that our rebalancing operations eventually rebalance the tree.

5.1. Concurrency correctness

Deadlock freedom: The structure of the tree is used to ensure deadlock freedom. Locks must always be taken in a top down manner; if a thread has locked a node, it can only lock nodes that are lower in the tree. Rebalancing operations need to hold locks to change parent–child links. Because the tree does not have a loop and rebalancing operations safely manage locks, no deadlock cycle can occur and the tree is deadlock free.

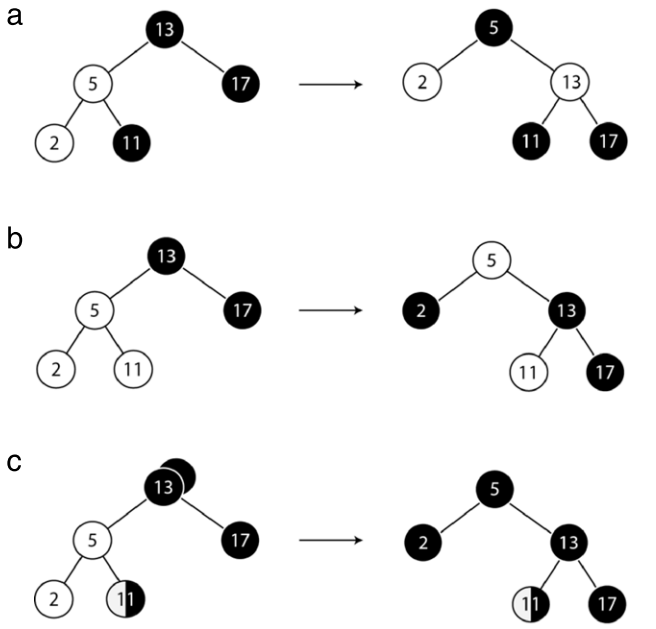

Fig. 6. In all cases the node with key 17 can be both black or double black because it does not interfere with the rotation. (a) Case 2-1: Single rotation (b) Case 2-b: Single rotation with double red, after applying Case 2-a push red. (c) Case 2-c Single rotation with p double black.

Linearizability: Because we use the same technique for traversals and rotations as Bronson et al. [2] we will not include the proof for linearizability here; this leaves only the linearizability of color changes to the tree. To change a node's color its updater must hold its parent locks, so all color changes are serialized for other updating threads.

5.2. Operation correctness

We want to prove that if a series of concurrent updates are applied to a valid partially external red–black tree, then at any time thereafter and when there are no updaters working on the tree, the tree is a valid partially external red–black tree. To do this we must prove two things:

1. Every imbalance that appears in the tree is accounted for.
2. Each of these imbalances is solved by some combination of the operations described above.

While there are updaters working on the tree, we will call it a CRBT. Thus a CRBT is a tree that may have one or more imbalances due to the updates being applied.

Informally, we shall show that whenever a rebalancing operation is applied to a CRBT, the operation will produce a tree that is closer (defined below) to being balanced. First, we shall prove statement (1) above by showing that each imbalance is associated to a specific updater (Theorem 1) that will remain updating the tree until the imbalance is removed. Then we demonstrate statement (2) by showing that the red–red imbalances can be removed without creating new double black imbalances (Theorem 2). Next, we address double black imbalances (Theorem 3) and cases where both red–red and double black imbalances are solved by one rebalancing operation (Theorem 4). Finally we address the special case where we must first rotate the red sibling of a double black imbalance and then atomically apply a different operation (Theorem 5).

Lemma 1. *Given a red–black tree T and a CRBT T' produced by the actions of u updaters concurrently working on T; then T' has at most u imbalances.*

```
 84 /*
 85  * Case 2 and 3 when n is a left child. A symmetric method exist for when n
 86  * is a right child. At this point both g and p are locked
 87  */
 88 Node rebalance_to_Right(Node g, Node p, Node n) {
 89   synchronized (n) {
 90     Node nL = n.left, nR = n.right;
 91     NodeColor nLcolor = color(nL), nRcolor = color(nR);
 92     NodeColor pColor = p.color;
 93
 94     if ((nLcolor != NodeColor.Red && nRcolor != NodeColor.Red)) {
 95       return n;
 96     } else if (nLcolor == NodeColor.Red && nRcolor == NodeColor.Red) {
 97       rotateRight_nl(g, p, n, nR);
 98       // Case 2-b and 2-c
 99       if (pColor == NodeColor.DoubleBlack) {
100         //Case 2-c Rotate and push red
101         p.color = NodeColor.Black;
102         n.color = NodeColor.Black;
103         nL.color = NodeColor.Black;
104         return p;
105       } else {
106         // Case 2-b Rotate
107         nL.color = NodeColor.Black;
108         // If n is now root return it
109         if (g != rootHolder) {
110           return g;
111         } else {
112           return n;
113         }
114       }
115     } else if (nLcolor == NodeColor.Red) { //Prefer single...
116       rotateRight_nl(g, p, n, nR);
117       // Case 2-a and 2-c Left-Left Situation. We rotate right.
118       if (pColor == NodeColor.DoubleBlack) {
119         p.color = NodeColor.Black;
120         n.color = NodeColor.Black;
121         nL.color = NodeColor.Black;
122       } else {
123         n.color = NodeColor.Black;
124         p.color = NodeColor.Red;
125       }
126       return p;
```

Fig. 7. Red–red conflict resolution for when *n is the left child of p*. In line 115 we prefer doing a single rotation to a double rotation.

Proof. Each insertion in the tree produces at most one new red–red imbalance, at the inserted leaf's parent, and does not produce any double black imbalances. Vice versa, unlinking a node produces at most one new double black imbalance, also at the parent of the removed leaf, but no new red–red imbalance. Thus each updater adds at most one new imbalance to the tree during its initial update.

A simple case-by-case analysis shows that each rebalancing operation maintains or reduces the number of imbalances in the tree. Thus each updater adds at most one new imbalance to the tree and applying rebalancing operations never increases the number of imbalances. □

Theorem 1. *Let T be a CRBT with imbalances; then each imbalance is owned by at least one updater.*

Proof. An update generates at most one imbalance that is owned by the updater doing the operation. Thus an updater owns all imbalances due to inserting or unlinking a node.

Likewise each rebalancing operation solves the imbalance for which it is applied and generates at most one new imbalance owned by the same rebalancing updater. The updater applying the transformation resolves the imbalance it owns and, if a new one is generated, acquires at most one new imbalance. It follows that every updater owns at most one imbalance and every imbalance is owned by at least the updater that generated it.

A simple case-by-case examination demonstrates that for each rebalancing operation at least one imbalance is resolved and at most one new imbalance is generated. Thus, the updater applying the transformations is always associated with at most one imbalance. Likewise every insertion or removal produces at most one imbalance associated with the updater doing the operation. An updater, after inserting or unlinking a node, is associated with one imbalance in the tree. By applying a rebalancing transformation it resolves at least one imbalance, but may also simultaneously resolve other imbalances in the vicinity, such as in Fig. 2(a). □

Lemma 2. *Let T be a CRBT and assume any one of the transformations defined in Section 3 is applied to T. Then the transformed T' is either a CRBT if there are still imbalances in the tree; or a red–black tree if the transformation resolved the last remaining imbalance and there are no updaters working on the tree.*

Proof. If there are no imbalances then the tree is a valid red–black tree. If the tree now has one or more imbalances then by Theorem 1

```
127     } else // ... to a double rotation
128     {
129       // Case 3-a 3-b 3-c Left-Right. Double rotation
130       synchronized (nR) {
131         rotateRightOverLeft_nl(g, p, n, nR);
132         if (pColor == NodeColor.DoubleBlack) {
133           // Case 3-c
134           nR.color = NodeColor.Black;
135           n.color = NodeColor.Black;
136           p.color = NodeColor.Black;
137           return p;
138         } else if (nRcolor == NodeColor.Red
139             || nLcolor == NodeColor.Red) {
140           // Case 3-b Rotate and push red
141           n.color = NodeColor.Black;
142           // If n es root return it
143           if (g != rootHolder) {
144             return g;
145           } else {
146             return nR;
147           }
148         } else {
149           // Case 3-a
150           nR.color = NodeColor.Black;
151           p.color = NodeColor.Red;
152           return p;
153         }
154       }
155     }
156   }
157 }
```

Fig. 7. (*continued*)

each imbalance has an updater associated with it and so T' is a valid CRBT. □

Lemma 3. *If T is a CRBT, then T has at least one node on which a rebalancing transformation can be applied.*

Proof. There are three cases in which a node in an imbalance cannot be balanced. We will show that in each case there must exist a different node that can be balanced:

Case 1. An updater owns a red–red imbalance in which the parent of n is also red. In this case the tree must contain a node closer to the root (it could be the root itself) that is not red where a rebalancing transformation can be applied.

Case 2. An updater owns a double black imbalance in which the sibling of n is red and is participating in a red–red imbalance. In this case the updater must wait for that imbalance at n's sibling to be resolved. Since that imbalance is red–red it falls into case 1.

Case 3. An updater must push black, but the parent of n is already double black. Similarly to case 1, the tree must contain a node closer to the root that is double black where a rebalancing operation can be applied or one of the previous cases apply.

Before we state our next lemma we will define some terms to help us characterize the balance state of a CRBT.

Let T be a CRBT tree and n a node in T. By $outside(n)$ we denote the number of T's nodes that are not contained in the subtree rooted at n. Let $c = (n, v)$ be a red–red imbalance in T and n its leading node. The *red–red distance* of c, $rd(c)$, is defined by $rd(c) = outside(c)$ and the *total red–red distance* in T, $rd(T)$, by

$$rd(T) = \sum_{c \in R(T)} rc(c),$$

in which $R(T)$ is the set of red–red imbalances of T.

If n is an double black node, then the *double black distance* $od(n)$ of n is defined by $od(n) = outside(n)$ and the *total double black distance* of T, $od(T)$ is defined by

$$od(T) = \sum_{n \in N(T) \text{ and } w(n)=2} od(n),$$

where $N(T)$ is the set of nodes of T.

We shall characterize the balance of tree T by the quadruple $(o(T), od(T), r(T), rd(T))$, in which $o(T)$ is the number of double black imbalances in T, $r(T)$ is the number of red–red imbalances in T, and $od(T)$ and $rd(T)$ are as above. Let $<$ denote the alphabetical order of the quadruples (i.e., $(a', b', c', d') < (a \cdot b \cdot c \cdot d)$ if $a' < a$ or $a' = a$ and $b' < b$ or $a' = a$ and $b' = b$ and $c' < c$ or $a' = a$ and $b' = b$ and $c' = c$ and $d' < d$). We say that a tree T is *closer* to a red–black tree T, denoted by $T' < T$, if

$$(o(T'), od(T'), r(T'), rd(T')) < (o(T), od(T), r(T), rd(T)).$$

Notice that the smallest elements in this relation are red–black trees.

We show now that whenever a rebalancing transformation is applied in CRBT T, it will result in tree T', with $T' < T$. □

Theorem 2. *Let T be a CRBT with at least one red–red imbalance, and let T' be a CRBT obtained from T by applying one of the transformations of case* 1-a*, case* 2-a*, case* 2-b*, case* 3-a*, or case* 3-b *from Section 4. Then $T' < T$.*

Proof. When one of the operations is applied to n (the leading node of the imbalance $i = (n, v)$ to be removed) then the number of red–red imbalances will be decreased in the subtree S originally rooted at n.

One imbalance with leading node n is removed and no new imbalance are generated inside the subtree; so, within the subtree S the number of red–red imbalances has decreased.

Some red–red imbalances inside S may be pushed lower down the tree in operations of case 2-a, 2-b, 3-a and 3-c, but whenever

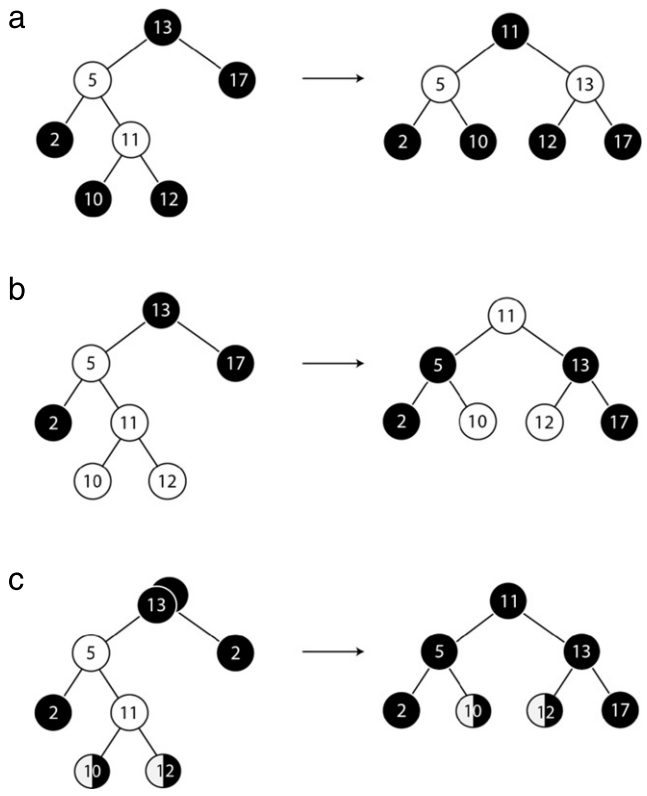

Fig. 8. (a) Case 3-a: Double rotation, (b) Case 3-b: Double rotation with double red, (c) Case 3-c: Double rotation extra black. In all cases c must also be locked. In case (b) it is only necessary for one of nodes 10 or 12 to be red.

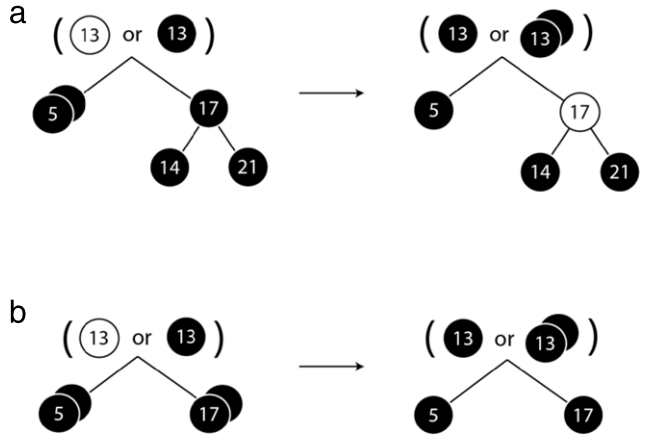

Fig. 9. (a) Case 4-a: Push black, (b) Case 4-b: Push black with a double black sibling. Because node 17 must have two black children it must also be locked to prevent possible red–red conflicts.

this occurs for an imbalance $i' = (n', v')$ the number of nodes in subtree rooted at n' does not change, so its red–red distance does not change.

In transformations case 1-a, case 2-b and case 3-b, if the grandparent of the leading node is red, when the parent of the leading node becomes red this generates a new red–red imbalance. But this imbalance has a smaller red–red distance than the removed one. Denote by S' the tree obtained from S by the transformation. If the parent of the root of S' is red we have one new red–red imbalance. However the red–red distance of this imbalance is smaller than the red–red distance of the removed

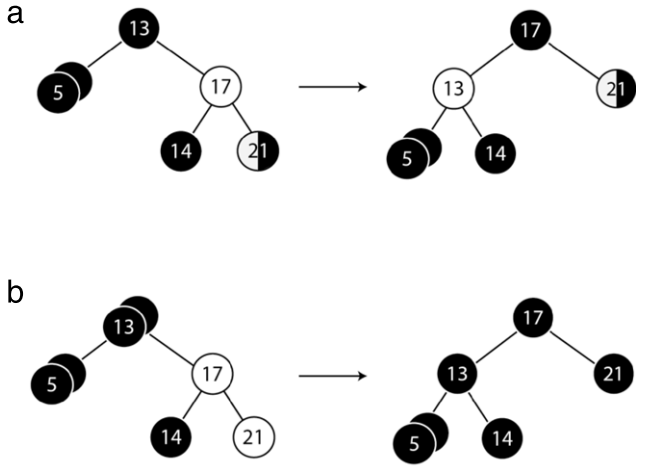

Fig. 10. (a) Case 5-a: Red sibling, (b) Case 5-b: Red sibling with a red right nephew and a double black parent.

imbalance. Hence we conclude that either $r(T') < r(T)$ or $r(T') = r(T)$ and $rd(T') < rd(T)$.

As to double black imbalances, it is clear that no new imbalances are created. Some double black imbalances may be pushed lower in the tree, but for every double black node n in T, $od(n)$ is maintained. Thus $o(T') \leq o(T)$ and $od(T') \leq od(T)$.

We can conclude that $(o(T'), od(T'), r(T'), rd(T')) < (o(T), od(T), r(T), rd(T))$ and thus $T' < T$. □

Theorem 3. *Let T be a CRBT with at least one double black imbalance, and let T' be a CRBT obtained from T by applying one of the transformations of case* 4 *or case* 6 *from Section* 4. *Then $T' < T$.*

Proof. The transformation of case 4, case 6-1.a and case 6-1.b either removes the imbalance or pushes it higher up in the tree. Thus in this case $o(T') \leq o(T)$ and $od(T') < od(T)$. □

Theorem 4. *Let T be a CRBT with at least one double black imbalance and one red–red imbalance, and let T' be a CRBT obtained by applying one of the transformations of case* 1-b, 2-c, *or* 3-c *from Section* 4. *Then $T' < T$.*

Proof. In cases 1-b, 2-c and 3-c both imbalances are resolved so it is clear that $o(T') < o(T)$ and $r(T') < r(T)$ so $T' < T$. □

Theorem 5. *Let T be a CRBT with at least one double black imbalance, and let T' be a CRBT obtained from T by atomically applying case* 5 *followed by one of the cases* 4-a, *case* 6-1.a, *or case* 6-2.a. *Then $T' < T$.*

Proof. One of operations of case 4-a, 6-1.a or 6-2.a can always follow the transformations of case 5. All of them correct the imbalance and do not generate a new imbalance. Some other imbalances may be pushed lower in the tree but for every double black node c in T, $od(c)$, and for every red–red imbalance c' in T, $o(c')$, is maintained. Thus by atomically applying an operation of case 5 and 4-a or atomically applying an operation of case 5 and 6-1.a or atomically applying an operation of case 5 and 6-2.a on the double black node n $o(T') < o(T)$. Thus $T' < T$. □

The following theorem tells us that we do not need to fix the order in which the updater threads remove imbalances.

Theorem 6. *Given a CRBT, any sequence of rebalancing transformations that is long enough modifies the tree into a red–black tree.*

```
158 /*
159  * Tries to remove a black leaf. The leaf should be a route node.
160  * Invokes the correct rebalancing case. If the rebalance is successful
161  * it then removes the node.
162  */
163 boolean attempRemoveBlackLeaf(Node n, Node parent, Node g) {
164   Node doubleBlackNode = null;
165   synchronized (g) {
166     if (parent.parent != g || isUnlinked(g.changeOVL))
167       return false;
168     synchronized (parent) {
169       if (n.parent != parent || isUnlinked(parent.changeOVL))
170         return false;
171       // Link g -> p -> n is locked. Link p -> s is also locked.
172       // Colors of g,p,n,s are also locked.
173       synchronized (n) {
174         NodeColor nColor = n.color;
175         Node nL = n.left;
176         Node nR = n.right;
177
178         // ... Handle other removal cases
179
180         char nDir;
181         if(parent.left == n)
182           nDir = Left;
183         else
184           nDir = Right;
185         Node s = parent.sibling(nDir);
186         synchronized (s) {
187           Node sL = s.left, sR = s.right;
188           // Result is null on failure or hold a double
189           // black node if the imbalance propagates up
190           Node[] result;
191           // First resolve the imbalance...
192           if (s.color == NodeColor.Red)
193           {
194             // Case 5-a 5-b
195             result = resolveRedSibling(g, parent, s, sL,
196                 sR, nDir);
197           }
198           else
199           {
200             // Case 4-a 4-b 6-a 6-b
201             result = resolveBlackNode(g, parent, s, nDir);
202           }
203           //Check the result
204           if (result == null)
205           {
206              // Rebalance unsuccessful. Retry
207             return false;
208           }
209           // The rebalance was successful. We can remove it
210           doubleBlackNode = result[0];
211           removeSonLessNode(parent, n);
212         } // lock on s
213       } // lock on n
214     } // lock on parent
215   } // lock on g
216
217   if (doubleBlackNode != null)
218   {
219     // The double black node is propagated upwards
220     correctBlackCount(doubleBlackNode);
221   }
222   return true;
223 }
```

Fig. 11. Removal of a black leaf. The actual unlink (Line 211) is only done after applying one rebalance (either Line 195 or Line 201) to transform the black leaf into a red node. This rebalance may cause a double black conflict that must be rebalanced (Line 220).

Proof. In Lemma 1 we have seen that we can always apply a transformation to a CRBT that is not a red–black tree. In Lemma 2 we have seen that any transformation modifies a tree T into a tree T' with $T' < T$. The last tree in the sequence is a tree S with $(o(S), od(S), r(S), rd(S)) = (0, 0, 0, 0)$, which is a red–black tree. □

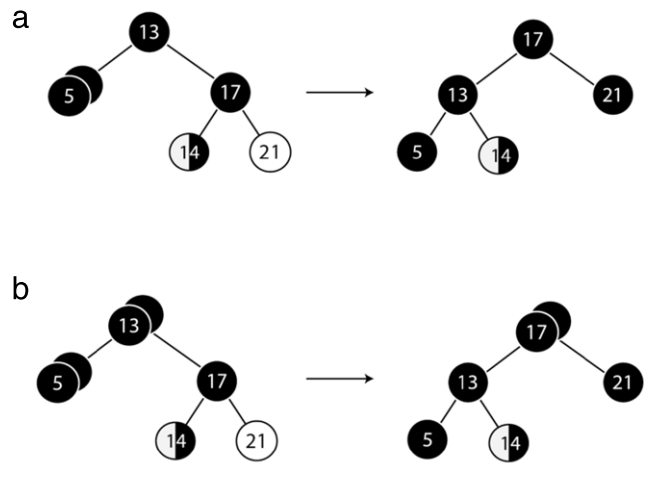

Fig. 12. (a) Case 6-1.a: Rebalance with a single rotation; (b) Case 6-1.b: Rebalance with a single and a double black parent.

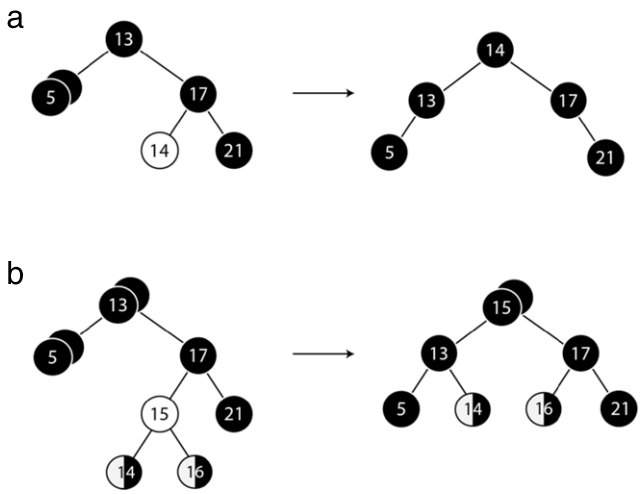

Fig. 13. (a) Case 6-2.a: Rebalance using a double rotation, (b) Case 6-2.b: Rebalance using a double rotation with double black parent. As in red–red conflict single rotations are cheaper then double rotations and are preferred.

6. Results

In this section we compare the results of our concurrent red–black tree to Bronson's concurrent AVL tree and to Lea's lock-free concurrent skip list. Both are written in Java and are the fastest concurrent dictionaries we know of. We use the same methodology of Bronson and Herlihy et al. [6] and Bronson et al. [2]. We chose to compare to these two concurrent dictionaries instead of chromatic trees or some other older concurrent red–black tree because they are newer and, in the case of the concurrent skip list, in wider use.

We report the average of 6 experiments. In each experiment we vary two parameters to evaluate their influence: the key range and the percentage of each of the three operations. With these two parameters we can influence how much contention there is for the tree. We will informally define contention as the concentration of structural changes, i.e. updates, on the tree during a run. For example a smaller tree will have higher contention than a larger tree because the updates are more physically concentrated. By changing the key range we influence the maximum size of the tree and therefor contention: a smaller range increases contention while a larger range reduces it. We use four key ranges from 2×10^3 to 2×10^6. The data is chosen randomly inside these ranges.

By using a different proportion of operations we can change how concentrated the structural changes are in time. A large proportion of updates imply that the tree is experiencing a large number of structural changes in close proximity. This allows us to approximate how the data structure reacts in three different scenarios: high contention with a large proportion of updates, low contention with a low number of updates and a medium sized load.

In both the AVL tree and our tree these scenarios reflect how the tree reacts to a different number of imbalances and rotations. As the number of updates grows the tree must handle more rebalances.

Fig. 17 shows the consolidated values from all of the runs. Each row represents a key range, from largest (top) to smallest (bottom), and each column is a different scenario. Contention increases from top to bottom and from right to left, so the lower left corner has the highest contention and the top right corner, the lowest.

Throughput increases in a given case as the number of threads increases, until we reach the number of hardware threads. Once we have more than 16 threads, throughput decreases due to contention for CPU time and interference between threads. When the tree is small it will not scale well. As the size of the tree increases the number of threads that can operate efficiently in it increases because of the larger workspace for each thread. In a small tree with many threads contention for locks and rotations will affect a large number of threads. In a large tree the same contention will be less common and the tree will scale well. An important factor is the access pattern: even in a large tree if all the threads are inserting in the same portion of the tree (e.g., a sequential set of numbers is inserted) then contention will be high because all of the updates will be concentrated in one portion of the tree. In the case of insertions this can produce red clusters, which reduces concurrency.

As the key range decreases more competition for nodes occurs between updating threads, but throughput also increases because the size of the tree is smaller: insertions, removals and searches traverse smaller paths reducing their turnaround time. In general, all of the operations improve their performance, but when there are many updates, rotations affect more threads. At a key range of 2×10^3, throughput is the largest and it decreases as the key range grows in all scenarios (see Fig. 17).

The red–black tree's throughput is very similar to the AVL tree and exhibits the same general tendencies. This shows the importance of the technique used to navigate the tree and its importance compared to how the tree is balanced. Both trees support contention efficiently and scale well but have problems when supporting multiprogramming workloads. In these workloads the skip-list performs well and its throughput is similar to both trees.

The effect of rotations on both trees can be seen as we compare their performance for one key range across different scenarios (Fig. 17). Updates are costlier than *get*'s because they affect the trees structure and cause rebalancing. As the number of *get*'s increase, the trees become more efficient and throughput improves. Our tree is more efficient than the AVL tree when there are many updates, but performs worse when the proportion of *get*'s increases. Using the average of all key ranges in 50–50–0 scenarios, our tree is on average 5% better than the AVL tree, but as fewer updates occur its superiority decreases to only 2% better at 20–10–70 until it performs 2% worse at 9–1–90. This correlates well with the number of rotations done by each tree. The red–black tree performs from 13.8% to 16.9% fewer rotations than the AVL tree (Fig. 18). As the number of updates increases the difference between the numbers of rotations increases in favor of the red–black tree. Because the red–black tree needs fewer rotations updates are less costly than in AVL trees. Unfortunately reducing the number of rotations lengthens the expected path length, which in turn results in costlier *gets* than in an AVL tree. We

```
224 /*
225  * Solves case 5-a and 5-b
226  * This operation first applies a rotation and then invokes a
227  * different operation that resolves the imbalance
228  * Return value is null if rebalance unsuccessful
229  * If rebalance is successful it depends on the second operation
230  * invoked (return value of resolveBlackNode)
231  */
232 Node[] resolveRedSibling(Node g,
233     Node parent, Node s, Node sL,
234     Node sR, final char nDir) {
235   Node newS = s.child(nDir); // newS is inner child of s
236   if (newS.color == NodeColor.Red) {
237     return null;
238   }
239   synchronized (newS) {
240     NodeColor pColor = parent.color;
241     NodeColor sLColor = color(sL);
242     NodeColor sRColor = color(sR);
243     if (nDir == Left) {
244       if (sLColor == NodeColor.Red)
245         return null;
246       else if (pColor == NodeColor.DoubleBlack) {
247         if (sRColor == NodeColor.Black)
248         {
249           // We won't be able to rebalance after the rotation
250           return null;
251         }
252         else {
253           // Case 5-b
254           parent.color = NodeColor.Black;
255           s.color = NodeColor.Black;
256           sR.color = NodeColor.Black;
257           rotateLeft_nl(g, parent, s, sL);
258         }
259       } else if (pColor == NodeColor.Red)
260       {
261         // First solve the red-red conflict
262         return null;
263       }
264       else { // p is black
265         // Case 5-a
266         s.color = NodeColor.Black;
267         parent.color = NodeColor.Red;
268         rotateLeft_nl(g, parent, s, sL);
269       }
270     } else {
271       // ... same for n as right child
272     }
273     // Now without releasing any locks apply the next operation
274     return resolveBlackNode(s, parent, newS, nDir);
275   } // lock on newS
276 }
```

Fig. 14. Handling of case 5. Because two rebalancing operations must be applied atomically we must first acquire all necessary locks. In line 239 the inner child of *s* is locked. It will participate at line 274 `resolveBlackNode` in the next rebalancing operation.

can see a clear tradeoff between the two approaches that correlates well with the sequential versions of each tree.

All three solutions scale similarly as the number of threads increases. This is expected as larger trees have less contention. In all scenarios throughput increased as the quantity of threads increases until reaching a threshold of 16 threads. The threshold is caused by the hardware limit. At 16 threads all of the hardware threads are being used so the computer was at maximum capacity. When there are more than 16 threads there is competition for the computing resources and an expected decrease in throughput per thread. The key range also affects scalability; a larger range shows a clear increase in efficiency per thread, although not in total throughput due to the increase in traversal times (because the tree is larger) and the threads impinge less on each other.

With respect to scaling, we performed the following simple experiment to measure speedup and isoefficiency [8] empirically: we executed from 2560 all the way up to 4,096,000 insertions in an initially empty tree, and computed speedup and isoefficiency metrics for 1, 2, 4, 8 and 16 threads. The results are shown in Figs. 19 and 20. In particular, Fig. 20 shows that our approach exhibits isoefficiency. For example, for an efficiency of 0.5: for 2 processors, this is reached for $N = 32{,}000$ insertions; for 4 processors, N must be 64,000; for 8 processors, N must be 384,000; and for 16 processor N must be larger than 4,000,000.

7. Conclusion

We have developed a new concurrent red–black tree that supports high contention and is efficient in several scenarios. Its

```
277 /*
278  * Solves cases 4-a 4-b 6-a 6-b
279  * Returns null if rebalance is unsuccessful
280  * Return new double black node if it is propagated
281  */
282 Node[] resolveBlackNode(Node g, Node parent, Node s, char nDir) {
283   Node doubleBlackNode;
284   Node sL = s.left, sR = s.right;
285   NodeColor pColor = parent.color;
286   NodeColor sLColor = color(sL), sRColor = color(sR);
287
288   if (s.color == NodeColor.DoubleBlack) {
289     // Case 4-b
290     // This case can't happen if we come from resolveRedSibling
291     if (pColor == NodeColor.DoubleBlack)
292       return null;
293     s.color = NodeColor.Black;
294     if (pColor == NodeColor.Red) {
295       parent.color = NodeColor.Black;
296       doubleBlackNode = null;
297     } else {
298       parent.color = NodeColor.DoubleBlack;
299       doubleBlackNode = parent;
300     }
301   } else if (sLColor != NodeColor.Red && sRColor != NodeColor.Red) {
302     // Case 4-a
303     if (pColor != NodeColor.DoubleBlack) {
304       s.color = NodeColor.Red;
305       if (pColor == NodeColor.Black) {
306         doubleBlackNode = parent;
307         parent.color = NodeColor.DoubleBlack;
308       } else {
309         parent.color = NodeColor.Black;
310         doubleBlackNode = null;
311       }
312     } else {
313       // Can't occur if we come from resolveRedSibling
314       return null;
315     }
316   } else // Rotation cases, similar to rebalance_to_Left
317   {
318     // Case 6-a 6-b
319     doubleBlackNode = rebalanceSibling(g, parent, s, sL, sR, nDir);
320   }
321   Node[] result = {doubleBlackNode };
322   return result;
323 }
```

Fig. 15. Rebalance a double black node. Line 319 `rebalanceSibling` corresponds to case 6 and is not shown here.

throughput can be as good as 5% better than Bronson's AVL tree and 72% better than the best-known concurrent skip list solution (although it can also be 2% worse than Bronson's AVL tree and 6% worse than concurrent skip lists). These results are the best known for a concurrent red–black tree and represent a significant advance towards solving the concurrent dictionary problem.

By leveraging the advantages of red–black trees compared to AVL trees, our solution reduces the number of rotations necessary to maintain the balance of the tree. In particular our tree performs at least 13% less rotations than AVL trees. This helps our tree in high contention scenarios where it performs similarly to concurrent skip lists and better than AVL trees. In low contention scenarios it performs similarly to AVL trees and significantly better than concurrent skip lists.

Our tree has several new rebalancing operations that handle concurrent situations that are not necessary in sequential red–black trees. We added the least number of new rebalancing operations that still allows us to solve all cases efficiently. Most of the new operations extend sequential solutions that had not considered the possibility of simultaneous imbalances in the vicinity of a node. All the operations used only affect a small neighborhood,

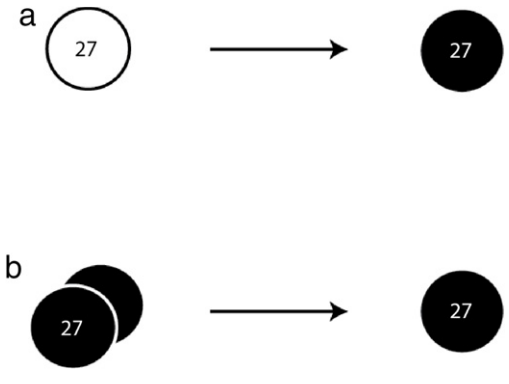

Fig. 16. (a) Red root. (b) Extra black root.

which improves the concurrency of several updates. Because of this and by using optimistic concurrency to traverse the tree, we have built a very simple but efficient concurrent red–black tree. Due to scheduling issues it is not possible to guarantee $O(\log N)$.

Red–black trees play a significant part in sequential dictionaries. With our results we believe that they can also become an

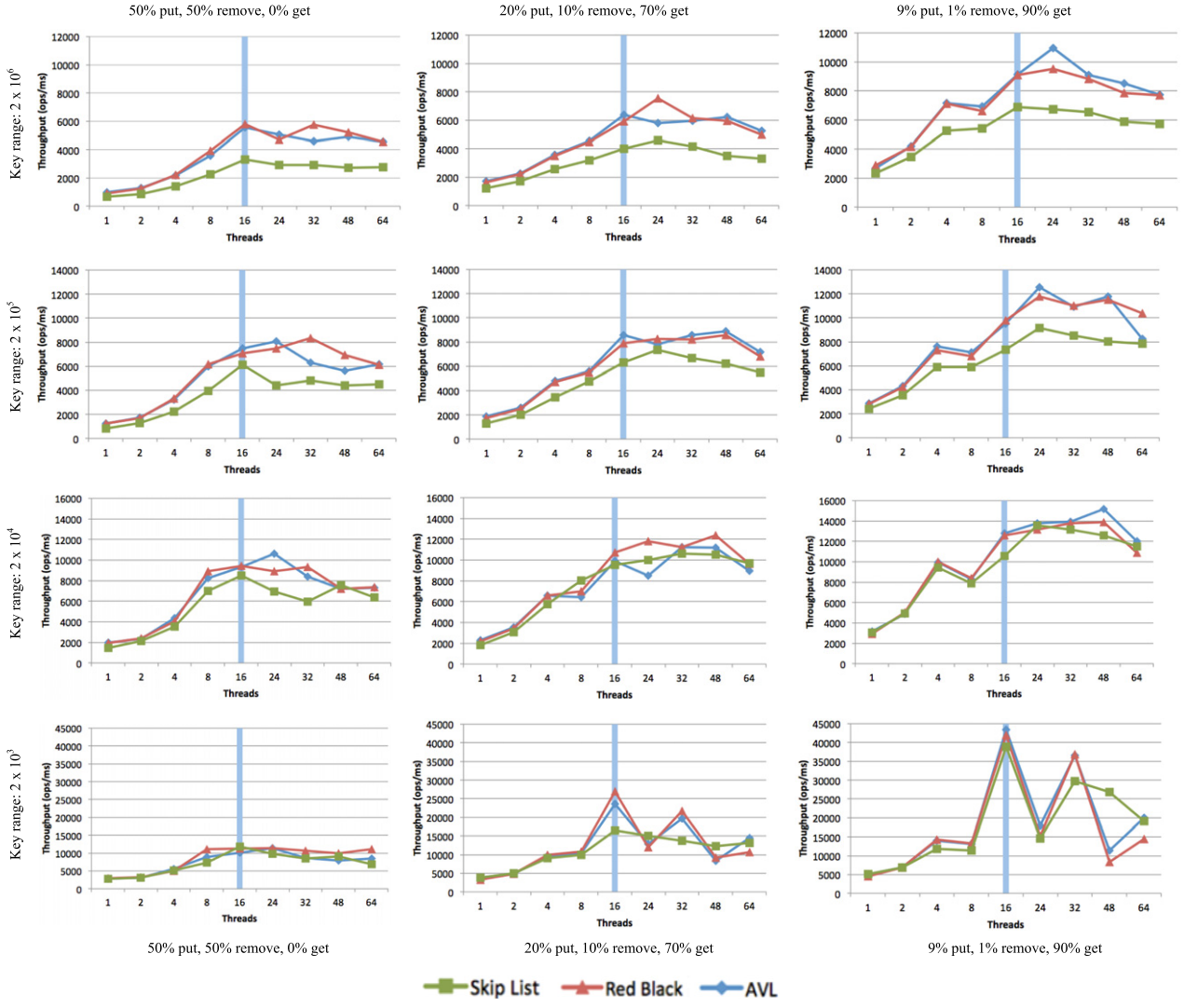

Fig. 17. Results. Contention increases from bottom to top and from left to right. As contention increases the red–black tree outperforms both other solutions. In the bottom left corner it has a 14% higher average throughput than the AVL tree.

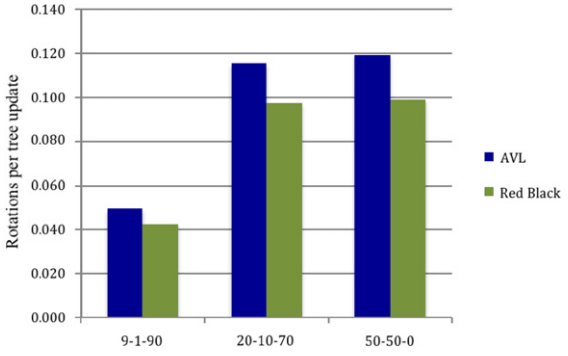

Fig. 18. The red–black tree consistently executes fewer rotations than the AVL tree. In each scenario the values did not vary across number of threads or key range.

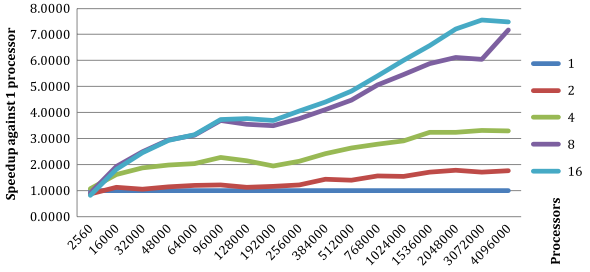

Fig. 19. Speedup of the red–black tree. As the size of the tree increases contention decreases and speedup improves. Constructing a tree of smaller size may benefit from fewer threads.

important solution to concurrent dictionaries. Their low number of rotations is an important benefit compared to other tree-based solutions and they are more efficient than concurrent skip-lists in our tests. Although more complex than AVL trees, they also present more potential for optimizations.

It remains to be studied how to apply red–black trees effectively as concurrent dictionaries and to leverage their advantages for specific programming situations. There are also many theoretical proofs that are still open such as the theoretical isoefficiency and other scaling measures.

Our research opens the question of what the current bottleneck of concurrent binary search trees is. The largest inefficiencies – rotations – have been reduced through new techniques for traversing the tree and our results seem to indicate that reducing their quantity may not improve the throughput of the tree markedly. Rotations are potential bottlenecks but they affect very local points in the tree. Thus, they only delay the threads that are concurrently traversing the region that is involved in a rotation.

	Isoefficiencies Values				
N/Processors	1	2	4	8	16
2,560	1.000	0.434	0.266	0.114	0.051
16,000	1.000	0.565	0.407	0.241	0.114
32,000	1.000	0.525	0.469	0.311	0.153
48,000	1.000	0.572	0.494	0.368	0.184
64,000	1.000	0.603	0.508	0.391	0.197
96,000	1.000	0.604	0.571	0.463	0.234
128,000	1.000	0.566	0.538	0.444	0.235
192,000	1.000	0.581	0.485	0.437	0.231
256,000	1.000	0.609	0.534	0.471	0.254
384,000	1.000	0.718	0.605	0.515	0.275
512,000	1.000	0.701	0.661	0.559	0.301
768,000	1.000	0.784	0.697	0.633	0.338
1,024,000	1.000	0.773	0.729	0.682	0.375
1,536,000	1.000	0.856	0.809	0.734	0.411
2,048,000	1.000	0.888	0.810	0.765	0.451
3,072,000	1.000	0.856	0.830	0.756	0.472
4,096,000	1.000	0.884	0.822	0.898	0.468

Fig. 20. Isoefficiency values of creating a tree of size n.

References

[1] J. Boyar, K.S. Larsen, Efficient rebalancing of chromatic search trees, Journal of Computer and System Sciences 49 (3) (1994) 667–682.

[2] N.G. Bronson, J. Casper, H. Chafi, K. Olukotun, A practical concurrent binary search tree, SIGPLAN Notices 45 (5) (2010) 257–268.

[3] L.J. Guibas, R. Sedgewick, A dichromatic framework for balanced trees, in: Proceedings of the 19th IEEE Symp. Foundations of Computer Science, Ann Arbor, MI, USA, 1978, pp. 8–21.

[4] S. Hanke, The performance of concurrent red–black tree algorithms, in: Algorithm Engineering, Springer, Berlin, Heidelberg, 1999, pp. 286–300.

[5] S. Hanke, T. Ottmann, E. Soisalin-Soininen, Relaxed balanced red–black trees, Lecture Notes in Computer Science 1203 (1997) 193–204.

[6] M. Herlihy, Y. Lev, V. Luchangco, N. Shavit, A provably correct scalable concurrent skip list, in: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, PODC'03, ACM, New York, NY, USA, 2003, pp. 92–101.

[7] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann Publishers, 2008.

[8] V. Kumar, G. Anshul, Analysis of scalability of parallel algorithms and architectures: a survey, in: Proceedings of the 5th International Conference on Supercomputing, ICS'91, ACM, New York, 1991, pp. 396–405.

[9] K.S. Larsen, Amortized constant relaxed rebalancing using standard rotations, Acta Informatica 35 (10) (1998) 859–874.

[10] O. Nurmi, E. Soisalon-Soininen, Chromatic binary search trees, Acta Informatica 33 (5) (1996) 547–557.

[11] R. Sedgewick, Left-leaning red–black tree. 2008, September. From www.cs.princeton.edu/rs/talks/LLRB/LLRB.pdf.

[12] N. Shavit, Data structures in the multicore age, Communications of the ACM (2011) 76–84.

Juan Besa is a master's student at the Department of Computer Science Pontificia Universidad Católica de Chile, and at same time he works for Synopsys developing EDA software design tools. His master's thesis focuses on concurrent data structures.

Yadran Eterovic obtained his degree in Electrical Engineering at Pontificia Universidad Católica de Chile (PUC-Chile) and his Ph.D. in Computer Science at UCLA. He is now Associate Professor and Head of the Department of Computer Science at PUC-Chile, where he teaches programming, data structures, and concurrent programming. His research focuses on aspect oriented software development and concurrent programming,