PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA

A CONCURRENT RED BLACK TREE

JUAN JOSE BESA VIAL

Tesis para optar al grado de
Magister en Ciencias de la Ingeniería

Profesor Supervisor:
YADRAN ETEROVIC

Santiago de Chile, mayo 2012

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA

A CONCURRENT RED BLACK TREE

JUAN JOSE BESA VIAL

Tesis presentada a la Comisión integrada por los profesores:

YADRAN ETEROVIC

JORGE BAIER

FEDERICO MEZA

MARCELO GUARINI

Para completar las exigencias del grado de
Magister en Ciencias de la Ingeniería
Santiago de Chile, mayo 2012

A mis Padres y Tere, que me apoyaron siempre.

AGRADECIMIENTOS

Agradezco a Yadran Eterovic, por apoyarme y esperarme, discutiendo la investigación hasta que estuvo lista.

INDICE GENERAL

Pág.

INDICE DE FIGURAS

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 complex 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.

KEY WORDS

Concurrent Data Structure, Red Black Tree, Algorithm design.

RESUMEN

La necesidad de tener estructuras de datos capaces de soportar varios procesos ha crecido con la masificación de los computadores multiprocesadores. Las estructuras de datos concurrentes buscan proveer una eficiencia similar a las estructuras de datos secuenciales permitiendo además el acceso concurrente a varios procesos y proveyendo mecanismos de sincronización transparentemente a esos procesos.

Los árboles rojo negros son una importante estructura de datos utilizada en muchos sistemas. Lamentablemente ha sido complejo implementar un árbol rojo negro concurrente eficiente para computadores de memoria compartida. Debido a esto la mayoría de los esfuerzos de investigación han sido dirigidos hacia otras estructuras de datos tipo diccionarios, principalmente *skiplists* y arboles AVL.

En esta tesis presentamos un nuevo tipo de árbol rojo negro concurrente que utiliza técnicas de *optimistic concurrency* and nuevas operaciones de balance para escalar eficientemente y soportar contención.

Nuestro árbol se comporta favorablemente comparado con otros diccionarios similares. En particular, en escenarios de alta contención se comporta hasta 14% mejor que la solución más conocida de los diccionarios concurrentes.

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 [Shavit, 2011]. Many programmers are beginning to confront the challenges 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 an efficient data structure 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 in a wide variety of application shows how effective a tool they are for 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. 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.

Red black trees are extensively used as dictionaries and are the default symbol table for Java, C++, Python, BSD Linux and other systems [Sedgewick, 2008]. 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(logn) expected performance—are the prevalent concurrent dictionaries [Herlihy and Shavit, 2008]. By correcting the inefficiencies of concurrent red black trees, we think they can also become practical concurrent dictionaries.

In concurrent environments the most popular dictionary are concurrent skip lists, a type of augmented linked lists that have expected logarithmic complexity for searches, insertions, and removals. Concurrent skip lists are preferred over balanced binary search tree 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 affect all other threads operating in the vicinity of the change. Periodic rebalances thus cause bottlenecks and contention. Recent research of new techniques for traversing binary search trees have developed a concurrent AVL tree [Bronson et al. 2010] 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 apply fewer rotations than AVL trees. In particular red black trees apply O(1) rotations to correct an imbalance in the tree while AVL trees apply O(logn), where n is the number of nodes in the tree. Due to this 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. 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 still maintains 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 presents our algorithm and details the new rebalancing operations introduced. Section 5 contains a formal proof of the correctness of our algorithms. Section 6 presents experimental results: we compare the performance of our tree to those of a concurrent AVL Tree and the concurrent skip list from Java's concurrent library. Finally, section 7 concludes with a discussion of the findings and presents possible future research.

1.1 A note on the code

We present important parts of our algorithm in Java. We expect basic knowledge of Java and especially of basic concurrency idioms. For a complete treatment of concurrency we recommend [Lea, 2000].

2 DEFINITIONS

A red black tree is a self-balancing binary search tree [Guibas and Sedgewick, 1978]. 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. A black unit is the level of blackness of a node. Thus each black node contributes one black unit, while red nodes do not contribute any black unit. A red node is transformed into a black node by adding a black unit to it and similarly 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 Figure 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.*

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

```
1 protected enum NodeColor { Red, Black, DoubleBlack };
2
3 protected static NodeColor color (final Node<?, ?> n)
4 {
5      return n == null ? NodeColor.Black : n.color;
6 }
```

Figure 1: There are three possible colors: Red, Black and Double black. Double blacks are imbalances due to removing a node. Null nodes are considered to be black nodes

The above rules guarantee $O(\log n)$ time complexity for *get* operations, *insertion* operations (also called *put* operations) and *removal* operations. Insertions and removals are *update* operations, and the thread performing the update is an **updater**. An **imbalance** is any local state (a node, or a parent node and its child) 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.

3 RELATED WORK

The most well known concurrent red black tree is the chromatic tree proposed by Nurmi and Soisalon-Soininen [1996] and also presented at PODS'91. In this 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 are 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 chromatic trees are external trees, 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 Boyer and Larsen [1994] 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 technique. Nurmi and Soisalon-Soininen use three kinds of locks: r-locks, w-locks—which function in a 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.

Hanke et al. [1997] and Larsen [1998] also presented a relaxed version of balanced red black trees. These are similar to chromatic trees but differ in the rebalancing operations allowed on the tree. Hanke et al. updating threads don't remove nodes, but only mark them to be removed; the rebalancing thread does the actual removal. This technique was also used by Bronson et al. concurrent AVL tree [2010] (see below) and in our tree, although extended to internal trees.

Hanke et al. 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. [2010] 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 (-∞, +∞) 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 doesn't increase the tree size significantly.

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 use 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 concurrent red black tree that we know of. It also is the first efficient concurrent red black tree to not use relaxed balance but instead use the updaters to correct any imbalance introduced in the tree. We extend the standard red black operations to consider new situations that occur in concurrent environments and take advantage of them.

4 A NEW ALGORITHM TO REBALANCE CONCURRENT RED BLACK TREES

We now present our algorithm for constructing a concurrent red black tree. Our focus is on maintaining the rebalance of the tree, so our focus is on updates and rebalancing the tree. We begin by showing those related to insertions and then removals. Each operation has a figure associated to it while the text highlights relevant concurrency issues.

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, then 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 was 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 Figure 2. First, the node with the key is transformed into a route node by setting its value to *null (*line 17.) This effectively removes *k* from the tree: the route node still holds the key but its value is marked as *null*, so the node only contains routing information. Second, the updater attempts to remove, or *unlink,* the node from the tree (line 20).

To remove a node with two children we must first swap it with it successor. Unfortunately the successor may be many links away so it is very costly to do in a

```
 7 private Object attemptUpdateNode(final Object newValue,
 8     final Node<K, V> n, final Node<K, V> parent) {
 9     if (newValue == null) { // removal
10     if (n.value == null)
11       return null;
12     final Object oldValue;
13     synchronized (n) {
14       if (isUnlinked(n.changeOVL))
15         return SpecialRetry;
16       oldValue = n.value;
17       n.value = null; // Transform into route node
18     }
19     if (n.left == null || n.right == null)
20       removeNode(n);
21     return oldValue;
22   } else {  // update
23     synchronized (n) {
24       if (isUnlinked(n.changeOVL))
25         return SpecialRetry;
26       final Object oldValue = n.value;
27       n.value = newValue;
28       return oldValue;
29     }
30 } }
```

Figure 2. Update and remove key. Removal of the actual node is left to a separate method, `removeNode` (line 20). By setting the nodes value to null, the node is transformed into a route node (line 17).

concurrent tree so we do not support this removal. So 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. 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 discuss 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; if necessary acquiring more locks.
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 and repeat.

Acquiring locks only affect the 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.

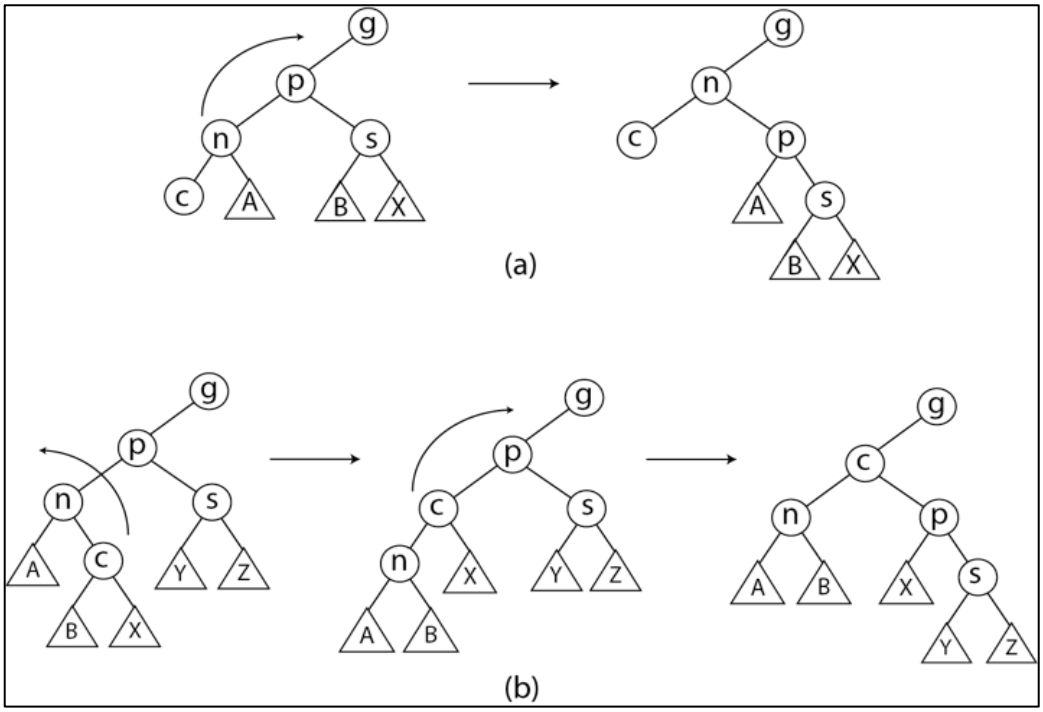

Figure 3. Rotation in a red black tree: (a) Single rotation, (b) Double rotation. Nodes *g*, *p*, and *n* must always be locked. For a double rotation *c* must also be locked.

In the next sections we use the following labels (as seen in Figure 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 Figure 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, must be locked. In our design we prefer executing a single rotation to a double rotation (see code line 90 in Figure 7).

For all operations and rotations we only present those where *n* is the left child of *p*, all operations —except for the root operations— have a symmetrical version for when *n* is the right child of *p*.

4.2 Red – red imbalances

To insert a new node we attach a red node to a leaf of the tree. 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, i.e. *n*, is the red parent.

We have 6 different cases to rebalance a conflicted tree. In cases 1, 2 and 3 we show the rebalancing operations to correct red-red imbalances. We also include new operations for cases were 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, 5 and 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 this and the following section, in each figure node *n* is the node with key 5; black nodes are black with white letter, red nodes are white with black letters, double black nodes are represented by two overlapping black nodes and, if the nodes color is not important its background and letters are black and white.

```
31 private Node<K,V> pushRed(final Node<K,V> g,
32     final Node<K,V> p, final Node<K,V> n)
33 {
34   p.left.color = p.right.color = NodeColor.Black;
35   if (p.color == NodeColor.DoubleBlack) {
36     p.color = NodeColor.Black;
37     return n;
38   } else { // p is black
39     p.color = NodeColor.Red;
40     return g != rootHolder ? g : p;
41 } }
42
43 private Node<K, V> rotateOrPushRed(final Node<K, V> n) {
44   final Node<K, V> p = n.parent;
45   final Node<K, V> g = p.parent;
46   if (g == null) {
47     return blackenRoot(n);
48   } else if (color(p) == NodeColor.Red) {
49     fixColorAndRebalance(p);    // For red clusters
50     return n;
51   } else {
52     synchronized (g) {
53       if (isUnlinked(g.changeOVL) || p.parent != g)
54         return n;
55       synchronized (p) {
56         if (isUnlinked(p.changeOVL) || n.parent != p)
57           return n;
58         if (n.color != NodeColor.Red
59             || p.color == NodeColor.Red)
```

```
60            return n;
61          if (color(p.left) == NodeColor.Red
62              && color(p.right) == NodeColor.Red) {
63            return pushRed(g, p, n);
64          } else if (p.left == n)
65            return rebalance_to_Right(g, p, n);
66          else
67            return rebalance_to_Left(g, p, n);
68 } } } }
```

Figure 4. The methods `rebalance_to_left` and `rebalance_to_right` (line 65 and 67) 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 59 it does not try to rebalance *p* and simply backtracks. In line 46 if *g* is null then *p* is the root holder and *n* is the root and must be painted black.

4.2.1 Case 1: If *s* is also red

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

If the grandparent *g* of *n* is also red, then by pushing red we generate a new red-red imbalance higher up in the tree. The updater then resolves the new imbalance similarly (Line 40 in Figure 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 their update concluded. 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.

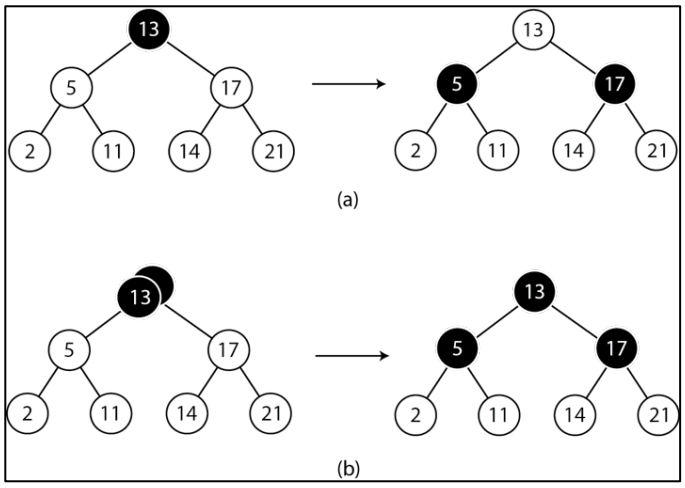

Figure 5. (a) Case 1-a: Push red, *p* must be red (b) Case 1-b Push red and *p* is a double black node. In both cases only one of the four bottom nodes must be red to do the operation

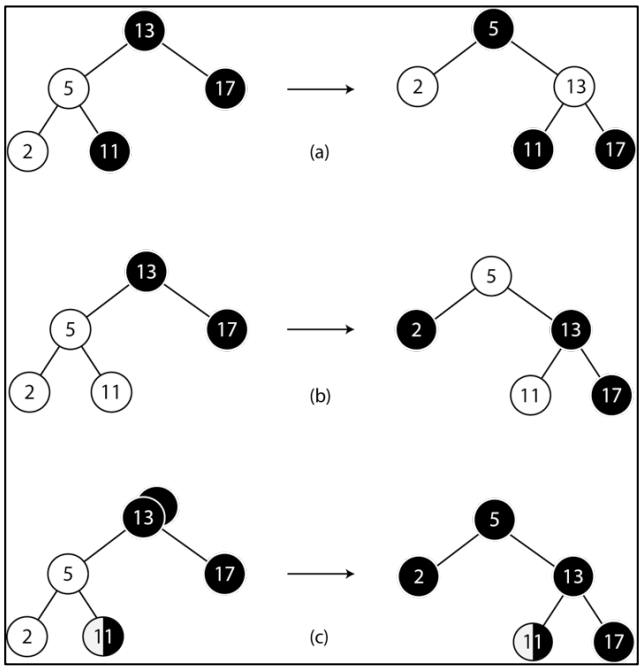

Figure 6. In all cases the node with key 17 can be both black or double black because it doesn't interfere with the rotation. (a) Case 2-1: Single rotation (b) Case 2-b: Single rotation with double red, after applying Case 2-a push red. (c) Case 2-c Single rotation with *p* double black.

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

In Figure 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 case 2-1 would resolve the imbalance at *n* but generates 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 doesn't execute the operation has concluded its update.

Operation c) is an example of when a red-red imbalance collides with a double black imbalance. In this case both imbalances can be solved with one rotation.

```
69 private Node<K, V> rebalance_to_Left(Node<K, V> g,
70     Node<K, V> p,
71     Node<K, V> n) {
72   synchronized (n) {
73     final Node<K, V> nL = n.left, nR = n.right;
74     final NodeColor nLcolor = color(nL), nRcolor = color(nR);
75     final NodeColor pColor = p.color;
76     if (nLcolor != NodeColor.Red
77       && nRcolor != NodeColor.Red) {
78       return n;
79     } else if (nLcolor == NodeColor.Red
80         && nRcolor == NodeColor.Red) {
81       rotateLeft_nl(g, p, n, nL);
```

```
82        if (pColor == NodeColor.DoubleBlack) {
83          n.color = nR.color = p.color = NodeColor.Black;
84          return p;
85        } else {
86          nR.color = NodeColor.Black;
87          return g != rootHolder ? g: n;
88        }
89      }
90      else if (nRcolor == NodeColor.Red) { //Prefer a single...
91        rotateLeft_nl(g, p, n, nL);
92        if (pColor == NodeColor.DoubleBlack) {
93          n.color = nR.color = p.color = NodeColor.Black;
94        } else {
95          n.color = NodeColor.Black;
96          p.color = NodeColor.Red;
97        }
98        return p;
99      } else {    // ... to a double rotation
100        synchronized (nL) {
101          rotateLeftOverRight_nl(g, p, n, nL);
102          if (p.color == NodeColor.DoubleBlack) {
103            nL.color = n.color = p.color = NodeColor.Black;
104            return p;
105          } else if (nLcolor == NodeColor.Red
106              || nRcolor == NodeColor.Red) {
107            n.color = NodeColor.Black;
108            return g != rootHolder ? g : nL;
109          } else {
110            nL.color = NodeColor.Black;
111            p.color = NodeColor.Red;
```

```
112           return p;
113     } } }
114 } }
```

Figure 7. Red-red conflict resolution for when *n is the left child of p.* In line 90 we prefer doing a single rotation to a double rotation.

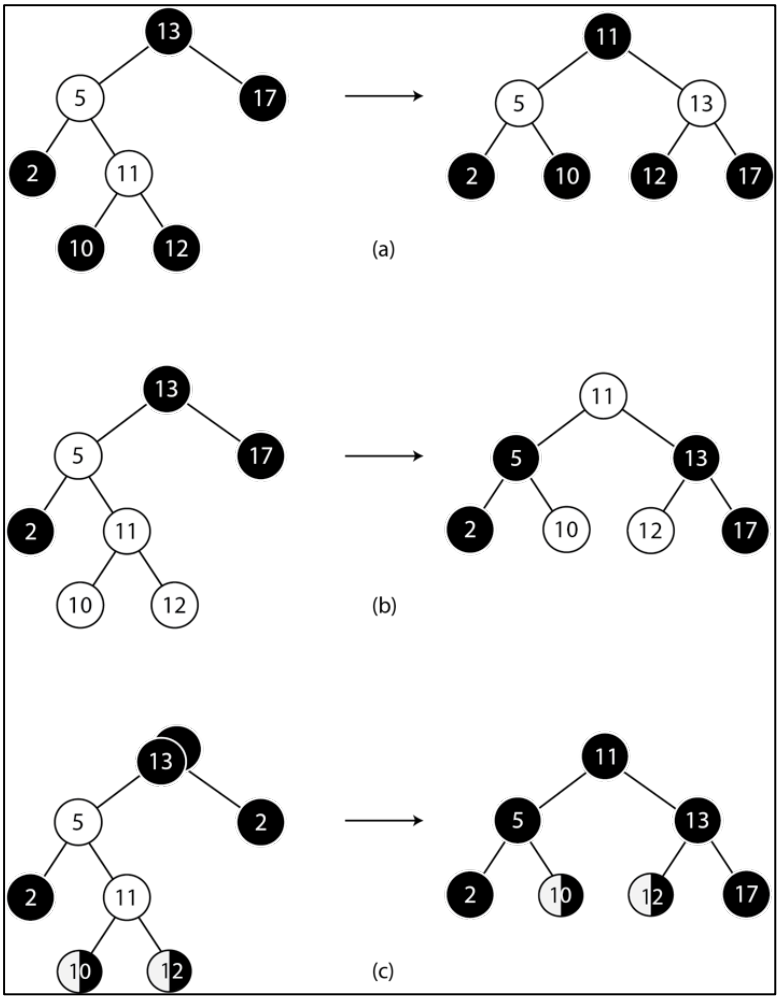

Figure 8. (a) Case3-a: Double rotation, (b) Case 3-b: Double rotation with double red, (c) Case 3-c: Double rotation extra black.

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.

4.2.3 Case 3: *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. 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.

4.2.4 Red clusters

In a red-red imbalance if *p* is also red then *n* can't 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 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 can't 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 rebalancing *n*. This prevents updaters from retrying to balance *n*, a node that can't be balanced 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 Double black imbalances

The following section presents the necessary operations to correct double black imbalances. These 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 removing 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 removes the node. To other updaters this is an atomic action, so at no moment rule 3 is broken.

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.

The following figures and section will 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 very difficult to rebalance. It is preferable not to allow the removal of double black nodes or adding a black unit to a double black node. Fortunately triple black nodes (or higher) are very uncommon and require a very specific pattern of removals. Their frequency of appearance is very small so they are better handled by not allowing them to exist.

```
115   private boolean attempRemoveBlackLeaf(final Node<K, V> n,
116       final Node<K, V> parent, final Node<K, V> g) {
117     final Node<K, V> damaged;
118     Node<K, V> doubleBlackNode = null;
119     synchronized (g) {
```

```
120       if (parent.parent != g || isUnlinked(g.changeOVL))
121         return false;
122       synchronized (parent) {
123         if (n.parent != parent || isUnlinked(parent.changeOVL))
124           return false;
125      // Link g -> p -> n is locked. Also Link p -> s is also
locked.
126      // Colors of g,p,n,s are also locked.
127         synchronized (n) {
128           final NodeColor nColor = n.color;
129           final Node<K, V> nL = n.left;
130           final Node<K, V> nR = n.right;
131
132           // ... Handle other cases
133
134           final char nDir = parent.left == n ? Left : Right;
135           final Node<K, V> s = parent.sibling(nDir);
136           synchronized (s) {
137             final Node<K, V> sL = s.left;
138             final Node<K, V> sR = s.right;
139             final Node<K, V>[] result;
140             if (s.color == NodeColor.Red)
141               result = resolveRedSibling(g, parent, s, sL,
142                   sR, nDir);
143             else
144               result = resolveBlackNode(g, parent, s, nDir);
145             if (result == null)
146               return false;
147             damaged = result[0];
148             doubleBlackNode = result[1];
```

```
149             removeSonLessNode(parent, n);
150             } // Synchronization of s
151           } // Synchronization n
152         } // Synchronization parent
153       } // Synchronization g
154       if (doubleBlackNode != null)
155         correctBlackCount(doubleBlackNode);
156       if (damaged != null)
157         fixColorAndRebalance(damaged);
158       return true;
159     }
```

Figure 11. Removal of a black leaf. The actual unlink (Line 149) is only done after applying one rebalance (either Line 141 or Line 144) to transform the black leaf into a red node. This rebalance may cause a double black conflict that must be rebalanced (Line 155). Line 157 checks if removing the node has made possible unlinking a route node.

4.3.1 Case 4: Double black and there is no red

This operation is very similar to push red but instead of pushing red the updater pushes black, i.e. it adds a black unit to the father node and removes a black unit from each of the two children nodes. 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.

4.3.2 Case 5: Double black and *s* is red

When *s* is a red node there is no operation to immediately fix the imbalance. 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*.

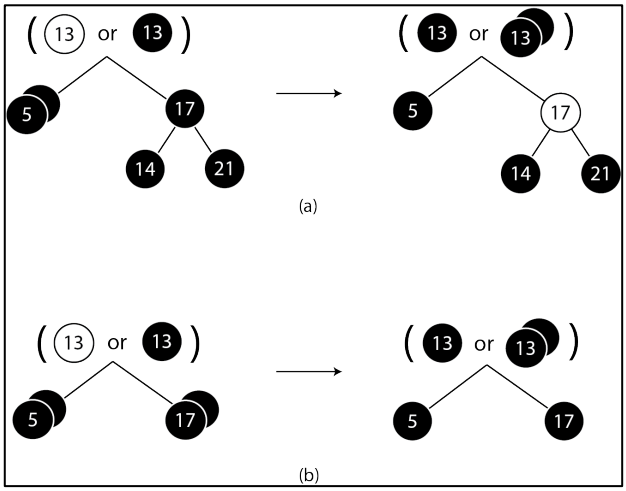

Figure 7. (a) Case 4-a: Push black, (b) Case 4-b: Push black with a double black sibling. Because node 17 must have two black children it must also be locked to prevent possible red-red conflicts.

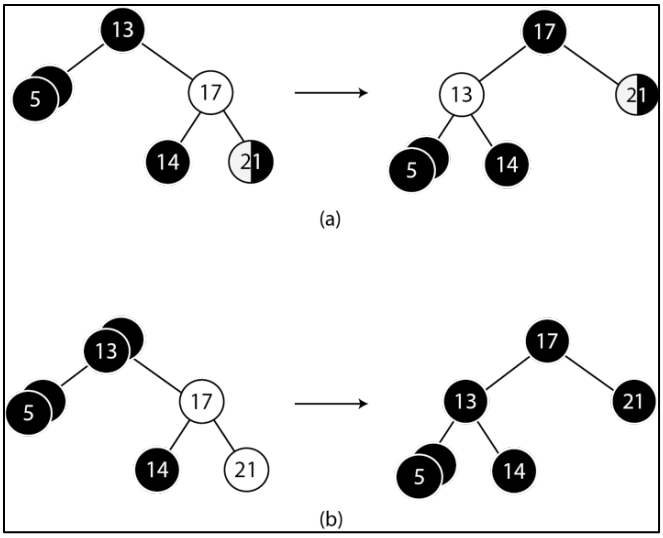

Figure 10. (a) Case 5-a: Red sibling, (b) Case 5-b: Red sibling with a red right nephew and a double black parent.

Both are immediately followed by an operation that corrects the double black node at 5.

The inner child of *s* must be locked before rotating so it can participate in the next operation. Both operations are "concatenated", which corrects the imbalance.

If the inner child of *s* is also red, then *n*'s updater can't 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 removal instead of waiting for a more favorable state.

```
160 private Node<K, V>[] resolveRedSibling(final Node<K, V> g,
161   final Node<K, V> parent, final Node<K, V> s, final Node<K, V>
sL,
162   final Node<K, V> sR, final char nDir) {
163   final Node<K, V> newS = s.child(nDir); // newS is inner child of
s
164   if (newS.color == NodeColor.Red) {
165     return null;
166   }
167   synchronized (newS) {
168     final NodeColor pColor = parent.color;
169     final NodeColor sLColor = color(sL);
170     final NodeColor sRColor = color(sR);
171     if (nDir == Left) {
172       if (sLColor == NodeColor.Red)
173         return null;
174       else if (pColor == NodeColor.DoubleBlack) {
175         if (sRColor == NodeColor.Black)
176           return null;
177         else {
178           parent.color = NodeColor.Black;
179           s.color = NodeColor.Black;
180           sR.color = NodeColor.Black;
```

```
181           rotateLeft_nl(g, parent, s, sL);
182         }
183       } else if (pColor == NodeColor.Red)
184         return null;
185       else { // p is black
186         s.color = NodeColor.Black;
187         parent.color = NodeColor.Red;
188         rotateLeft_nl(g, parent, s, sL);
189       }
190     } else {
191       // ... same for n as right child
192     }
193     // Now without releasing any locks apply the next operation
194     return resolveBlackNode(s, parent, newS, nDir);
195   } // Synchronization newS
196 }
```

Figure 14. Handling of case 5. Because two rebalancing operations must be applied atomically we must first acquire all necessary locks. In line 167 the inner child of *s* is locked. It will participate at line 194 `resolveBlackNode` in the next rebalancing operation.

4.3.3 Case 6: 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*

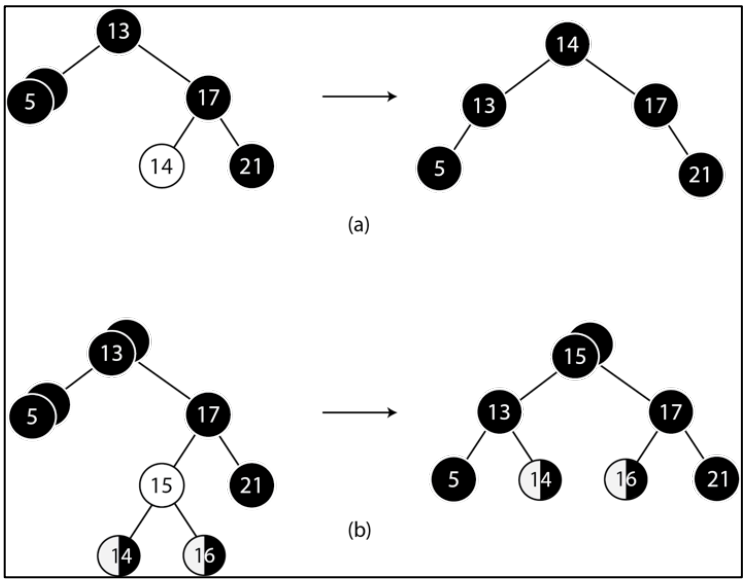

Figure 12. (a) Case 6-1.a: Rebalance with a single rotation; (b) Case 6-1.b: Rebalance with a single and a double black parent.

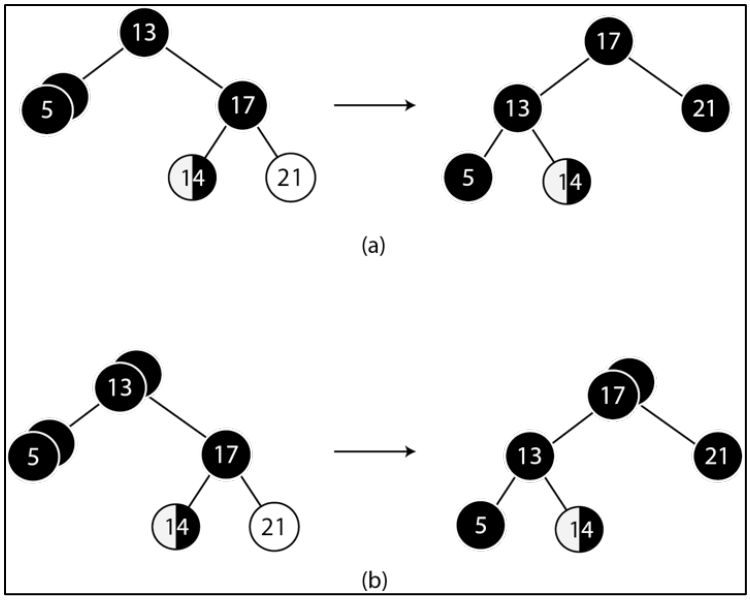

Figure 13. . (a) Case 6-2.a: Rebalance using a double rotation, (b) Case 6-2.b: Rebalance using a double rotation with double black parent. As in red-red conflict single rotations are cheaper then double rotations and are preferred.

```
197 private Node<K, V>[] resolveBlackNode(final Node<K, V> g,
198   final Node<K, V> parent, final Node<K, V> s, final char nDir) {
199   final Node<K, V> doubleBlackNode;
200   final Node<K, V> damaged;
201   final Node<K, V> sL = s.left;
202   final Node<K, V> sR = s.right;
203   final NodeColor pColor = parent.color;
204   final NodeColor sLColor = color(sL);
205   final NodeColor sRColor = color(sR);
206   if (s.color == NodeColor.DoubleBlack) {
207     if (pColor == NodeColor.DoubleBlack)
208       return null;
209     s.color = NodeColor.Black;
210     if (pColor == NodeColor.Red) {
211       parent.color = NodeColor.Black;
212       damaged = parent;
213       doubleBlackNode = null;
214     } else {
215       parent.color = NodeColor.DoubleBlack;
216       damaged = s;
217       doubleBlackNode = parent;
218     }
219   } else if (sLColor != NodeColor.Red && sRColor != NodeColor.Red)
220   {
221     if (pColor != NodeColor.DoubleBlack) {
222       s.color = NodeColor.Red;
223       if (pColor == NodeColor.Black) {
224         doubleBlackNode = parent;
225         parent.color = NodeColor.DoubleBlack;
226       } else
```

```
227       {
228         parent.color = NodeColor.Black;
229         doubleBlackNode = null;
230       }
231
232       if (s.value == null && sL == null && sR == null) {
233         // We can remove s freely
234         removeSonLessNode(parent, s);
235         damaged = parent;
236       } else {
237         damaged = s;
238       }
239     } else {
240       return null;
241     }
242   } else // Rotation cases, similar to rebalance_to_Left
243   {
244     doubleBlackNode = rebalanceSibling(g, parent, s, sL, sR,
nDir);
245     damaged = s;
246   }
247   Node[] result = { damaged, doubleBlackNode };
248   return result;
249 }
```

Figure15. Rebalance a double black node. Line 244 `rebalanceSibling` corresponds to case 6 and is not shown here. In line 232 we take advantage of the locks we hold and remove *s* if it is a route node with no children.

4.3.4 Rebalancing operations involving red-red imbalances and double black imbalances

When red-red imbalances collide with double black imbalances it is beneficial to the rebalancing operation 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 and distribute the extra black unit to both branches.

This is very different 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 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 who 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: if the root is red and if 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.

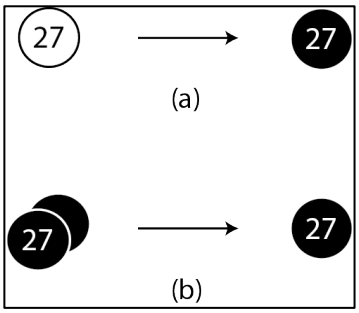

Figure 16. (a) Red root, (b) Extra black root

5 CORRECTNESS

In this section we prove that our algorithm finishes and correctly rebalances a red black tree. Our proof is an adapted from the proof given by Nurmi and Soisalon-Soininen [1996] for chromatic trees. We show that an updater owns every imbalance. We say that an updater is working on a concurrent red black tree (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.

Linearizability: Because we use the same technique for traversals and rotations as Bronson et al. [2010] we will not include the proof for linearizabilty here; this leaves only 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 start from a valid partially external red black tree; then a series of updates occur. We prove that at any time thereafter if there are no updaters working on the tree, it is a valid red black tree. While there are updaters working on the tree, we will call it a CRBT, to show that it may have one or more imbalances due to the updates being applied and that there are updaters working on the tree.

W shall show that whenever a rebalancing transformation is applied in a CRBT T it will result a tree T' with T'<T.

First we show that the red-red imbalances can be removed by using the transformations without creating new double black in the tree. We then address double black imbalances and cases where both red-red and double black imbalances are solved by the rebalancing operation.

We show that we can also solve imbalances where red-red imbalances and double-black imbalances collide.

Now we will show that we can remove double black imbalances.

Finally we address the case where we must first rotate the red sibling and then atomically apply a different transformation.

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

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. ——

Lemma 0. Let T be a CRBT with n 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 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 afterwards. 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 to at most one imbalance. Likewise every insertions or removal produces at most one imbalance associated to the updater doing the operation. An updater after inserting or removing a node is associated to 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 figure 2.a. ——

Lemma 1. 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 a CRBT if there are still imbalances in the tree or T' is 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 Lemma 0 each imbalance has an updater associated to it and so T' is a valid CRBT.

Lemma 2. 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 node 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

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

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 imbalance 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.b.c.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.

Lemma 3. 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, case2-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 is 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 this occurs for a imbalance

i'=(n',v') the number of nodes in subtree 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 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') <= o(T) and od(T') <= 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.——

Lemma 4. 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 may remove the imbalance or pushes it higher up in the tree. Thus in this case o(T') <= o(T) and od(T') < od(T). ——

Lemma 5. 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. ——

Lemma 6. 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'* or(*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 2. Given a CRBT, any sequence of rebalancing transformations that is long enough modifies the tree into a red-black tree.

Proof. In Lemma 1 we have seen that we can always apply a transformation in a CRBT that is not a red-black tree. In Lemma 2 we have seen that any transformation modifies a tree T to 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.

6 EXPERIMENTAL 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. All three are written in Java and are the fastest concurrent dictionaries we know of. We use the same methodology of Bronson and Herlihy et al.[2003] and Bronson et al [2010].

We report the average of 6 experiments. In each experiment we vary two parameters to evaluate their influence: key range and the percentage of each of the three operations.

Changing the key range we influences contention, a smaller range increases contention while a larger range reduces it. We use four key ranges from $2x10^6$ to $2x10^3$.

By using different proportion of operations we can 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 trees these scenarios reflect how the tree reacts to a different numbers of imbalances and rotations. As the number of updates grows the tree must handle more rebalances.

Figure 10 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 left to right, 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.

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 $2x10^3$, throughput is the largest and it decreases as the key range grows in all scenarios.

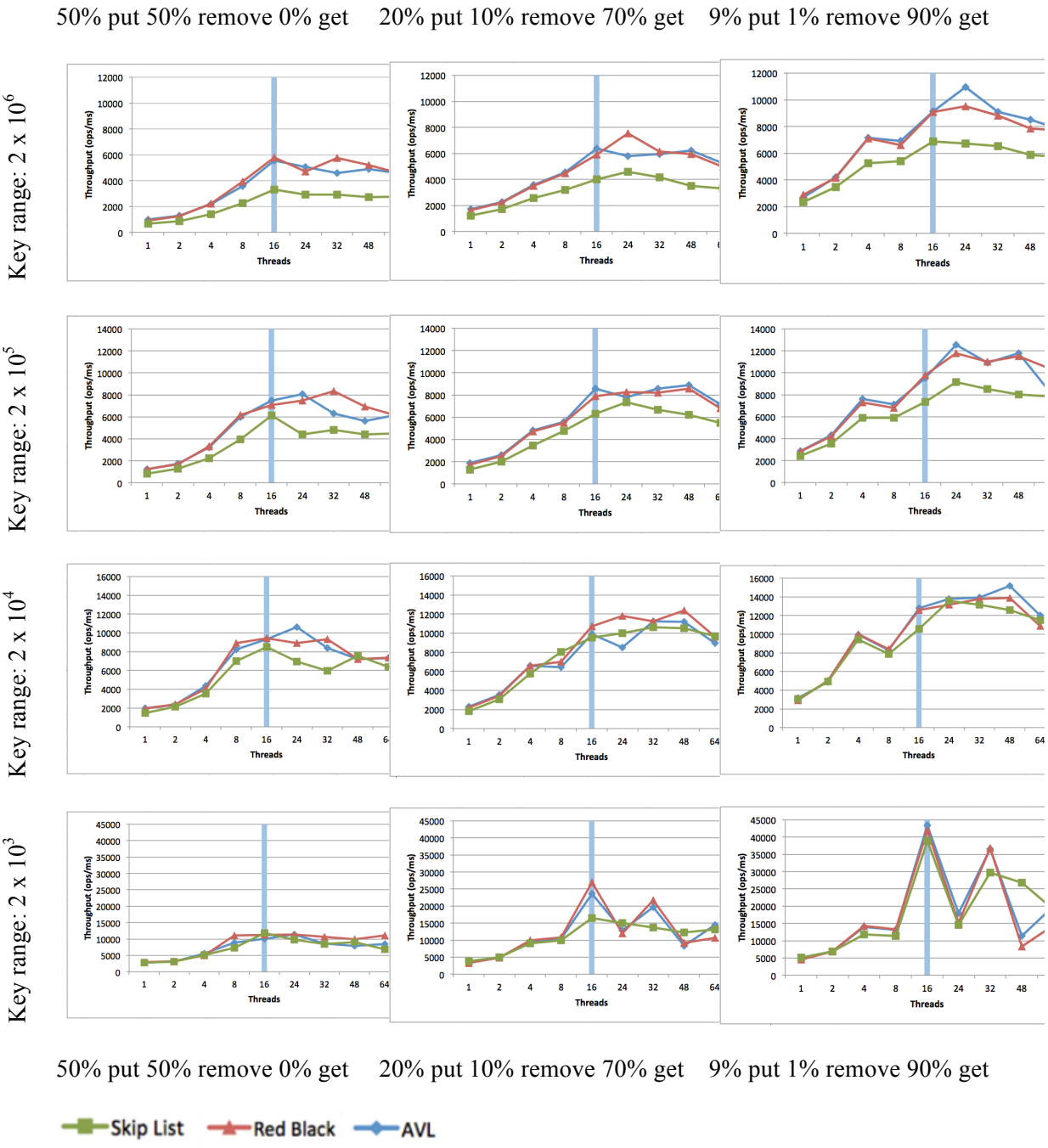

Figure 17. Results. Contention increases from bottom to top and from left to right. As contention increases the Red black tree outperforms both other solutions. In the bottom left corner it has 14% higher average throughput than the AVL tree.

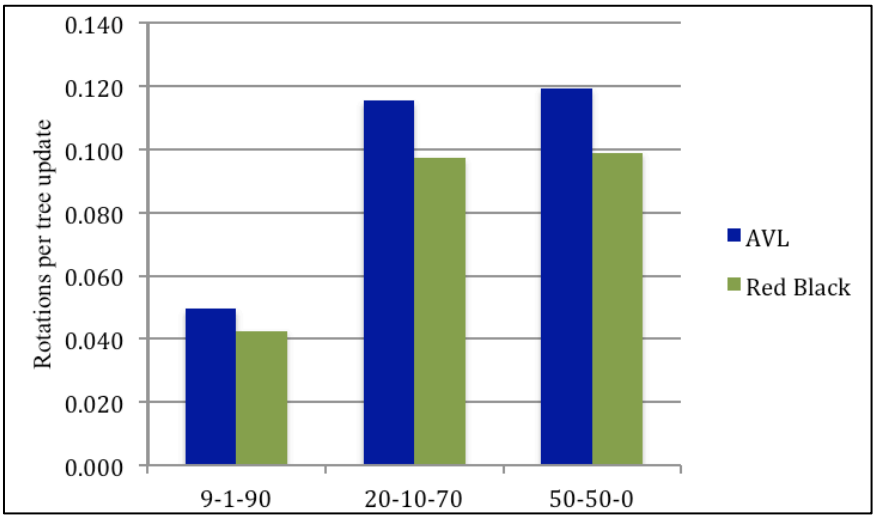

Figure 18. The red black tree consistently executes fewer rotations than the AVL tree. In each scenario the values did not vary across number of threads or key range.

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 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 (Figure 10). 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 less updates occur its superiority decreases to only 2% better at 20-10-70 until it performs 2% worse at 9-1-90. This correlates 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. As there are more updates the difference between the numbers of rotations increases in favor of the red black tree.

7 CONCLUSION

We have developed a new concurrent red black tree that supports high contention and is efficient in several scenarios. Its 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 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 hadn't considered the possibility of simultaneous imbalances in the vicinity of a node. All the operations used only affect a small neighborhood, 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.

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

Our research opens the question of what is the current bottleneck of concurrent binary search trees. 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.

REFERENCES

Boyar, J., & Larsen, K. S. (1994). Efficient rebalancing of chromatic search trees. *Journal of Computer and System Sciences , 49* (3), 667-682.

Bronson, N. G., Casper, J., Chafi, H., & Olukotun, K. (2010). A practical concurrent binary search tree. *SIGPLAN Not. , 45* (5), 257-268.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1990). *Introduction to Algorithms.* McGraw-Hill Book Company.

Guibas, L. J., & Sedgewick, R. (1978). A dichromatic framework for balanced trees. *Proceedings of the 19th IEEE Symp. Foundations of Computer Science*, (pp. 8-21). Ann Arbor, MI, USA .

Gabarro, J., Messeguer, X., & Riu, D. (1997). Concurrent rebalancing on HyperRed-Black trees. *Chilean Computer Science Society, International Conference of the , 0*.

Hanke, S. (1999). The Performance of Concurrent Red-Black Tree Algorithms. In *Algorithm Engineering* (pp. 286-300). Springer Berlin / Heidelberg.

Hanke, S., Ottmann, T., & Soisalin-Soininen, E. (1997). Relaxed Balance red-black trees. *Lecture Notes in Computer Science , 1203*, 193-204.

Herlihy, M., & Shavit, N. (2008). *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers.

Herlihy, M., Lev, Y., Luchangco, V., & Shavit, N. (2003). A Provably Correct Scalable Concurrent Skip List . *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing* (pp. 92-101). New York, NY, USA: ACM.

Larsen, K. S. (1998). Amortized constant relaxed rebalancing using standard rotations. *Acta Informatica , 35* (10), 859-874.

Lea, D. (2000). *Concurrent Programming in Java Second Edition.* Boston: Adisson-Wesley.

Nurmi, O., & Soisalon-Soininen, E. (1996). Chromatic binary search trees. *Acta Informatica , 33* (5), 547-557.

Sedgewick, R. (2008, September). *Left-leaning Red-Black Tree.* Retrieved from www.cs.princeton.edu/rs/talks/LLRB/LLRB.pdf.

Shavit, N. (2011, March). Data structures in the multicore age. *Commun. ACM* , 76-84.