12-1-2014

Concurrent Localized Wait-Free Operations on a Red Black Tree

Vitaliy Kubushyn
University of Nevada, Las Vegas, vkubushyn@gmail.com

CONCURRENT LOCALIZED WAIT-FREE OPERATIONS ON A RED BLACK TREE

by

Vitaliy Kubushyn

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2014

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Vitaliy Kubushyn

entitled

Concurrent Localized Wait-Free Operations on a Red Black Tree

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Ajoy Datta, Ph.D., Committee Chair

John Minor, Ph.D., Committee Member

Yoohwan Kim, Ph.D., Committee Member

Venkatesan Muthukumar, Ph.D., Graduate College Representative

Kathryn Hausbeck Korgan, Ph.D., Interim Dean of the Graduate College

December 2014

Abstract

A red-black tree is a type of self-balancing binary search tree. Some wait-free algorithms have been proposed for concurrently accessing and modifying a red-black tree from multiple threads in shared memory systems. Most algorithms presented utilize the concept of a "window", and are entirely top-down implementations. Top-down algorithms like these have to operate on large portions of the tree, and operations on nodes that would otherwise not overlap at all still have to compete with and help one another.

A wait-free framework is proposed for obtaining ownership of small portions of the tree at a time in a bottom-up manner. This approach allows operations interested in completely disparate portions of the tree to execute entirely uninhibited. Insert and search operations on a red-black tree are shown, and the different use cases explained.

Acknowledgements

I would like to thank my wife, Brianne Dalton, for rolling her eyes every time I made another excuse to procrastinate on my thesis, keeping me motivated to finish. Additionally, I drew a lot of perseverance and motivation from my family's belief in me. Finally, I would like to thank Zappos for being a great place to work while continuing education.

Vitaliy Kubushyn

University of Nevada, Las Vegas
December 2014

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

As technology progresses, CPU design is evolving from single large core systems to systems with multiple small cores. In order to utilize the great power of these new systems, software must evolve alongside them. Many software algorithms and methodologies have been developed to maximize the utilization of such systems. Programs operating in this space must ensure that any data changes enacted by them are consistent, and behave in a predictable way from a user's perspective.

Traditionally, data consistency across multiple processes has been achieved using locking algorithms. These algorithms effectivelly stall all but one thread from operating on a particular piece of data in the system, thus ensuring that no data or algorithmic inconsistencies result. These types of algorithms usually perform poorly if the contention for the piece of information locked is high. They also cause threads to "spin", performing no useful work while waiting for their turn in the lock. If a particular thread crashes while inside the lock, there are usually no mechanisms in place to free the lock and let other threads in, causing the whole system to grind to a halt [HS08].

Lock-free algorithms avoid these issues. Most commonly, lock-free algorithms use the **compare-and-set** operation [HS08]. The compare-and-set operation is available on most modern CPU architectures. Given a value x, a compare value y and a memory location m, a compare-and-set operation compares x to the actual memory value of m and if equal, sets m to store the value y and returns the boolean value of *true* if successful, and *false* otherwise. This operation is executed **atomically**, meaning that no other thread may interrupt the execution of this operation. This extremely powerful tool allows many algorithms to operate without locks.

A binary search tree is a data structure that is omni-present in modern computer science [CSRL01]. A red-black binary search tree is a self-balancing binary search tree in which insert and delete operations actually leave the tree in a roughly balanced state. Due to this property, red-black trees have made their way into many frameworks and pieces of software.

A significant amount of work on concurrent binary tree algorithms has been published. Ellen *et al.* proposed a framework for a non-blocking binary search tree [EFRvB10]. Kim *et al.* proposed a lock-free algorithm for a bottom-up implementation of operations on a red-black tree [JHKG06]. In their algorithm, in order not to interfere with one another, operations create a buffer space between the subtrees they operate on by setting ”intention markers” on nodes above their current working window. Operations in relative proximity would conflict with one another, and the operation that gets to claim the nodes marked with the intention markers would get to proceed first. Their framework did not have a concept of helping, however, and any operation that lost the contention to another would simply have to wait until the path became clear. This created a virtual lock.

The great difficulty of creating wait-free algorithms for a red-black tree comes from the nature of its operations. A transformative operation on a red-black tree, depending on the tree configuration at the time, could terminate in one transaction, or keep climbing the tree while executing transactions, eventually making it all the way to the root node. A concurrent algorithm has to account for all of these ”rebalancing” transactions. In addition, a sequential operation on a red-black tree goes in a bottom-up fashion. It first finds the node that needs to be deleted or the parent of the node to be inserted, and then acquires the operational subtree around it. This subtree is then transformed in some fashion in order to move the operation upward toward its conclusion. This forces the tree data structure to maintain parent pointers in its nodes. This is a problem for wait-free algorithms. The conventional method for dealing with transformations on a tree data structure in a wait-free environment is the practice of creating the modified subtree in memory and applying a single compare-and-set operation to the subtree's root's parent's down pointer to swap in the new subtree. If up pointers are maintained, this is no longer atomic, and up pointers from child nodes may need to be fixed.

R. E. Tarjan proposed a framework for applying the operations of a red-black tree in a completely top-down manner [Tar85]. This way of modifying a red-black tree opens traditional avenues for wait-free algorithms. Natarajan *et al.* used this top-down framework to create a fully wait-free concurrent algorithm for insertion and deletion on a red-black tree [NSM13]. In their approach, every transformative operation must first claim the root node's ”window”. If it finds this window already claimed by another operation, the original operation helps that operation move out of its way. This creates contention on the root node for every thread. Their algorithm is wait-free because a thread never ”spins” while doing no useful work. Any thread blocked from completing its operation is capable of removing that block by ”helping” the obstructing thread.

In this paper, we propose a **bottom-up** wait-free algorithm for red-black tree insertion. Just like [JHKG06], the algorithm proposed uses conventional red-black tree transformations to modify the tree. Unlike [JHKG06], the algorithm implements ”helping” blocking operations and competition for nodes through the use of a priority counter. The introduction of the *edge* data structure allows adjacent operations to execute concurrently . In the proposed algorithm, just like in [NSM13], search operations may execute concurrently with transformative operations, and run without executing any writes on the shared memory. Unlike [NSM13], insert operations do not need to acquire ownership of the root node. Each operation in the algorithm presented **only** acquires the nodes that it needs to perform the operation. If an operation can be performed in a single step, it does so and exits the algorithm, unless competition with another operation occurs.

Chapter 2

Red Black Tree

2.1 Description

Red-black tree or a dichromatic tree [GS78] is a special type of binary tree with some very specific rules that together guarantee that the tree will remain balanced through any number of *modify* operations.

The rules are as follows:

1. We give each node a color. A color is simply a marker for the node. A node can be either red or black.

2. The root node is always black.

3. All leaves are black. For our algorithm, or any other red-black tree algorithm, this rule is not entirely important. The leaf nodes here are represented by the left and right child pointers of every last node that actually has a value. These ”leaf” nodes in this implementation, like many others, are simply equal to NULL and are *assumed* to be colored black.

4. Every child of a red node must be black. In practice this means that a child of a red vertex is a valid black vertex, or it is NULL, which effectively means that it is a black-colored leaf node.

5. This is probably the most important rule. Every path from any node to any of its leaves must contain the same number of black nodes. Note that ”leaves” here means the NULL pointers of the furthermost node that actually has a value and not the node itself.

From here on out, we will use the term *leaves* to denote the last node with a valid value, rather than it’s NULL children.

These rules work together to maintain a tree that is roughly height balanced. In practice, this generates a tree where there is plus or minus one node in any path from the root to any of the leaves.

2.2 Search Operations

A search operation on a red black tree behaves exactly as it would on a regular binary search tree. It will traverse the tree, taking the left path if the value it is looking for is smaller, and the right path if larger, than the value attached to the node currently being considered.

2.3 Modify Operations

Modify operations are those operations that actively change the layout of the tree - insert and delete. It is important to note that a modify operation is *not* a single step operation. When we execute particular cases of a modify operation, we are left with a tree whose state does not abide by the red black tree rules enumerated earlier. Thus, we must continue the operation further up towards the root of the tree until we are confident that a valid tree state has been achieved.

We will delve more into the concrete implementation of these types of operations during the discussion of the algorithm.

2.4 Java Atomic

Algorithm implementations regularly rely on special atomic CPU instructions in order to achieve wait-freedom. There are different operations implemented in different processor architectures.

The most common of these is the Compare-and-Set atomic operation, or CAS for short. This operation allows a program to compare a value and depending on the boolean result to set its own value in one atomic step. This atomicity allows competing executions to operate on the same block of data at the same time without acquiring a lock.

A convenient implementation of this architecture is the Java Atomic package [Ora14]. It provides objects that allow CAS operations to be executed without worrying about the underlying CPU architecture. The implementation of this algorithm uses the **AtomicReference** and the **AtomicInteger** classes of this package.

Chapter 3

Algorithm

3.1 Data Structure

First and foremost, it is important to understand the data structure that the algorithm will operate on. There are several important attributes of the data structure that differ from most other tree implementations.

- The tree is bi-directional because we are acquiring ownership of locks in a bottom-up fashion.
- The tree structure has a concept of an **Edge**, allowing for operations adjacent to one another to operate concurrently.
- The tree maintains a SENTINEL node above the root node, to allow concurrent CAS operations to execute for the first node to be added.

Now lets examine the inner workings of **Node** and **Edge**.

Node

A Node object maintains an object reference to its parent edge and its right and left child edges. It also maintains a *value* numeric object that represents a key that a search operation will use to traverse the tree and find its target node. The *value* does not necessarily have to be a number. It can be any object that can be compared with another of its type to determine further direction of traversal.

Edge

An Edge object maintains two AtomicReference objects. AtomicReference type is used in Java to denote a reference to a different object that may be modified by compare-and-set operations. The two references maintained are up and down, referencing the parent and the child connected by this edge respectively.

Please refer to Figure 3.1 for a graphical representation of this data structure.

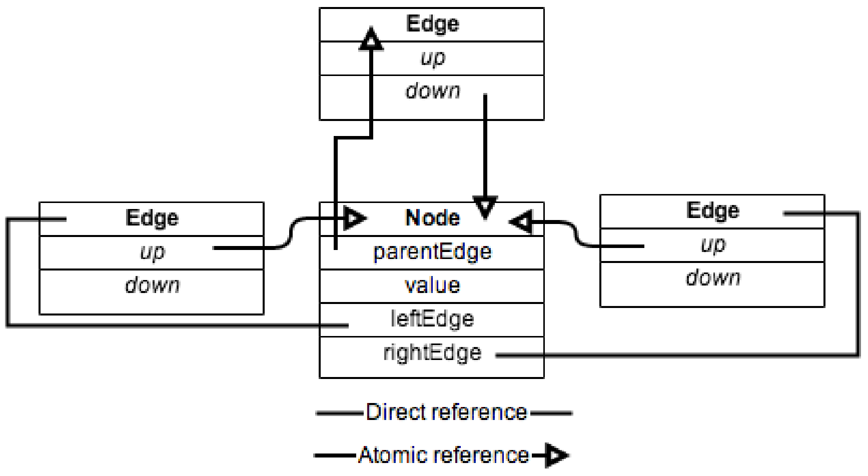


Figure 3.1: The general outline of a node and its surrounding edges.

3.1.1 Operation Statuses

During the execution of a modify operation, an execution may have one of the following statuses.

LOCKING

An operation always begins in this status. When an execution is in this status, it indicates that the operation is still in the process of claiming all of the nodes it will need to complete its function.

LOCKING_REBALANCE

In some instances, when an operation completes, it leaves the tree in a state that may possibly break some of the red-black rules described earlier. In this case, the root of the newly transformed subtree automatically receives a LOCKING_REBALANCE status that will cause the operation to continue forward with a **rebalance** operation, fixing the state of the tree. A LOCKING_REBALANCE operation always has a higher priority than a LOCKING operation, and competes with other LOCKING_REBALANCE operations not only based on priority, but also on their relative positions in the tree.

OVERRIDEN

An operation can only enter this state when a different, higher priority operation needed to claim the same node as the original operation. In this case, the winner of the contention marks the loser OVERRIDEN and moves on with its operation. Only operations in LOCKING or LOCKING_REBALANCE states can be overriden.

LOCKED

An operation is LOCKED when it has acquired all of the nodes necessary to complete its function, and is ready to perform the swap.

FINALIZED

An operation is FINALIZED when it has completed its operation and swapped in the new, modified tree.

DISCARDED

A discarded operation is treated as though the node the operation was attached to didn't have any operation at all. A LOCKING or LOCKING_REBALANCE operation can become discarded when it loses a competition for a node to a different execution and goes on to help that execution finish. After the winner has been helped, the OVERRIDEN operation gets DISCARDED and the LOCKING operation is retried. In the case of a LOCKING_REBALANCE operation, the DISCARDED operation may be immediately "revived" as a new identical operation, only attached to the action node of the original DISCARDED operation.

3.1.2 OperationMark

An OperationMark is something an operation can attach to a node in order to signify exactly for what purpose that node is to be used for. These correspond directly to the positions of the nodes in the subtree the operation will lock and transform.

For insert operations, these are INSERT_PARENT, INSERT_GRANDPARENT, INSERT_UNCLE, and INSERT_REBALANCE. INSERT_REBALANCE refers to the node at the bottom of the tree, where an insert node would be, during a rebalance operation.

In a delete operation, the subtree formed refers to the nodes around the **swap node** - the node that will replace the node to be deleted. The operation marks, therefore, are DELETE_ME, referring to the node to be deleted, DELETE_SWAP, referring to the node to be swapped in the deleted node's place, DELETE_PARENT, DELETE_SIBLING, DELETE_NIECE_LEFT, DELETE_NIECE_RIGHT, and DELETE_REBALANCE for fixing an inconsistent tree.

3.1.3 Action Node

An action node is the first node claimed by any operation. For an *insert* operation, this is the node under which the new node will be inserted as a child, marked with an INSERT_PARENT operation mark. For a *delete* operation, this is the node being deleted, marked with a DELETE_ME operation mark. For a *rebalance* operation, this is the root node of the tree swapped in during the last iteration of a rebalance, either an INSERT_REBALANCE or a DELETE_REBALANCE.

3.1.4 OperationInfo

A concurrent algorithm has to provide a method for threads to compete for resources with one another, and claim victory or defeat on those resources. However, when the resources in question consist of an unspecified number of nodes in a tree that is constantly changing shape and form, the problem becomes rather difficult.

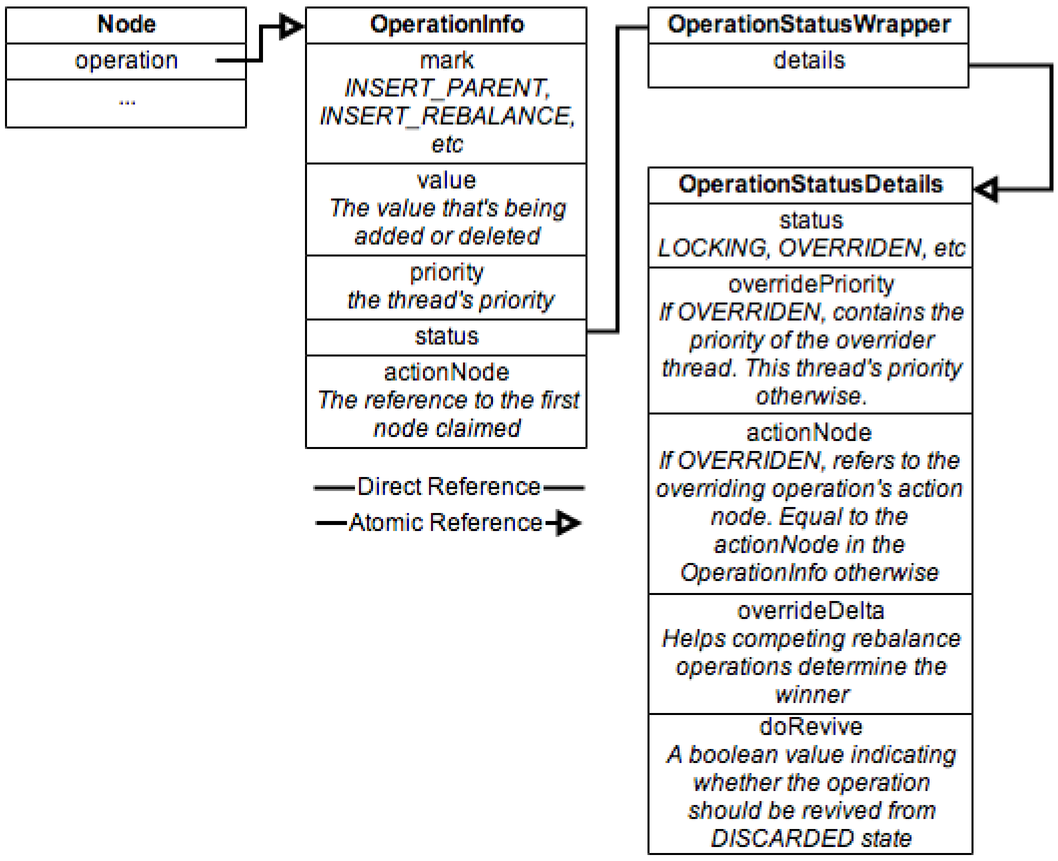
There are several particular requirements that are absolutely necessary for threads to compete with one another on a node by node basis.

- An operation must be able to claim each vertex of the tree independently of any other vertex. In this way, threads can compete on an individual node basis, whenever collisions occur.
- An operation must be able to override another, already executing operation. This means that the override must apply to every node currently claimed by the overriden thread **atomically**. If it does not happen atomically, complex synchronization problems arise.
- If an operation is unable to override another, it must be able to help that operation finish. This is the only way to avoid locking in multi-threaded, contentious system.

An **OperationInfo** data structure is proposed to address the requirements described above. OperationInfo is a container for OperationMark, OperationStatus and OperationStatusWrapper. Please refer to Figure 3.2 for a graphical representation of the OperationInfo data structure.

Each node claimed by an operation has a **different** OperationInfo. However, each operation only creates one OperationStatusWrapper object that is shared between every OperationInfo the execution attaches to any node. In this way, a modify operation can claim nodes one at a time by attaching a different OperationInfo to each vertex in turn. At the same time it can change the status of every node that it has claimed **atomically**, by creating and applying a CAS operation to the OperationStatusWrapper's *details* AtomicReference.

This architecture allows competing threads to go from LOCKING to LOCKED and LOCKED to FINALIZED atomically. This design also permits executions to set other threads' statuses OVER-



Node
operation
...
OperationInfo
mark
INSERT_PARENT, INSERT_REBALANCE, etc
value
The value that's being added or deleted
priority
the thread's priority
status
actionNode
The reference to the first node claimed
OperationStatusWrapper
details
OperationStatusDetails
status
LOCKING, OVERRIDEN, etc
overridePriority
If OVERRIDEN, contains the priority of the overrider thread. This thread's priority otherwise.
actionNode
If OVERRIDEN, refers to the overriding operation's action node. Equal to the actionNode in the OperationInfo otherwise
overrideDelta
Helps competing rebalance operations determine the winner
doRevive
A boolean value indicating whether the operation should be revived from DISCARDED state
Direct Reference
Atomic Reference

Figure 3.2: The structure of the OperationInfo data.

RIDEN and DISCARDED without the need to set the value on each claimed node in turn. This approach also enables threads to help one another. An operation can ”steal” an OperationStatusWrapper from a vertex locked by a different operation it is attempting to help, and effectively ”become” that other operation, traversing the same nodes and applying the same OperationStatusWrapper to any OperationInfo it attempts to set.

Figure 3.3 graphically shows a common scenario that arises when an operation has claimed all of the nodes it needs and is ready to lock itself in order to perform the transformation. In this case, simply performing a CAS call on the *details* AtomicReference of the StatusWrapper object changes the status of every node claimed. The new StatusDetails, connected to the StatusWrapper with the red line will become the current status of the operation, while the old StatusDetails will become de-referenced and garbage collected.

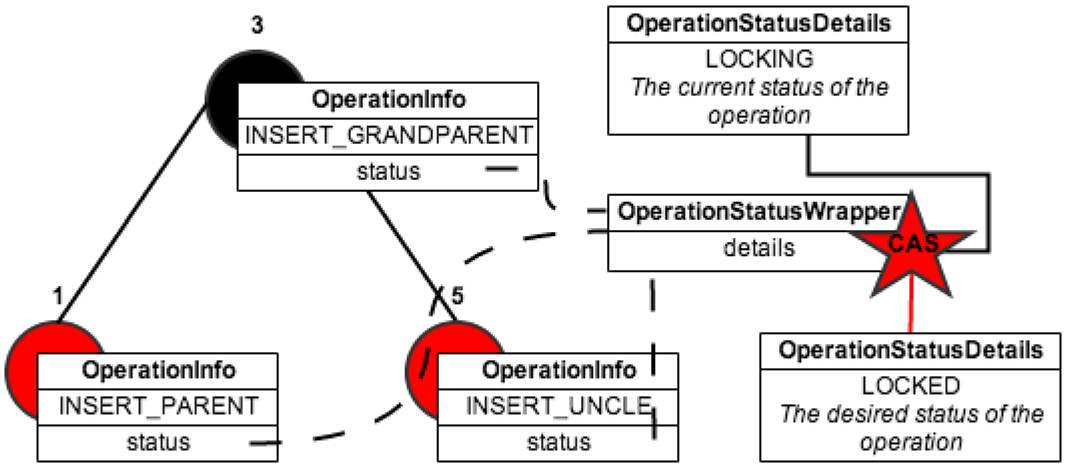


Figure 3.3: Changing the status of an operation.

3.2 Operation Phases

The algorithm consists of several execution phases.

Search Phase

During this phase, an operation finds a place in the tree where the new node should be inserted.

Locking Phase

During this phase, an operation locks all the nodes necessary for it to be able to complete its operation. It competes with other threads, overriding or helping them depending on the outcome of the competition. If a LOCKING operation loses competition for a node, it will return to the search phase after helping the winner thread. A LOCKING_OVERRIDEN operation will be revived upon finishing helping its overrider complete.

Transformation Phase

During this phase, an operation creates a transformed copy of a subtree it has locked, and swaps it in using CAS operations. During this phase, an operation may not be overriden by another.

3.2.1 Search Phase

The search phase is the most straightforward of all the phases. It operates in a fashion almost identical to a simple search operation.

The operation will start at the SENTINEL node and traverse the tree downwards, as if searching for the node that it is attempting to add. If the operation actually finds a node with the same value, an exception is raised indicating that the node already exists in the tree. Otherwise, the leaf node under which the new node should be added is returned.

It is possible that the search operation will traverse some nodes that are no longer part of the tree. This can happen if a concurrent operation swaps in a new subtree over the node the search algorithm was interrupted on. If this is the case, and the parent node the operation finds is no longer part of the tree, then its OperationInfo will indicate the node as FINALIZED. Because of this, the **Locking Phase** will not be able to lock the node, and the search will be restarted.

This restarting behavior can cause an operation to continuously find nodes that it cannot lock and starve. This paper does not present a good solution for dealing with this problem. However, in practical applications, any operation should be able to acquire its starting node in short order. This problem, along with others, will be discussed in the Future Work section.

Algorithm 1 presents the method for finding the parent node under which to insert.

3.2.2 Locking Phase

During the Locking Phase, an operation executes the Node Lock Protocol on every node it needs for the completion of its work. During this time, if another thread is competing for the same node, the operation will either override the competitor, or help it complete its task, depending on the outcome of the competition. If the operation helps its competitor, it will discard all the nodes it has locked by setting their status to DISCARDED and try again starting from the Search phase. The operation will retain the same priority it started with, making it less and less likely that it will lose a competition to any other thread.

Algorithm 2 describes the kickoff of the Locking Phase.

Once the parent node is found and OperationInfo has been created, the algorithm will attempt to lock all the other needed nodes for the operation. This is accomplished in the AddNodeLockTree, which maintains a reference to the OperationInfo and its expected StatusDetails object.

For LOCKING operations, the algorithm will only execute once, unless it finds itself LOCKED by a helping thread. If a helping thread has LOCKED the operation, it will restart acquiring nodes from its action node, making sure that the tree it has acquired is identical to the one that the helping thread has LOCKED. Once the operation collects all those nodes, it has the power to take the operation from LOCKED to FINALIZED.

An operation will help its overrider, discard itself, and then break out of its iteration with a value of **false** if the operation was OVERRIDEN by another operation, indicating that operation

```
public Node searchForInsertNode (int value, Node startNode)
{
    input  : value: key of the node to insert.
             startNode: the root node of the tree to search.
    output: The parent node to insert under. SENTINEL node if the tree is empty. Throws
             an AlreadyExistsException if the node already exists in the tree
    Node current ← startNode;
    Node found ← null;
    while (found == null  &  current ≠ null)
    {
        if ((current ≠ SENTINEL  &  current→value == value)
        {
            throw AlreadyExistsException;
        }
        else if ((current == SENTINEL  &  current→downEdge == null) |
        (current→value < value  &  current→rightEdge == null) |
        (current→value > value  &  current→leftEdge == null))
        {
            found ← current;
        }
        else if (current == SENTINEL)
        {
            current ← (SENTINEL→downEdge);
        }
        else if (current→value < value)
        {
            current ← (current→rightEdge);
        }
        else if (current→value > value)
        {
            current ← (current→leftEdge);
        }
    }

    return found;
}
```

Algorithm 1: Searching for insert node.

```
public void add (int value)
{
    input: value: key of the node to insert.
    priority ← atomicallyGetNewPriority();
    while (true)
    {
        Node parent ← searchForInsertNode(value, SENTINEL);
        if (parent == SENTINEL)
        {
            // Empty tree, try adding immediately
            Node n ← (new black node with value);
            if (SENTINEL→downEdge.cas(null, n))
            {
                // If we succeeded, then our node is the new root.
                return;
            }
        }
        else
        {
            OperationInfo info ← createNewInfo(INSERT_PARENT, value, priority);
            StatusDetails details ← createNewDetails(LOCKING, parent);
            info.statusDetails ← details;
            // We provide the details separately from the info to indicate what
            // details we expect to see when performing the locks.  If the
            // expected details don't match the details currently on the tree,
            // we know that someone has modified our status for us.
            AddNodeLockTree lockTree ← newLockingPhase(info, details);
            if (lockTree.assembleAndContinue())
            {
                return;
            }
        }
    }
}
```
Algorithm 2: Inserting a node initalization.

should start over from the Search phase. Repeated cases of this may cause a thread to starve. This paper does not present a solution to this problem. However, in practical applications, this should not happen.

A LOCKING_REBALANCE operation that finds itself OVERRIDEN will help its overrider. Once the helping is complete, the LOCKING_REBALANCE operation will revive itself using the revive algorithm described in Algorithm 7. The operation will then assume the OperationInfo used for reviving as its own, and attempt to finish that operation instead.

Once the operation completes assembling all of the nodes it requires, it will attempt to lock them using a CAS operation. If successful, it will perform the operation by utilizing the constructed **AddNodeOperationTree**. Otherwise, depending on what the new details of the operation are, it may help its overrider, continue looping, return as success or failure.

Algorithm 3 describes the initial stages of the lock algorithm. During this stage, if the operation is a rebalance operation at the root of the tree, it can safely repaint the action node black and exit. No other operation may override the current operation, because all other rebalance operations are lower on the tree. It is safe to repaint the root black, because it adds one black node to **every** path to every node in the tree, thus conserving the red-black tree properties.

If the operation is not a rebalance operation at the root of the tree, all the needed subtree nodes are assembled and Algorithm 4 is invoked to take the operation further.

Algorithm 3 also sets up the locking loop. Line 31 sets up the exit condition. For rebalance operations, the loop is continued until the operation has succeeded, as rebalance operations may not be discarded at any point at the risk of unbalancing the tree. For simple LOCKING operations, the operation may terminate if it finds that it fails the CompareAndSet operation on its assembled tree of nodes. This can signify two situations.

1. A thread may have overriden the current operation, in which case the current operation's status will be OVERRIDEN or DISCARDED, and the operation should help its overrider, discard itself and either break out of the loop to try again from Algorithm 2, or revive itself and restart from the revived node.

2. A thread may have taken upon itself to help the current thread, in which case the new status is either LOCKED or FINALIZED. FINALIZED indicates that the helping thread has actually taken the current operation to the end, in which case the current thread will not attempt a tree swap. LOCKED indicates that another thread may have helped the operation and locked the subtree. Because the operation is not yet FINALIZED, we cannot help the overriden nodes finish yet. It is possible that the helping operation has assembled a different set of nodes than the current operation. Because of this, we do not attempt to swap in our constructed tree.

We let the while loop make another iteration in order to construct a subtree equivalent to the one that the thread that actually took the status to LOCKED is operating on.

Algorithm 4 presents the pseudo-code for actually changing the status of a gathered subtree to a LOCKED state by a thread performing an operation. Please refer to Figure 3.3 for a graphical representation of this process.

Handling Directly Overriden Nodes

Please note the call to the *handleDirectlyOverriden()* method in algorithm 4. Algorithm 5 shows pseudocode of this method. The algorithm makes sure that none of the nodes it has **directly** overriden retain the OperationInfo of the overriden operation. This ensures that some complicated corner cases don't have a chance to arise when nodes of overriden operations are used in an operation tree of the overrider. Figure 3.4 shows this process in action.

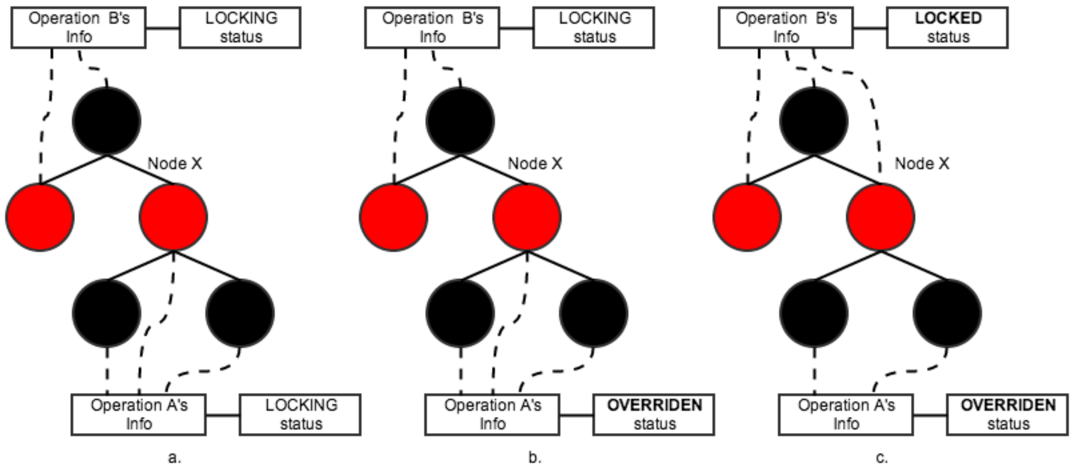


Figure 3.4: Handling directly overriden nodes. a.) Operation A has acquired node X. b.) Operation B overrides operation A. c). Operation A sets its own OperationInfo on Node X, thereby taking it away from A.

Algorithm 9 decribes the actions needed to actually swap in the new tree. There are two items here worth considering - carrying forward a possible rebalance operation, and continuing on with a rebalance operation after a discard and a revive.

Carrying Forward a Possible Rebalance Operation

It is possible that after we swap in a new subtree for the one we have LOCKED and FINALIZED the tree will be in a state that breaks one or more properties of red-black trees. In these cases, the

```
public boolean assembleAndContinue
{
    output: True if successfully locked and finalized. False otherwise.
    Node parent ← (globalScope.opInfo→actionNode);
    Node rebalance ← null;
    boolean result ← true;
    repeat
    {
        if (globalScope.opInfo→rebalanceNode ≠ null)
        {
            rebalance ← (globalScope.opInfo→rebalanceNode);
            parent ← (rebalance→parent);
            if (parent == SENTINEL & rebalance→color == RED)
            {
                // A rebalance operation that's reached the root of
                // the tree is a special case and can be handled very easily.

                // We don't care if we succeed or not because a rebalance on the
                // root node cannot be overriden
                (globalScope.opInfo→status).cas(globalScope.expectedDetails,
                newDetails(LOCKED));

                Node newRoot ← copyInBlack(rebalance);
                if (SENTINEL→downEdge.cas(rebalance, newRoot)
                {
                    // Fix up pointers of the children
                    (newRoot→rightEdge).cas(rebalance, newRoot);
                    (newRoot→leftEdge).cas(rebalance, newRoot);
                }
                return true;
            }
        }
        // AddNodeOperationTree is responsible for actually performing the
            operation
        AddNodeOperationTree operationTree ← assembleAddNodeOperationTree(parent);

        result ← lockAndContinue(operationTree, parent, rebalance);
    }
    until ((not result & globalScope.opInfo→rebalanceNode == null) |
    globalScope.expectedDetails→status == LOCKED);
    return result;
}
```

Algorithm 3: Initial stages of the lock algorithm.

```
public boolean lockAndContinue (AddNodeOperationTree operationTree, Node parent,
Node rebalance)
{
    output: True if successfully locked and finalized. False otherwise.
    attemptTreeSwap ← true;
    // A tree is considered constructed when it has been completely assembled
    // and was not found to be overriden or otherwise in a bad state.
    // A tree is considered finalized if the last time the nodes were checked
    // the status details were in a FINALIZED state.
    if (operationTree.isConstructed() & not operationTree.isFinalized())
    {
        StatusDetails newDetails ← newDetails(LOCKED, parent, rebalance);
        if (globalScope.expectedDetails→status == LOCKED)
        {
            newDetails ← globalScope.expectedDetails;
        }
        if (not (globalScope.opInfo→details).cas(globalScope.expectedDetails, newDetails))
        {
            // We were not successful, lets examine the new details.
            newDetails ← (globalScope.opInfo→details);
            if ((newDetails→status) == OVERRIDEN | (newDetails→status) ==
            DISCARDED)
            {
                if (newDetails→status == OVERRIDEN)
                {
                    help(newDetails→actionNode);
                }
                handleDiscardRevive(globalScope→opInfo, newDetails);
            }
            attemptTreeSwap ← false;
        }
        globalScope.expectedDetails ← newDetails;
        handleDirectlyOverriden(operationTree);
        if (attemptTreeSwap)
        {
            swapTree(operationTree);
        }
    }
    else if ((globalScope.opInfo→details→status) == DISCARDED)
    {
        handleDiscardRevive(globalScope.opInfo, globalScope.opInfo→details);
    }
    globalScope.expectedDetails ← (globalScope.opInfo→details);
    return globalScope.expectedDetails == FINALIZED;
}
```

Algorithm 4: Locking the subtree.

```
public void handleDirectlyOverriden (AddNodeOperationTree operationTree)
{
    output: None.
    if (globalScope.expectedDetails→status == LOCKED)
    {
        // We have to set any nodes that we've directly overriden
        // to have our operation attached due to the intricacies of the
        // the way rebalance operations help ops they've overriden.
        foreach (Node n in operationTree)do
        {
            OperationInfo info ← (n→info);
            StatusDetails details ← (info→details);
            if (details→status == OVERRIDEN)
            {
                // Set the reference to our info for the node.
                (n→info).cas(info, globalScope.opInfo);
            }
        }
        end
    }
}
```

Algorithm 5: Handling directly overriden nodes.

root node of the newly swapped in tree will already have an OperationInfo attached, marking the new root as the **rebalance** node of a new operation. In this case, the current operation will carry its root's rebalance operation onwards, locking a new subtree starting from its new root.

If two threads are both racing to complete the same operation, then only the winner of the race will have a rebalance tree to carry forward. Thus, it is possible that the thread that was simply helping the actual progenitor of the operation will be the one to carry onwards the rebalance operations. In this case, the progenitor thread will simply return a success to the end user of the tree.

Reviving a DISCARDED Rebalance Operation

Rebalance operations can never simply be restarted, because their existence indicates that the tree is in a state that is inconsistent with the Red-Black tree properties outlined previously. Therefore, a DISCARDED rebalance operation must not be lost.

When an operation discards another operation, or itself, it applies a CAS operation to the "details" atomic reference of the OperationStatusWrapper, replacing the details with a new object marked DISCARDED. If an operation being overriden is a rebalance operation, the new details swapped in contain an additional flag, *doOverride*, that if true, indicates that the operation must be revived before any of the discarded nodes may be claimed. The new details also contain a reference to the DISCARDED operation's action node in this case as well. The discard procedure is described in Algorithm 6.

In order to revive an operation, a thread first clones the OperationInfo object attached to the action node of the discarded operation. It then attaches a new status wrapper with a new LOCKING_REBALANCE details. The reviver then uses CAS on the old OperationInfo object attached to the action node of a discarded operation, replacing it with the one it has cloned. The safety of the revive procedure is ensured by requiring every operation attempting to claim a discarded node to go through the revive procedure if the *doRevive* flag is true. Only one operation may succeed in using CAS to swap in the revived OperationInfo, as every other operation will either fail the CAS due to the nature of compare-and-set, or will see the node as already having been revived. Algorithm 7 describes this procedure.

Carrying Forward a Revived Operation

In Algorithm 4, *handleDiscardOverride* is invoked if we fail to assemble the whole operational subtree, or if we fail to lock it using a CAS operation, and find ourselves either OVERRIDEN or DISCARDED. Algorithm 8 describes this procedure in detail. The method helps a rebalance operation that has lost its competition and became OVERRIDEN or even DISCARDED as a result to revive itself, and assume the identity of the revived operation. The function simply performes the following checks and operations.

- If the operation is OVERRIDEN, the overrider is first helped, and the operation is then discarded. Remember that Algorithm 6 calls *reviveOperation()* for any discarded **rebalance** operation. Thus we don't have to call *reviveOperation* explicitly to revive it here.
- If the operation is DISCARDED and if the *doReviveOperation* flag is set, then it may or may not have already been revived. At this point, we call *reviveOperation* explicitly to ensure that it has been revived properly.
- Finally, if the operation is a rebalance operation, we reset the *opInfo* and *expectedDetails* pointers that we maintain to the newly revived operation.

3.2.3 Node Lock Protocol

In the sections above, we loosely mentioned assembling the AddNodeOperationTree to be locked and operated on by an operation. But how is this done, and how do the individual nodes in the

```
public void discardOperation (OperationInfo operation, StatusDetails details)
{
    output: None.
    StatusDetails discardDetails ← newDetails(DISCARDED);
    if ((operation→rebalanceNode) ≠ null)
    {
        (discardDetails→doReviveOperation) ← true;
        (discardDetails→actionNode) ← (operation→rebalanceNode);
    }
    (operation→details).cas(details, discardDetails);
    if (discardDetails→doReviveOperation)
    {
        reviveOperation(operation, operation→rebalanceNode);
    }
}
```

Algorithm 6: Discarding an operation.

```
public void reviveOperation (OperationInfo discardOp, Node actionNode)
{
    output: None.
    OperationInfo actionOp ← (actionNode→operation);
    if ((discardOp→status) == (actionOp→status))
    {
        OperationInfo reviveOp ← newOperationInfo(discardOp→value,
        INSERT_REBALANCE, discardOp→priority);
        (reviveOp→actionNode) ← actionNode;
        (reviveOp→rebalanceNode) ← actionNode;
        StatusDetails reviveDetails ← newStatusDetails(actionNode, discardOp→priority,
        LOCKING_REBALANCE);
        (reviveOp→status→details) ← reviveDetails;
        (actionNode→operation).cas(discardOp, reviveOp);
    }
}
```

Algorithm 7: Reviving an operation.

```
public void handleDiscardRevive (OperationInfo operation, StatusDetails details)
{
    output: None.
    if ((details→status) == OVERRIDEN)
    {
        help(details→actionNode);
        discardOperation(globalScope.opInfo, details);
    }
    else if ((detalis→status) == DISCARDED & details→doReviveOperation)
    {
        reviveOperation(globalScope.opInfo, details→actionNode);
    }
    if ((opInfo→rebalanceNode) != null)
    {
        globalScope.opInfo ← (details→actionNode→operation);
        globalScope.expectedDetails ← (globalScope.opInfo→details);
    }
}
```

Algorithm 8: Handling rebalance override.

```
public void swapTree (AddNodeOperationTree operationTree)
{
    output: None.
    operationTree.finalizeTree();
    if (operationTree.hasRebalanceTree())
    {
        // We're in a situation where we need to carry the operation
        // forward as a rebalance operation.
        (operationTree→rebalanceTree).assembleAndContinue();
    }
    return operationTree.isSwapped();
}
```

Algorithm 9: Swapping in the new subtree.

subtree obtain their status? This is where the Node Lock Protocol (NLP) comes in. For each node that an operation needs, the same protocol for setting the StatusDetails is invoked.

Algorithm 10 describes the way that an operation claims all the nodes that it needs. NLP has two ways to lock a node. The **lockNode** method is used to lock the first node for any LOCKING operation. The **iterativeLockNode** method is used to lock every node after the initial one. This is split in two because LOCKING operations are not capable of competing for the first node they attempt to claim.

The operation initializes a new NLP instance for each node it attempts to lock with data it will need, and simply applies the algorithm to the node. Algorithm 11 shows how the results of the NLP protocol are handled. NLP has a concept of a **shortcut** return. An NLP with a shortcut value of **true** indicates that it has found the operation already FINALIZED by someone, and therefore we no longer need to proceed with claiming nodes to lock.

In order to understand the Node Lock Protocol, it is vital to understand the way operations compete for nodes.

Determining the Winner

When an operation comes across a node that already has an OperationInfo attached to it, it first checks whether the operation that has claimed the node is in a state that can allow for competition.

When an operation overrides another, it uses a CompareAndSet operation to atomically swap out the StatusDetails of the overriden thread with a new StatusDetails that has a status of OVERRIDEN and a reference to the action node of the winner thread, be it an INSERT_REBALANCE or a INSERT_PARENT node. However, what happens when a third operation comes in and tries to claim the same node? The third contender will follow the override chain, traversing the StatusDetails of the OVERRIDEN operations to the reference to the action node of their overrider. It will then compete with the overrider of the node, and if successful, will override the overrider of the node, creating a chain of overrides that can be followed by any thread wishing to claim the nodes. Figure 3.5 shows an example of an override chain. In the figure, A, B and C are action nodes of their respective operations. Operation B has overriden operation A, and operation C has in turn overriden operation B.

The only two states in which competition is allowed is LOCKING and LOCKING_REBALANCE. A LOCKING_REBALANCE operation **always** wins over a LOCKING operation. Two LOCKING operations compete strictly based on their priorities - the operation with the lowest priority wins. The competition for two LOCKING_REBALANCE operations is more involved. Because LOCKING_REBALANCE operations can continue in a chain all the way to the root of the tree, it is

```
public void assembleAddNodeOperationTree (Node parentNode, OperationInfo opInfo, StatusDetails expectedDetails)
{
    output: The assembled AddNodeOperationTree.
    AddNodeOperationTree tree ← new AddNodeOperationTree(value ← (opInfo→value), parent ← parentNode);
    if (opInfo→rebalanceNode ≠ null)
    {
        (tree→actionNode) ← (opInfo→rebalanceNode);
        isRebalance ← true;
    }
    continueLocking ← true;
    // We attempt an override if it's a rebalance operation because its first
       node has already been acquired.
    NLP nlp ← new NLP(node←parentNode, info←createAppropriateInfo(), attemptOverride←isRebalance);
    (nlp→details) ← globalScope.expectedDetails;
    // The status wrapper is the object that will be carried over from each
       node's OperationInfo to ensure that the status details refer to the same
       object.
    (nlp→statusWrapper) ← (globalScope.opInfo→statusWrapper);
    if (isRebalance)
    {
        nlp.iterativeLockNode(isRebalance);
    }
    else
    {
        nlp.lockNode(isRebalance);
    }
    continueLocking ← handleLockResult(nlp, tree);
    Node n ← getNextNode();
    while (continueLocking & n ≠ null)
    {
        (nlp→node) ← n;
        (nlp→info) ← createAppropriateInfo();
        (nlp→attemptOverride) ← true;
        nlp.iterativeLockNode(isRebalance);
        continueLocking ← handleLockResult(nlp, tree);
        n ← getNextNode();
    }
    if (not tree.isConstructed())
    {
        tree.setConstructed(continueLocking);
    }
    return tree;
}
```

Algorithm 10: Assembling the AddNodeOperationTree using NLP.

```
public void handleLockResult (NLP nlp, AddNodeOperationTree tree)
{
    output: True if we should continue locking. False otherwise.
    continueLocking ← false;
    if (nlp→shortcut)
    {
        (tree→finalized) ← true;
    }
    else if (nlp→success)
    {
        continueLocking ← true;
    }
    return continueLocking;
}
```

Algorithm 11: Handling result of the Node Lock Protocol.

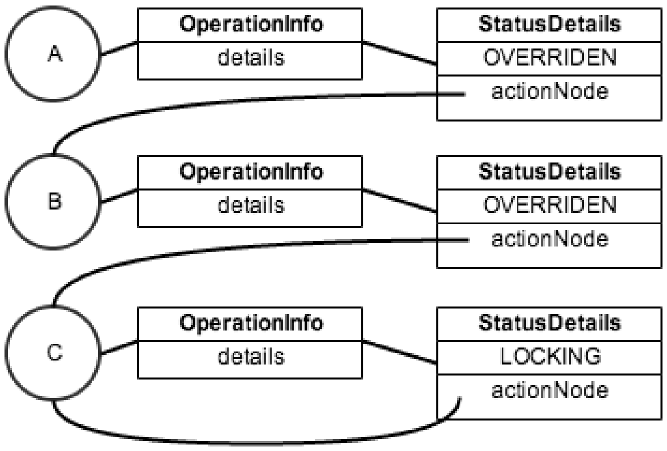


Figure 3.5: A chain of overrides.

important that the operations that are "higher" up on the tree are the first ones to complete. Thus, when two LOCKING_REBALANCE operations compete, they compete firstly on their position relative to one another, and secondly on their priorities.

Figure 3.6 shows examples of this kind of competition. Please note that INSERT_REBALACE nodes have been omitted in the figure. In part **a**, operation A, outlined in orange, will win the competition because it is "higher" on the tree, meaning that its INSERT_GRANDPARENT node is above operation B's (blue) in the tree. In part **b**, both operations are at the same level, and will therefore compete based on their priorities and not on their position in the tree.

Lemma 3.2.1 *A LOCKING_REBALANCE operation A will never* **directly override** *a node N of operation B that is marked with an INSERT_REBALANCE OperationMark.* **Directly override** *in* *this context means that the overriden node is used as part of the overrider operation's AddNodeOperationTree.*

Proof: Assume that operation A directly overrode operation B's INSERT_REBALANCE node N. This means that both A was at the same level or higher than B and A needed N in its subtree. If A needed N in the INSERT_PARENT, INSERT_GRANDPARENT or INSERT_UNCLE positions, then B was higher in the tree than A and should therefore have won the competition. If A needed N as its INSERT_REBALANCE node, then A and B are the same operation and one is helping the other. Therefore, A could not have directly overriden the node N. ■

Due to Lemma 3.2.1, we know that we never have to worry about overriding the actual INSERT_REBALANCE node of another LOCKING_REBALANCE operation. This gives us the liberty to set the status of any overriden LOCKING_REBALANCE operations to OVERRIDEN, and then revive them back to LOCKING_REBALANCE when it is safe to do so without worrying about whether the INSERT_REBALANCE node is still in the tree or not.

In practice, this is implemented by assigning a **rebalance priority** to each of the operation marks possible in an insert operation. The following are operation marks and their assigned rebalance priorities: INSERT_GRANDPARENT (4), INSERT_PARENT (2), INSERT_UNCLE (2), INSERT_REBALANCE (0). Lower rebalance priority indicates a higher priority value. If operation A has a rebalance priority of 2 for Node X and operation B has a rebalance priority of 4 for the same node, then operation A will win the competition.

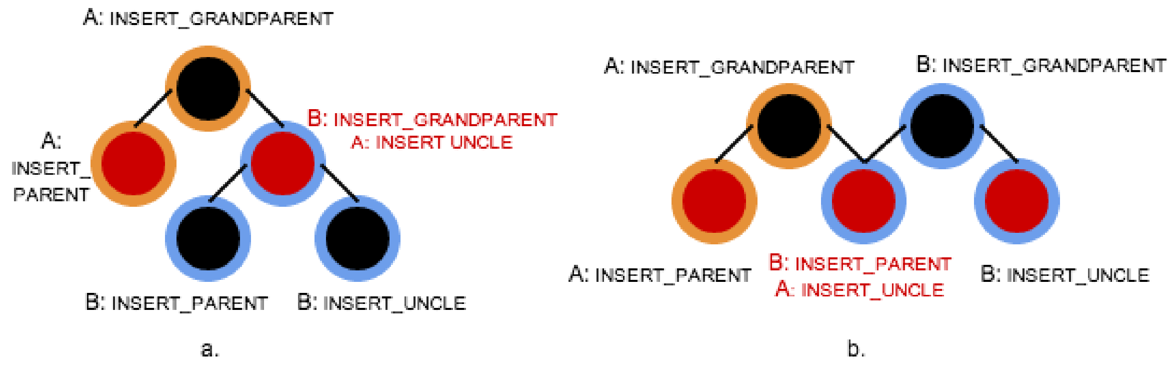


Figure 3.6: LOCKING_REBALANCE competition.

However, one problem with this approach is that operations override other operations as a whole, and not their nodes on an individual basis. An override is enacted when an operation uses a CompareAndSet operation successfully on another's StatusDetails, effectively changing the status of all the nodes that the loser operation had claimed before. So what happens in a situation when a third operation wants to compete for a node that's already been overriden?

N is an unclaimed node. Operation A comes in and claims the node by setting its OperationInfo on it, with an OperationMark of INSERT_GRANDPARENT. Now, operation B comes by, and wants to use N in the INSERT_PARENT position. Since its rebalance priority is higher, it successfully overrides A by setting an OVERRIDEN StatusDetails object on its OperationInfo. Now operation C comes by, and wants to use N in its INSERT_UNCLE position. Here, we have a problem. C knows that A wanted to use N in the INSERT_GRANDPARENT position. However, C has no way of knowing what position B wanted to use N in. C has to override B, since it needs to override the last overrider in the override chain shown in Figure 3.5, but does not have enough information to compete because it doesn't know what position B intends to use N in.

This is solved by introducing the concept of an **override delta**. An override delta is the difference between rebalance priorities of two rebalance operations that compete for the same node. Operations "higher" on the tree get executed first, which is reflected in the rebalance priorities assigned to the different OperationMarks. We store the difference between the two override priorities of competing operations in the OVERRIDEN StatusDetails for each override that occurs. Lets describe this with an example.

Lets say operations A and B are competing. A is trying to claim node N with OperationMark INSERT_PARENT, and B has already claimed it with INSERT_GRANDPARENT. Now, A wins out, and overrides B by applying a CompareAndSet operation to the StatusDetails reference, with OVERRIDEN status. However, we also include the difference between the two rebalance priorities in the StatusDetails. We subtract the priority of the loser from the priority of the winner, 2 (INSERT_PARENT) - 4 (INSERT_GRANDPARENT) = -2.

Now lets say operation C comes along, and attempts to claim some node X of B, that is not the same as N. C wants to claim X as INSERT_UNCLE, and B's OperationInfo has the OperationMark of INSERT_PARENT for X. C detects the override and follows the override chain, adding up the **override delta** all the way. When it gets to A's OperationInfo, it already knows that the delta so far is -2. It will then calculate its own delta to node X to understand its own position relative to B, 2 (INSERT_UNCLE) - 2 (INSERT_PARENT) = 0. It will then subtract the **override delta** computed during the traversal of the override chain, 0 - (-2) = 2. Because the number is a positive one, we know that C is lower in the tree relative to A, and therefore should lose the competition for X.

Iteratively Locking Nodes

After the first node has been claimed by an operation, each subsequent node it needs to claim has to go through some additional logic to make sure that the operation is still in a viable state. Algorithm 12 describes this logic. We make sure to check that the StatusDetails we are expecting are still valid. If so, we go through the locking algorithm.

If we find our StatusDetails LOCKED, then someone has been helping us and has acquired all the nodes we need and locked the operation. In this case, we do not attempt the locking logic again, and simply accept this node as LOCKED and acquired.

If we find our StatusDetails OVERRIDEN, then we must help our overrider and discard the operation.

If we find our StatusDetails DISCARDED, and if the details are marked with *doReviveOperation*, then we revive it.

If we find ourselves FINALIZED, then someone has completed the operation entirely for us. If so, we return a success value with a shortcut, indicating that no additional nodes should be claimed by the operation and that we've already been completed.

Locking the Node

Locking the node is fairly trivial if not considering the intricacies of overriding. Algorithm 13 shows the details of locking a node. We first assume that the node is entirely unclaimed and has no operation attached. We attempt a CAS operation to claim the node by attaching our own OperationInfo to it. If we failed the CAS operation, then some other operation claimed the node first.

We obtain the OperationInfo and the StatusDetails of the contender. If the details are DISCARDED and are not in need of revival, then we treat the node as if it was unclaimed. We again attempt a CAS operation on the node, this time expecting the contender's OperationInfo to be attached. If successful, we simply return success. If not, then someone else has claimed the node first and we restart from the beginning. If the operation needs to be revived, we execute *reviveOperation* on it before attempting to claim it.

If the status wrapper of the contender's OperationInfo is the same object as our operation's StatusWrapper, then we know that the contender is someone helping us complete our operation. In this case, we simply report that we've been successful in locking the node, because our helper has.

Otherwise, if attemptOverride has been set to true, we run the override algorithm against the node. The attemptOverride flag is only ever false on the attempt to claim the first node of a simple

```
public void iterativeLockNode (boolean isRebalance)
{
    output: None.
    globalScope.lockSuccess ← false;
    globalScope.shortcut ← false;
    StatusDetails actualDetails ← (globalScope.info→details);
    if (actualDetails == globalScope.expectedDetails)
    {
        lockNode(isRebalance);
    }
    else if ((actualDetails→status) == LOCKED)
    {
        globalScope.shortcut ← true;
    }
    else if ((actualDetails→status) == OVERRIDEN)
    {
        help(actualDetails→actionNode);
    }
    else if ((actualDetails→status) == DISCARDED)
    {
        if (actualDetails→doReviveOperation)
        {
            reviveOperation(globalScope.info, actualDetails→actionNode);
        }
    }
    else if ((actualDetails→status) == FINALIZED)
    {
        globalScope.lockSuccess ← true;
        globalScope.shortcut ← true;
    }
    if (not globalScope.lockSuccess)
    {
        // Re-obtain the details in case we helped another operation, and then
           marked ourselves OVERRIDEN
        StatusDetails actualDetails ← (globalScope.info→details);
        if (actualDetails→status in (LOCKING, LOCKING_REBALANCE, OVERRIDEN))
        {
            discardOperation(globalScope.info, actualDetails);
        }
    }
}
```

Algorithm 12: Iteratively claiming the node.

LOCKING operation. The various parameters to the attemptOverride method will be explained in the next section.

Line 1 of the algorithm describes a method for fixing any possible leaf edges of a tree swapped in by another operation. We delve more into this concept in the **Fixing Up Pointers** section.

Overriding

The most involved part of the Node Lock Protocol is overriding. Overriding allows operations to compete for nodes, help one another, and loosely enforces an ordering of operations. There are many cases to consider.

Algorithm 14 shows the first case of an override operation. This case deals with what to do when NLP is handling an operation and encounters an OVERRIDEN operation in the override chain of the node it is attempting to claim. The operation can only be overriden by a rebalance operation, or if the operation itself is not a rebalance operation, as specified by the condition on line 9 of the algorithm.

Lines 12-15 deal with the situation when someone has already overriden the node on our behalf. In this case we simply return successfully as we're already the overriders of the node.

Lines 16-19 show a recursive invocation of the override algorithm. The algorithm traverses the override chain as explained earlier, each time running through the same protocol. Special care is paid to ensure that the overrideDelta is properly computed each step of the way.

It is possible that the overrider at the end of the override chain is DISCARDED. In this case, we do not override that particular operation. We return a value of **true** for the **overriderDiscarded** property and actually attempt to override the operation one step higher in the override chain. Lines 23-31 handle this case.

In the end of every override chain, if we're not successful in overriding the root operation, we invoke the **help** protocol to make sure that operation finishes. When traversing the override chain back up from helping the root overrider, we have to make sure that we discard all OVERRIDEN operations.

Algorith 16 outlines the logic that processes the competition between two LOCKING_REBALANCE operations. The operations are compared using Algorithm 15. This logic takes into effect the relative positions of the operations in questions first and foremost, through the use of the **overrideDelta** concept. If equal, it will then consider their actual operation priorities.

Once the result of the competition is known, we process it accordingly. A value of 0 indicates that the LOCKING_REBALANCE operation is actually equivalent to the one currently attempting to override the node. In this case we simply return successfully.

```
public void lockNode (boolean isRebalance)
{
    output: None.
    globalScope.lockNode.fixPrevUpNodes();
    while (true)
    {
        if ((globalScope.lockNode→operation).cas(null, globalScope.info))
        {
            globalScope.lockSuccess ← true;
        }
        else
        {
            OperationInfo contender ← (globalScope.lockNode→operation);
            StatusDetails contenderDetails ← (contender→details);
            if (contenderDetails→status == DISCARDED)
            {
                if (details→doReviveOperation)
                {
                    reviveOperation(contender, contenderDetails→actionNode);
                }
                if (globalScope.lockNode→operation).cas(contenderDetails, globalScope.info))
                {
                    globalScope.lockSuccess ← true;
                }
                else
                {
                    continue;
                }
            }
            else if (contenderDetails == globalScope.expectedDetails)
            {
                globalScope.lockSuccess ← true;
            }
            else if (globalScope.attemptOverride)
            {
                globalScope.lockSuccess ← (attemptOverride(globalScope.lockNode, null,
                isRebalance, false, 0)→success);
            }
            else
            {
                globalScope.lockSuccess ← false;
            }
        }
    }
}
```

Algorithm 13: Locking a node.

```
public OverrideResult attemptOverride (Node n, OperationInfo contenderInfo, boolean
isRebalance, boolean nestedOverride, int overrideDelta)
{
    output: The result of the override. Success field reports whether the override was
            successful. The overriderDiscarded field reports whether the most recent
            overrider was found to be discarded.
    StatusDetails overrideDetails ← createNewOverrideDetails(globalScope.info→actionNode,
    globalScope.info→priority);
    while (true)
    {
        if (contenderInfo == null)
        {
            contenderInfo ← (n→operation);
        }
        StatusDetails contenderDetails ← (contenderInfo→details);
        if (contenderDetails→status == OVERRIDEN & (isRebalance |
        contenderInfo→rebalanceNode == null))
        {
            OperationInfo overriderInfo ← ((contenderDetails→actionNode)→operation);
            if (overriderInfo→statusWrapper == globalScope.info→statusWrapper)
            {
                result ← new OverrideResult(success ← true, overriderDiscarded ← false);
            }
            else
            {
                result → attemptOverride(n, overriderInfo, isRebalance, true,
                (overrideDelta+(contenderDetails→overrideDelta)));
            }
            OverrideResult result ← new OverrideResult(success ← false, overriderDiscarded
            ← false);
            if (result→success)
            {
                if (result→overriderDiscarded & not
                (contenderInfo→details).cas(contenderDetails, overrideDetails))
                {
                    continue;
                }
                (result→success) ← true;
            }
            else if (contenderInfo→rebalanceNode == null)
            {
                (contenderInfo→details).cas(contenderDetails, newDiscardDetails());
            }
            return result;
        }
    }
}
```

Algorithm 14: Overriding a node (Part 1).

```
public RebalanceCompareResult compareRebalanceOperations (OperationInfo
myOp, OperationInfo otherOp, Node n, int overrideDelta)
{
    output: The result of comparing rebalance operations. Compare value reports the actual
            result. If negative, then myOp has lower priority, and vice versa. Override delta
            reports the delta between the position of the node's initial operation and myOp.
    RebalanceCompareResult result ← new RebalanceCompareResult(compareVal←0,
    overrideDelta←0);
    if (not myOp→priority == otherOp→priority)
    {
        OperationInfo nodeInfo ← (n→operation);
        localDelta ← ((myOp→mark)→rebalancePriority) -
        ((otherOp→mark)→rebalancePriority) - overrideDelta;
        if (localDelta $\neq$ 0)
        {
            (result→overrideDelta) ← localDelta;
            (result→compareVal) ← localDelta;
        }
        else
        {
            (result→compareVal) ← (myOp→priority compareTo otherOp→priority);
        }
    }
}
```

Algorithm 15: Comparing rebalance operations.

If we encounter a value less than 0, then we know that the current operation should override the operation already on the node. In this case we attempt to use a CompareAndSet operation to actually override the details of the existing operation. If we're not successful, then some other operation succeeded in changing the StatusDetails of the existing operation first, and we should try again.

Finally, if we encounter a value greater than 0, then our operation has lower priority than the one already on the node. In this case we simply help the existing operation complete. Algorithm 21 describes the way helping another operation is accomplished.

Algorithm 17 outlines the logic for overriding operations in a LOCKING state. We attempt the CompareAndSet operation on the StatusDetails if we are a rebalance operation, and therefore win over a LOCKING operation by default, or if our priority is higher. If our priority is the same as the LOCKING operation's, then someone is helping us and we return successfully. Otherwise, we've lost the competition and help the operation complete.

Algorithm 18 describes what is done when we are a LOCKING operation and we encounter a LOCKED or LOCKING_REBALANCE operation. In this case we simply help that operation complete and return false.

Algorithm 19 describes the eventuality of finding the operation DISCARDED. First, if the *doOverrideOperation* flag is set, we invoke *reviveOperation* to ensure the operation is properly revived. If the node we are attempting to override is actually the action node of the discarded operation that we've revived, we must reconsider this node. In this case we obtain the OperationInfo of the node again, and continue from the beginning of the loop.

If we are in a nested override, meaning that we're more than one level deep in the override chain, we simply return true as the root overrider has been DISCARDED. However, we make sure to return with a value of true for **overriderDiscarded** property so that we will know to override the operation that's one level higher in the override chain, as explained in Algorithm 14. If we're not in a nested override, then we actually attempt to claim the node by applying a CompareAndSet to the OperationInfo reference and **not** to the StatusDetails. If we fail, we have to make sure to reset the contenderInfo reference to the whatever it became, since it must have changed for a CompareAndSet to have failed. Then we can examine that new OperationInfo and act accordingly.

Algorithm 20 describes what happens when an operation is already FINALIZED. In this case, we not only cannot override the operation, but we're also on a node that is no longer part of the tree. We simply return false and try again.

Algorithm 21 shows the helping procedure.

If we find that the operation on the node is OVERRIDEN, we help its overrider. We then discard

```
public OverrideResult attemptOverride (Node n, OperationInfo contenderInfo, boolean
isRebalance, boolean nestedOverride, int overrideDelta)
{
    output: The result of the override. Success field reports whether the override was
            successful. The overriderDiscarded field reports whether the most recent
            overrider was found to be discarded.
    ...;
    while (true)
    {
        ...;
        if (contenderDetails→status == OVERRIDEN & (isRebalance |
        contenderInfo→rebalanceNode == null))
        {
            ...;
        }
        else if (isRebalance & contenderDetails→status == LOCKING_REBALANCE)
        {
            RebalanceCompareResult result ← compareRebalanceOperations(globalScope.info,
            contenderInfo, n, overrideDelta);
            if (result→compareVal == 0)
            {
                return new OverrideResult(success ← true, overriderDiscarded ← false);
            }
            else if (result→compareVal < 0)
            {
                (overrideDetails→overrideDelta) ← (result→overrideDelta);
                if ((contenderInfo→details).cas(contenderDetails, overrideDetails))
                {
                    return new OverrideResult(success ← true, overriderDiscarded ← false);
                }
                else
                {
                    continue;
                }
            }
            else
            {
                help(contenderDetails→actionNode);
            }
        }
    }
}
```

Algorithm 16: Overriding a node (Part 2).

```
public OverrideResult attemptOverride (Node n, OperationInfo contenderInfo, boolean
isRebalance, boolean nestedOverride, int overrideDelta)
{
    output: The result of the override. Success field reports whether the override was
            successful. The overriderDiscarded field reports whether the most recent
            overrider was found to be discarded.
    ...;
    while (true)
    {
        ...;
        else if (isRebalance & contenderDetails→status == LOCKING_REBALANCE)
        {
            ...;
        }
        else if (contenderDetails→status == LOCKING)
        {
            if (isRebalance | contenderInfo→priority > globalScope.info→priority)
            {
                if ((contenderInfo→details).cas(contenderDetails, overrideDetails))
                {
                    return new OverrideResult(success ← true, overriderDiscarded ← false);
                }
                else
                {
                    continue;
                }
            }
            else if (contenderInfo→priority == globalScope.info→priority)
            {
                return new OverrideResult(success ← true, overriderDiscarded ← false);
            }
            else
            {
                help(contenderDetails→actionNode);
                return new OverrideResult(success ← false, overriderDiscarded ← false);
            }
        }
    }
}
```

Algorithm 17: Overriding a node (Part 3).

```
public OverrideResult attemptOverride (Node n, OperationInfo contenderInfo, boolean
isRebalance, boolean nestedOverride, int overrideDelta)
{
    output: The result of the override. Success field reports whether the override was
            successful. The overriderDiscarded field reports whether the most recent
            overrider was found to be discarded.
    ...;
    while (true)
    {
        ...;
        else if (contenderDetails→status == LOCKING)
        {
            ...;
        }
        else if (contenderDetails→status == LOCKED | contenderDetails→status ==
        LOCKING_REBALANCE)
        {
            help(contenderDetails→actionNode);
            return new OverrideResult(success ← false, overriderDiscarded ← false);
        }
    }
}
```

Algorithm 18: Overriding a node (Part 4).

the operation, and if it was a rebalance operation and the node we're interested in is the same as the rebalance node of the discarded operation then we know that the operation has been revived, and its action node should now have a new valid OperationInfo on it. In this case we recursively call *help()* on the revived action node.

If it is in LOCKING, LOCKING_REBALANCE, and LOCKED states, we simply start up a new AddNodeLockTree with the node's OperationInfo and StatusDetails, and invoke it's algorithm to complete the operation.

If the operation is DISCARDED, and needs to be revived, we revive the operation. If the node we are attempting to help is the action node of the revived operation, we help that node complete.

There's nothing we can do for FINALIZED operations, so we simply return.

3.2.4 Transformation Phase

The transformation phase represents a very standard algorithm for performing an operation on a LOCKED subtree. Here we won't go into exact specifics of the algorithm, but will instead show the different possible cases that can arise, and note some particular data that needs to be added to the tree in order for other threads to be able to help [Bay72].

When we've assembled and locked the subtree, we have to consider several different scenarios.

```
public OverrideResult attemptOverride (Node n, OperationInfo contenderInfo, boolean
isRebalance, boolean nestedOverride, int overrideDelta)
{
    output: The result of the override. Success field reports whether the override was
            successful. The overriderDiscarded field reports whether the most recent
            overrider was found to be discarded.
    ...;
    while (true)
    {
        ...;
        else if (contenderDetails→status == LOCKED | contenderDetails→status ==
        LOCKING_REBALANCE)
        {
            ...;
        }
        else if (contenderDetails→status == DISCARDED)
        {
            if (contenderDetails→doReviveOperation)
            {
                reviveOperation(contenderInfo, contenderDetails→actionNode);
                if (globalScope.lockNode == (contenderDetails→actionNode))
                {
                    contenderInfo ← (n→operation);
                    continue;
                }
            }
            if (nestedOverride)
            {
                return new OverrideResult(success ← true, overriderDiscarded ← true);
            }
            else
            {
                if ((n→operation).cas(contenderInfo, globalScope.info))
                {
                    return new OverrideResult(success ← true, overriderDiscarded ← false);
                }
                else
                {
                    contenderInfo ← n→operation;
                    continue;
                }
            }
        }
    }
}
```

Algorithm 19: Overriding a node (Part 5).

```
public OverrideResult attemptOverride (Node n, OperationInfo contenderInfo, boolean
isRebalance, boolean nestedOverride, int overrideDelta)
{
    output: The result of the override. Success field reports whether the override was
            successful. The overriderDiscarded field reports whether the most recent
            overrider was found to be discarded.
    ...;
    while (true)
    {
        ...;
        else if (contenderDetails→status == DISCARDED)
        {
            ...;
        }
        else if (contenderDetails→status == FINALIZED)
        {
            return new OverrideResult(success ← false, overriderDiscarded ← false);
        }
    }
}
```

Algorithm 20: Overriding a node (Part 6).

There is an equivalent set of scenarios for the deletion of a node, but we will not delve into these because delete has not been implemented in the presented algorithm. For all of the below scenarios, the transformation is mirrored for the left and right sides of the tree. Thus, it does not matter what direction the parent node is relative to the grandparent, or what direction the rebalance node is relative to the parent. We will mirror the transformation.

If the parent node is BLACK, the algorithm does not attempt to acquire any other nodes. The reason for this is that we always insert new nodes already colored RED, and the rebalance node is always RED. Having a BLACK parent means that the RED child will not violate any of the red-black tree properties. In this case, we simply create a new subtree consisting of just the parent node and the child node (be it the rebalance node or a brand new node to insert), and swap it into the overall tree. Figure 3.7 graphically illustrates this process.

If the parent is RED, then the grand-parent must be BLACK. Lemma 3.2.1 guarantees that the tree we lock during any operation will be one that adheres to the red-black tree properties. This means that RED nodes cannot have RED children, which guarantees that the grandparent is BLACK.

There are now three options for the uncle node. Either it does not exist (A), it exists and is RED (B), or it exists and is BLACK (C). For the purposes of the algorithm, cases A and B are treated as the same case. When encountering this case, we can repaint the parent and uncle, if it exists,

```
public void help (Node n)
{
    output: None.
    OperationInfo info ← (n→operation);
    StatusDetails details ← (info→details);
    if ((details→status) == OVERRIDEN)
    {
        help(details→actionNode);
        discardOperation(info, details);
        if ((info→rebalanceNode) == n)
        {
            help(info→rebalanceNode);
        }
    }
    else if ((details→status) == DISCARDED & details→doReviveOperation)
    {
        reviveOperation(info, details→actionNode);
        if ((details→actionNode) == n)
        {
            help(n);
        }
    }
    else if ((details→status) in (LOCKING, LOCKING_REBALANCE, LOCKED))
    {
        newLockingPhase(info, details).assembleAndContinue();
    }
    // Nothing we can do if FINALIZED.
}
```

Algorithm 21: Helping.

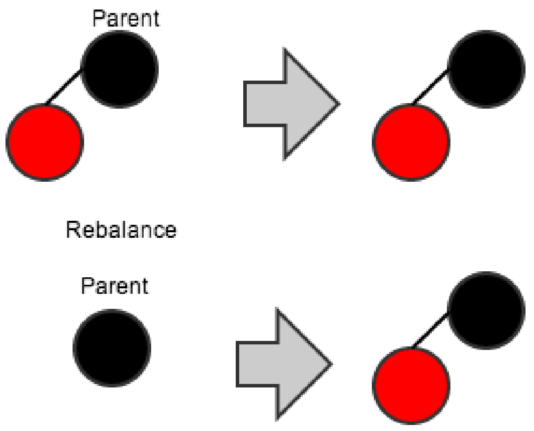


Figure 3.7: Transforming the subtree - case 1.

BLACK, and the grandparent RED. The paths going through the parent and grand parent node still contain the same number of BLACK nodes, since instead of going through a BLACK parent or a BLACK uncle, they both go through a BLACK grandparent. This situation, however, leaves the possibility that red-black tree property of "a RED node cannot have a RED child" is broken because the parent of our grandparent node may also be RED. Thus we must mark our grandparent node with an INSERT_REBALANCE operation, starting a new LOCKING_REBALANCE operation with the same priority as the original insert. This OperationInfo is already attached to the node when we swap in the tree, so there is no competition for the first node. Figure 3.8 graphically illustrates this scenario. The dotted line from grandparent to uncle indicates that the uncle node is optional.

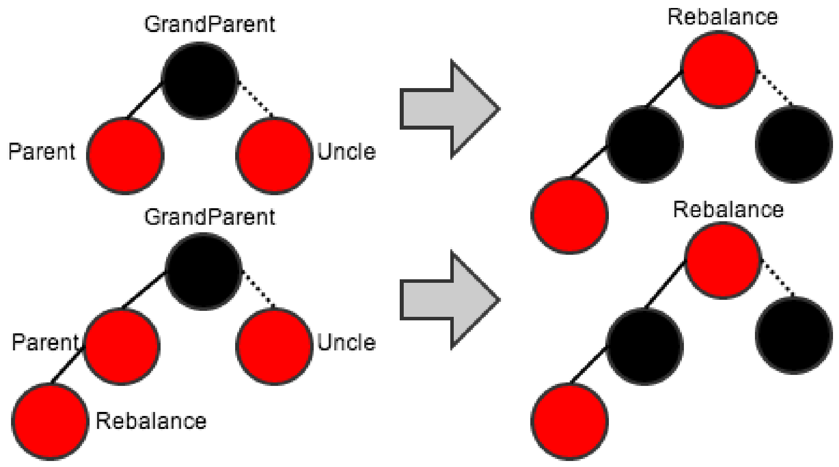


Figure 3.8: Transforming the subtree - case 2.

The last case deals with what to do when the uncle node is BLACK and the parent node is RED. This case is slightly more complex. Lets name the nodes for easier description. Let the insert or rebalance node be named N, the parent node P, the grandparent GP, and the uncle U.

If P is to the left of GP and N is to the right of P, or vice versa, if P is to the right of GP and N is to the left of P, we must perform a rotation. We will rotate N and P, making the N be the parent of P. We can now rename P to N and N to P.

Now, we're in a situation where P is on the same side of GP as N is of P. We perform a rotation on P and GP, making P the root of the subtree, and repainting P BLACK and GP RED. We now have a BLACK root, nowhere in the subtree do we have a RED parent with a RED child, and the same number of BLACK nodes is traversed for all of the paths through the tree. Paths stemming from P, N and GP still have only one BLACK node P in their path. The paths going through U still have two BLACK nodes in the path, but now these are P and U instead of GP and U. Figure 3.9 graphically illustrates this scenario.

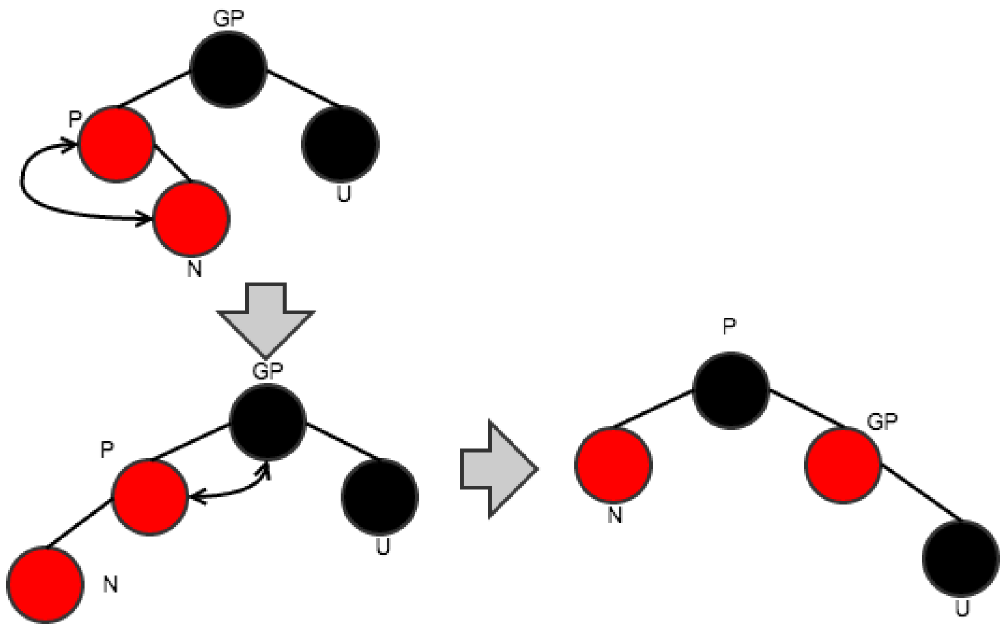


Figure 3.9: Transforming the subtree - case 3.

Fixing Up Pointers

It is possible that the thread that succeeds in swapping in the subtree by applying a CAS operation on the down pointer of the root edge is not the operation that will fix all the bottom edges to point up to the new subtree. The operation that succeeds in swapping in the new subtree attaches certain data to the root node and the leaf nodes of the new subtree. This information helps other threads finish up the operation and move on.

The root node of the swapped in subtree will contain references to all the leaf nodes that need their up pointers fixed. Each of the leaf nodes, in turn, will contain two AtomicReference objects. These are named prevLeftUpNode and prevRightUpNode. These atomic references will point to the old tree's nodes that the left and right edges were pointing up to before the tree was swapped.

When a thread fails the CAS operation to swap in its own subtree, it will look at the new **down** pointer of the root parent edge to find the new root node. It will then iterate through that node's list of leaf nodes. For each of the leaf nodes' edges it will attempt a CAS operation, expecting the node referenced in the prevLeftUpNode or prevRightUpNode and swapping in a reference to the replacement node. This operation can only occur once, as the swapped out node will never be seen in the tree again. Also, after performing the CAS operation, it then sets the prevLeftUpNode and prevRightUpNode references to null, ensuring that no other thread will attempt to fix the up pointers on this tree.

Once all up pointers have been fixed, a thread that failed to swap in the tree can safely mark the OperationInfo FINALIZED. Algorithm 22 shows the iteration that a swap loser would go through in order to make sure all nodes' up pointers have been fixed. Algorithm 23 shows the steps taken for each individual node to ensure that it's left and right edges are properly pointing up at it.

```
public void fixUpPointers (Node n)
{
    output: None.
    List nodesToFix ← (n→nodesToFix);
    if (nodesToFix ≠ null)
    {
        foreach (Node fixNode in nodesToFix)do
        {
            fixNode.fixPrevUpNodes();
        }
        end
        (n→nodesToFix) ← null;
    }
}
```

Algorithm 22: Fixing up pointers for subtree.

Figure 3.10 presents a graphical analysis of this situation. In this figure, the old subtree consisting of A, B and C is still a part of the tree. An operation is getting ready to swap in a new subtree - A', B', and C'. E(XR) represents the right edge of node X and E(XL) the left. The dotted lines represent the AtomicReference up and down pointers of the edges. At this stage, all up pointers are

```
public void fixPrevUpNodes ()
{
    output: None.
    Node prevNode ← prevLeftNode;
    if (prevNode ≠ null)
    {
        (leftEdge→up).cas(prevNode, thisNode);
        prevLeftNode ← null;
    }
    prevNode ← prevRightNode;
    if (prevNode ≠ null)
    {
        (rightEdge→up).cas(prevNode, thisNode);
        prevRightNode ← null;
    }
}
```

Algorithm 23: Fixing up pointers for node.

correctly referencing the old subtree.

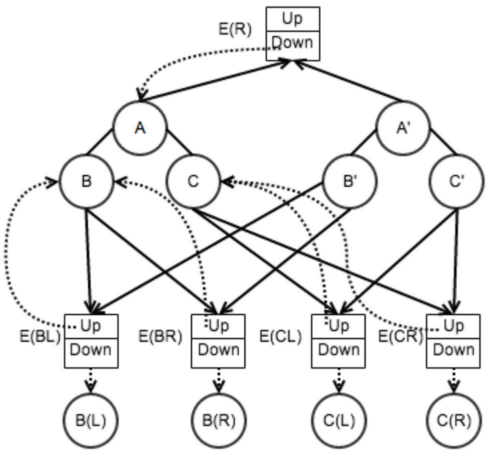


Figure 3.10: Fixing the edge pointers (1).

Figure 3.11 shows the situation right after the tree has been swapped in. The root edge down pointer is correctly pointing at A'. However, none of the leaf up edges are pointing at the new subtree. They are all still referencing the leaf nodes of the old subtree. The data structures are also shown. The nodesToFix list is attached to the new root of the subtree - A' - and is referencing the two new leaf nodes, B' and C'. Both of the leaf nodes contain the knowledge that their down edges are currently misconfigured and are pointing at the leaves of the old subtree. Both edges belonging to B' are currently pointing up at B, and same with C' and C.

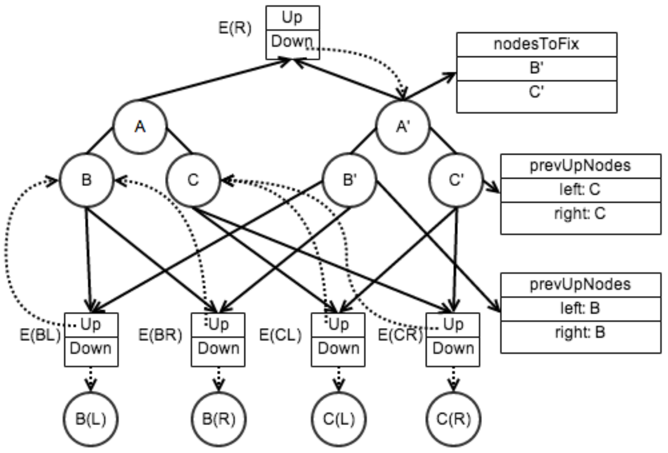


Figure 3.11: Fixing the edge pointers (2).

Figure 3.12 shows the situation after the execution of Algorithm 22. All of the up pointers have been fixed, and the references have been all set to null to prevent any unnecessary work for other operations that are now traversing this new subtree.

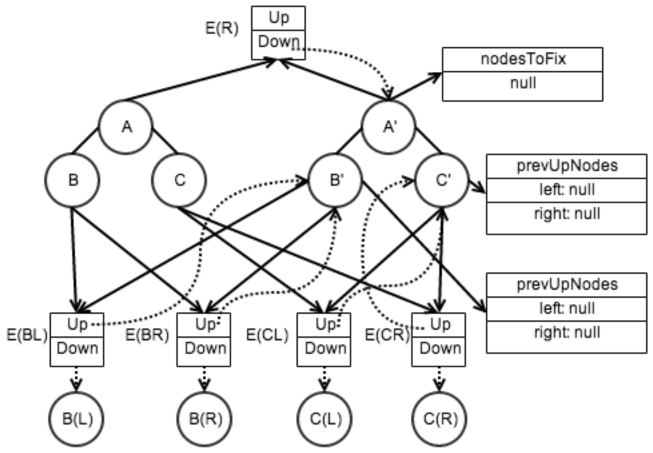


Figure 3.12: Fixing the edge pointers (3).

Chapter 4

Experimental Evaluation

The algorithm has been implemented using the Java programming language, in approximately 2000 lines of code. Some experimentation has been done to determine its effectiveness with high contention and a large number of nodes added.

The algorithm has been evaluated with 1, 2, 4, 10, and 20 threads, each with 5000, 10000, 20000, 40000 total unique values added over 20 iterations. The same set of randomly generated values was used for testing with all the different thread counts. In the case of multiple threads, the values to add were divided evenly amongst them. All threads were started at the same time, and the end time was recorded after the completion of the last thread's *add()* call. The averages, min, max and standard deviations of total completion times over 20 iterations have been recorded in the tables below. The values have been rounded to the nearest integer.

The tests were performed on a MacBook Pro with a 2.9 GHz Intel Core i7 CPU, 8 GB DDR3 RAM and a Solid State Hard Drive.

The tables clearly show that the number of threads put to the task impacts the runtime of the work greately. However, we see the greatest fall in runtime between running the algorithm completely sequentially and introducing just one other thread. Although the numbers do trend down as more threads are added, the gains in performance are significantly smaller. In some cases, using 20 threads was less efficient than just 10.

The most likely explanation of this phenomenon is that when a large number of threads is involved, contention becomes so great that helping other processes complete takes a significant portion of the execution time. Keeping track of the time spent after invoking the *help()* method may shed more light on the exact specifics of the algorithm execution.

Table 4.1: Experimental Average Run Times in Milliseconds

Total Values Added	Number of Threads				
	1	2	4	10	20
5,000	256	148	114	113	114
10,000	458	276	220	219	219
20,000	919	552	437	433	430
40,000	1,847	1,167	926	893	879

Table 4.2: Experimental Run Time Standard Deviations in Milliseconds

Total Values Added	Number of Threads				
	1	2	4	10	20
5,000	109	45	30	19	34
10,000	39	16	25	18	11
20,000	31	16	31	24	19
40,000	81	118	95	113	71

Table 4.3: Experimental Minimum Run Times in Milliseconds

Total Values Added	Number of Threads				
	1	2	4	10	20
5,000	215	125	98	100	99
10,000	434	262	208	206	208
20,000	885	530	408	415	414
40,000	1,780	1,084	835	822	826

Table 4.4: Experimental Maximum Run Times in Milliseconds

Total Values Added	Number of Threads				
	1	2	4	10	20
5,000	711	330	235	183	252
10,000	595	324	324	284	241
20,000	996	584	516	492	474
40,000	2,148	1,550	1,136	1,314	1,073

Chapter 5

Future Work

5.1 Delete Operation

The most obvious part of the algorithm that still needs to be implemented is the **delete** operation. Most of the algorithms described for the insert operation should apply well to the way delete is handled. However, there are some important distinctions.

5.1.1 Dealing with an Unconnected Operation Tree

A delete operation on a binary tree is usually accomplished by finding the node to be deleted, and then finding its least value right descendant or its greatest value left descendant in the tree. Once found, the value of the descendant is assigned to the original node to be deleted, and the descendant node is then removed from the tree. This is more involved in red-black trees, as the subtree formed around the descendant has to go through its own transformation and possible rebalances in order to maintain the red-black tree properties.

This approach to deleting poses a problem. We now have the original delete node to swap with a new value, and the subtree formed around the descendent to swap with a new subtree. So this is no longer just a single swap, but two. The algorithm outlined should work properly when used in sequence, for both swaps, once the operation has been LOCKED. However, this means that different helping threads may succeed in swapping either of the tree segments.

5.1.2 Searching while Delete is in Progress

Assume that a search operation is looking for the descendant node, traverses **past** the delete node in the tree, and gets interrupted. A concurrent delete operation completes and swaps the descendant node with the delete node. The search operation resumes, travels to the bottom of the tree and is unable to find the node because it was swapped out to a place higher in the tree. Thus, a search operation operating in this fashion can be unable to find a node even though it has not been deleted from the tree. This would completely break the algorithm.

Here, we propose a solution to this problem. We introduce a new object, called a SearchInfo. It maintains two fields. Value is the key to the node that the search operation is looking for. Swapped is the reference to the found node if swapped, initialized to null.

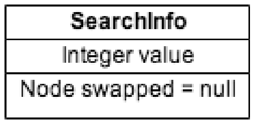


Figure 5.1: The SearchInfo object.

In lieu of a better wait-free data structure, this solution requires the tree to know about how may threads are going to be accessing the tree concurrently. The tree will maintain a list of AtomicReferences to SearchInfos for each thread. These will all be initialized to null.

When a search operation is first initialized, it will create and set a SearchInfo object for its thread ID in the global list, announcing that it is searching for a particular value.

When a delete operation successfully replaces the delete node with the swap, essentially making the swap node appear twice in the tree, but **before** it attempts to finalize the transformation on the subtree formed around the swap node that would physically delete it from the tree, it will iterate through the global list of SearchInfos, and when it encounters a SearchInfo with the value equivalent to that of the swap node, it will set **swapped** reference to the newly swapped node.

Finally, if a search operation is unable to find a node it seeks, it will check its SearchInfo to see if the swapped value was populated by a delete operation. If so, then it will return that populated node reference.

Any search operations that were above the delete node when the swap occurred will find the swap node in place of the delete node and return successfully. Any operations that managed to get past the delete node, and get interrupted will have their SearchInfo swapped values populated by the delete operation.

5.1.3 Rebalance Competition

The delete operation subtrees look very different from those created for insert operations. A delete subtree consists of a parent, sibling, and the two children of the sibling. When considering this subtree, one can see that Lemma 3.2.1 no longer applies. It is indeed possible that competing rebalance operations will be able to override the actual rebalance node of another operation.

It is vital to develop strategies for dealing with a rebalance sibling or niece as the delete operation is developed.

5.2 Priority Counter Rollover

Each of the operations presented in the algorithm obtain a steadily increasing priority in order to be able to compete with one another. In the implementation, this is achieved using an AtomicInteger class with the **incrementAndGet()** call. However, during a long enough run of the algorithm, this may cause the value to rollover, vending negative priorities. This would cause the operations that were active during the rollover to have a possibly lower priority than the new operations coming in.

Instead of a simple AtomicInteger, we can utilize a rollover-free atomic timestamp scheme, such as the one proposed in [DS97].

5.3 Rigorous proof

This paper does not provide a rigorous proof of the correctness of the algorithm. The algorithm has run without any problems for thousands of test runs involving dozens of threads and millions of values added. However, this does not guarantee that there is not a case that the algorithm cannot deal with.

5.4 Comparison to Other Algorithms

We've done some testing on the execution times of the algorithm using different numbers of threads and different sets of data. A comparison between our algorithm and locking algorithms, as well as other wait-free algorithms that have been developed will need to be performed in order to prove that the algorithm can compete or even surpass other implementations' times.

5.5 Starvation-freedom

The presented algorithm is not starvation-free. It is possible that a very unlucky thread could continuously fail to acquire all the nodes it needs and complete its operation. In practical applications, this should not occur. In a theoretical sense, this is a big flaw that needs to be addressed.

Some ways to address this would involve allowing threads to compete for the first node of their operations. One other issue to address would be the competition of LOCKING_REBALANCE threads. A LOCKING_REBALANCE thread could become stuck when its neighbors keep managing to move up the tree first. Some sort of ordering to the LOCKING_REBALANCE operations can be enforced at the node level, informing threads that a particular operation has already been waiting for its turn and instead helped another, and it should now go first.

Chapter 6

Conclusion

We have presented a wait-free algorithm for inserting nodes into a red-black bi-directional tree. The algorithm provides many improvements over the traditional "window"-based approaches to tree algorithms. It also comes with quite a few cons that need to be considered. The pros and cons are itemized as follows.

Pros:

- The algorithm allows transformative operations to traverse the tree in both directions. This freedom allows operations to compete with one another only in small, localized spaces of the tree. This localization allows threads operating on completely disparate parts of the tree to run entirely concurrently, without having to be synchronized to run in any particular order.
- Read-only operations are capable of traversing the tree without performing any writes on the shared memory, maintaining a set of executions that are quiescently consistent and linearizable.
- Operations compete with one another based on their priorities, enforcing a loose ordering of threads but not guaranteeing that the operation with the lower priority will win out.
- The Node Lock Protocol scheme can be applied to any algorithm operating on a bi-directional tree. It can be used to acquire and keep an arbitrary collection of nodes for a particular operation.

Cons:

- The algorithm is not starvation free. This means that in a theoretical sense a particular operation could fail to finish for an indefinite period of time.

- The algorithm is not complete. Without a delete operation implemented, it is of relatively limited use. However, the existing framework can be applied to finish the delete operation, as the insert and delete behave very similarly.

- During the execution of a transformative operation, such as insert or delete, the tree may exist in a state inconsistent with red-black tree properties. This is fixed quickly and operations that need to operate in the inconsistent part of the tree may help to rebalance and get the tree back into a stable configuration. Nonetheless, it is possible that a concurrent search operation may traverse a longer distance while the tree is being rebalanced.

Overall, we present a novel approach to wait-free operations on trees in general, and red-black trees in particular. By considering a bi-directional approach, we open new avenues to implementing various transformations on tree graphs.

Bibliography

[Bay72] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[DS97] Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM J. Comput.*, 26(2):418–455, April 1997.

[EFRvB10] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0:8–21, 1978.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[JHKG06] H. Cameron J. H. Kim and P. Graham. Lock-free red-black trees using cas. *Concurrency and Computation: Practice and Experience*, pages 1–40, 2006.

[NSM13] Aravind Natarajan, LeeH. Savoie, and Neeraj Mittal. Concurrent wait-free red black trees. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 8255 of *Lecture Notes in Computer Science*, pages 45–60. Springer International Publishing, 2013.

[Ora14] Oracle. java.util.concurrent.atomic package documentation. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html`, May 2014.

[Tar85] R. E. Tarjan. Efficient top-down updating of red-black trees. *Technical Reports*, 1985.

Vita

Graduate College
University of Nevada, Las Vegas

Vitaliy Kubushyn

Degrees:

Bachelor of Science in Computer Science 2009
University of Nevada Las Vegas

Thesis Title: Concurrent Localized Wait-Free Operations on a Red-Black Tree

Thesis Examination Committee:

Chairperson, Dr. Ajoy K. Datta, Ph.D.

Committee Member, Dr. John Minor, Ph.D.

Committee Member, Dr. Yoohwan Kim, Ph.D.

Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph.D.