The Amortized Analysis of a Non-blocking Chromatic Tree

Jeremy Ko
Department of Computer Science, University of Toronto, Canada
jerko@cs.toronto.edu

Abstract

A non-blocking chromatic tree is a type of balanced binary search tree where multiple processes can concurrently perform search and update operations. We prove that a certain implementation has amortized cost $O(\dot{c}+\log n)$ for each operation, where $\dot{c}$ is the maximum number of concurrent operations at any point during the execution and n is the maximum number of keys in the tree during the operation. This amortized analysis presents new challenges compared to existing analyses of other non-blocking data structures.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis, Theory of computation → Distributed algorithms

Keywords and phrases amortized analysis, non-blocking, lock-free, balanced binary search trees

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.8

Related Version A full version is available at `https://arxiv.org/abs/1811.06383`.

Funding Funding was provided by the Natural Sciences and Engineering Research Council of Canada.

Acknowledgements I want to thank my supervisor, Faith Ellen, whose guidance and feedback was very helpful in writing this paper. I would also like to thank Eric Ruppert and the anonymous reviewers.

1 Introduction

A *concurrent data structure* is one that can be concurrently updated by multiple processes. A *wait-free* implementation of a concurrent data structure guarantees that every operation completes within a finite number of steps by the process that invoked the operation. A *non-blocking* implementation of a concurrent data structure guarantees that whenever there are active operations, one operation will eventually complete in a finite number of steps. Since a particular operation may not complete in a finite number of steps, it not possible to perform a worst-case analysis for non-blocking data structures. However, such implementations are desirable since they are typically less complicated than wait-free implementations and often perform well in practice. Amortized analysis gives an upper bound on the worst-case number of steps performed during a sequence of operations in an execution, rather than on the worst-case step complexity of a single operation. This type of analysis is useful for expressing the efficiency of non-blocking data structures.

In this paper, we present an amortized analysis of a non-blocking chromatic tree. The chromatic tree was introduced by Nurmi and Soisalon-Soininen [12] as a generalization of a red-black tree with relaxed balance conditions, allowing insertions and deletions to be performed independently of rebalancing. Boyar, Fagerberg, and Larsen [2] showed that the amortized number of rebalancing transformations per update is constant, provided each

update and each rebalancing transformation is performed without interference from other operations.

Brown, Ellen, and Ruppert [5] gave a non-blocking implementation using LLX and SCX primitives [4], which themselves can be implemented using CAS. Unfortunately, this implementation does not have good amortized complexity. We prove that a slightly modified implementation has amortized cost $O(\dot{c}(\alpha) + \log n(op))$ for each operation op in an execution α, where $\dot{c}(\alpha)$ is the point contention of α and $n(op)$ is the maximum number of nodes in the chromatic tree during op's execution interval.

Our amortized analysis for the chromatic tree is based on the amortized analysis for the unbalanced binary search tree by Ellen, Fatourou, Helga, and Ruppert [7]. Several challenges make our analysis more difficult than their analysis.

The unbalanced binary search tree only has one transformation to perform insertions and one transformation to perform deletions. The chromatic tree also has 11 different rebalancing transformations (not including their mirror images). Our analysis has the feature that it does not require a separate case for each type of transformation performed. A very similar analysis should give the same amortized step complexity for other types of balanced binary search trees, especially for those implemented using LLX and SCX [3].

Unlike update transformations, which only occur at leaf nodes, rebalancing transformations may occur anywhere in the tree. Furthermore, operations may perform rebalancing transformations on behalf of other operations, or may cause them to perform additional rebalancing transformations they would otherwise not perform. For an amortized analysis done using the accounting method, these facts make it much more difficult to determine the steps in which dollars should be deposited into bank accounts to pay for future steps.

Much like the analysis of the unbalanced binary search tree, many overlapping transformations can cause all but one to fail. Many operations may also be working together to perform the same rebalancing transformations. The amortized analysis of the chromatic tree is complicated by the fact that the constant upper bound on the number of rebalancing transformations per operation is amortized, even when each update and rebalancing transformation is performed atomically. Since the number of rebalancing transformations performed by each operation may differ, a bad configuration in which many rebalancing transformations can occur may also have high contention. Executions including such configurations must be accounted for in the amortized analysis.

The remainder of this paper is organized as follows. In Section 2, we present the asynchronous shared memory model assumed in our analysis. In Section 3, we give an overview of related work done on the amortized analysis of concurrent data structures. In Section 4, we give an overview of the modified chromatic tree implementation. Finally, in Section 5, we determine its amortized step complexity.

2 Model

Throughout this paper, we use an asynchronous shared memory model [1]. Shared memory consists of a collection of shared variables accessible by all processes in a system. Processes communicate using read, write, and CAS primitives.

A *configuration* of a system consists of the values of all shared variables and the states of all processes. A *step* by a process applies a primitive to a shared variable and can also change the state of the process. An *execution* is an alternating sequence of configurations and steps, starting with a configuration. A *solo execution* from a configuration C by a process P is an execution starting from C in which all steps are performed by P.

An *abstract data type* is a collection of mathematical objects and operations that satisfies certain properties. A *concurrent data structure* for the abstract data type provides representations of the objects in shared memory and algorithms for the processes to perform the operations. An operation on a data structure by a process becomes *active* when the process performs the first step of its algorithm. The operation becomes *inactive* after the last step of the algorithm is performed by the process. The *execution interval* of the operation consists of all configurations in which it is active. In the initial configuration, the data structure represents an empty instance of the abstract data type and there are no active operations.

The *worst-case step complexity* of an operation is the maximum number of steps taken by a process during the execution interval of any instance of this operation in any execution. The *amortized step complexity* of a data structure is the maximum number of steps in any execution consisting of operations on the data structure, divided by the number operations invoked in the execution. One can determine an upper bound on the amortized step complexity by assigning an amortized cost to each operation, such that for all possible executions α on the data structure, the total number of steps taken in α is at most the sum of the amortized costs of the operations in α. The amortized cost of a concurrent operation is often expressed as a function of contention. For a configuration C, we define its *contention* $\dot{c}(C)$ to be the number of active operations in C. For an operation op in an execution α, we define its *point contention* $\dot{c}(op)$ to be the maximum number of active operations in a single configuration during the execution interval of op. Finally, for an execution α, we define its point contention $\dot{c}(\alpha)$ to be the maximum number of active operations in a single configuration during α.

3 Related Amortized Analyses of Non-blocking Data Structures

In this section, we give an overview of the amortized analyses of various non-blocking data structures. While there are many papers that give implementations of non-blocking data structures, there are relatively few that give the amortized step complexity of their implementations.

Fomitchev and Ruppert [9] modify an implementation of a singly linked list by Harris [11] with the addition of *backlinks*. A backlink is a pointer to the predecessor of a node that has been *marked* for deletion. Whenever an update operation fails, the backlinks can be followed until an unmarked node is reached, rather than restarting searches from the head of the linked list. They prove an amortized step complexity of $O(n(op) + \dot{c}(op))$ for their improved implementation, where $n(op)$ is the number of elements in the linked list when operation op is invoked. The amortized analysis is done by charging each failed CAS and each backlink traversal performed by an operation op to a successful CAS of a concurrent operation in the execution interval of op. Gibson and Gramoli [10] give a simplified implementation of a linked list and claim it has the same amortized step complexity.

Ellen, Fatourou, Ruppert, and van Breugel [8] give the first provably correct non-blocking implementation of an unbalanced binary search tree using CAS primitives. Ellen, Fatourou, Helga, and Ruppert [7] show that this implementation has poor amortized step complexity. They improve the implementation by allowing failed update attempts to backtrack to a suitable internal node in the tree, rather than restarting attempts from the root. They prove for their implementation, each operation op has amortized cost $O(h(op) + \dot{c}(op))$, where $h(op)$ is the height of the binary search tree when op is invoked. The amortized analysis is difficult because a single DELETE operation can cause a sequence of overlapping DELETE operations by other processes to fail. The accounting method is used to show how all failed attempts in an execution can be paid for. Chatterjee, Nguyen and Tsigas [6] give a different

implementation of an unbalanced binary search tree, and give a sketch that the amortized cost of each operation in their implementation is also $O(h(op) + \dot{c}(op))$.

Shafiei [13] gives an implementation of a non-blocking doubly-linked list. She proves an amortized cost of $O(\dot{c}(op))$ for update operations op. The analysis closely follows the analysis of the unbalanced binary search tree [7], except modified to use the potential method.

4 The Non-blocking Implementation of the Chromatic Tree

A chromatic tree is a data structure for a dynamic set of elements, each with a distinct key from an ordered universe. It supports the following three operations:

- INSERT(k), which adds an element with key k into the set and returns TRUE if the set does not contain an element with key k; otherwise it returns FALSE,
- DELETE(k), which removes the element with key k from the set and returns TRUE if there is such an element; otherwise it returns FALSE, and
- FIND(k), which returns TRUE if there is an element in the set with key k and FALSE otherwise.

A chromatic tree is a generalization of a red-black tree with relaxed balance conditions. It is *leaf-oriented*, so every element in the dynamic set represented by the data structure corresponds to a leaf and all nodes have 0 or 2 children. Each node in a chromatic tree has a non-negative weight. We call a node *red* if it has weight 0, *black* if it has weight 1, and *overweight* if it has weight greater than 1. The *weighted level* of a node x is the sum of node weights along the path from the root node to x. The balance conditions of red-black trees and chromatic trees as described in [2] are as follows.

▶ **Definition 1.** A *red-black tree* T satisfies following balance conditions:

B1. The leaves of T are black.
B2. All leaves of T have the same weighted level.
B3. No path from T's root to a leaf contains two consecutive red nodes.
B4. T has only red and black nodes.

▶ **Definition 2.** A *chromatic tree* T satisfies the following balance conditions:

C1. The leaves of T are not red.
C2. All leaves of T have the same weighted level.

Let x be a node that is in the chromatic tree with weight $x.w$. If x and its parent both have weight 0, then a *red-red violation* occurs at x. If $x.w > 1$, then $x.w - 1$ *overweight violations* occur at x. When a chromatic tree contains no violations, it satisfies the balance conditions of a red-black tree. The balance conditions of a red-black tree ensure its height is $O(\log n)$, where n is the number of nodes in the tree.

An example of a chromatic tree is shown in Figure 1. The number inside each node is its key and the number to its right is its weight. The weighted level of each leaf is 3. There is an overweight violation at the leaf with key 1, and a red-red violation at the internal node with key 6.

Rebalancing is done independently of insertions and deletions. This allows rebalancing to be done at times when there are fewer processes that want to perform operations. However, the chromatic tree may not be height balanced at all times. The following lemma proven by Boyar, Fagerberg, and Larsen [2] gives an upper bound on the total number of rebalancing transformations that can occur.

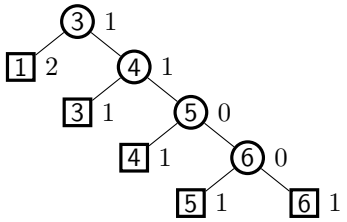

Figure 1 A chromatic tree containing elements with keys 1, 3, 4, 5 and 6.

▶ **Lemma 3.** *If $i > 0$ insertions and d deletions are performed on an initially empty chromatic tree, then at most $3i + d - 2$ rebalancing transformations can occur.*

In the non-blocking implementation by Brown, Ellen, and Ruppert [5], rebalancing is done as a part of INSERT and DELETE operations. They proved that the imbalance in the chromatic tree depends on the number of active operations.

▶ **Lemma 4.** *If there are c active* INSERT *and* DELETE *operations and the chromatic tree contains n elements, then its height is $O(c + \log n)$.*

4.1 An Implementation of a Chromatic Tree using LLX and SCX

An implementation of a chromatic tree was first given by Brown, Ellen, and Ruppert [5] using the LLX and SCX primitives. We describe a slightly modified implementation with better amortized step complexity. LLX and SCX are themselves implemented using CAS primitives. It is necessary to understand the step complexity of LLX and SCX to analyze the chromatic tree.

LLX and SCX operate on Data-records. A Data-record is a collection of fields used to represent a natural piece of a data structure. A successful LLX(r) on Data-record r returns a snapshot of the fields of r. A process can only perform SCX(V, R, fld, new) if it has previously performed a successful LLX on each Data-record in V since it last performed SCX (or the beginning of the execution if it has never performed SCX). A successful SCX(V, R, fld, new) *finalizes* all Data-records in $R \subseteq V$. It also updates fld, one field of a single Data-record in V, to the value *new*. This removes the Data-records in R from the data structure. An unsuccessful SCX returns FALSE, which only occurs when there is a concurrent SCX(V', R', fld', new') such that $V \cap V' \neq \emptyset$. Likewise an LLX can return FAIL if there is a concurrent SCX(V', R', fld', new') such that $r \in V'$. An LLX(r) returns FINALIZED if r has been previously finalized by an SCX. In both of these cases, the LLX is unsuccessful.

Each *node* of the chromatic tree is represented by one Data-record that contains fields for its key, weight, child pointers, *marked* bit, and *info* pointer. The marked bit is used to finalize nodes. A node is only removed from the chromatic tree after it is marked. Once a marked bit is set to TRUE, it cannot be changed back to FALSE. The info pointer points to an SCX-record, which contains information required for processes to perform a pending SCX on behalf of another process. It contains a *state* field whose value is either InProgress, Aborted, or Committed, and is initially InProgress.

Consider an SCX(V, R, fld, new) performed by a process P. For the chromatic tree, each node of R is reachable from the node pointed to by fld by only following child pointers of nodes in R. P first creates a new SCX-record U for this SCX. A node r is *frozen* for U if $r.info$ points to U and either $U.state$ = InProgress, or $U.state$ = Committed and r is marked. Freezing acts like a lock on a node held by a particular operation. For each node

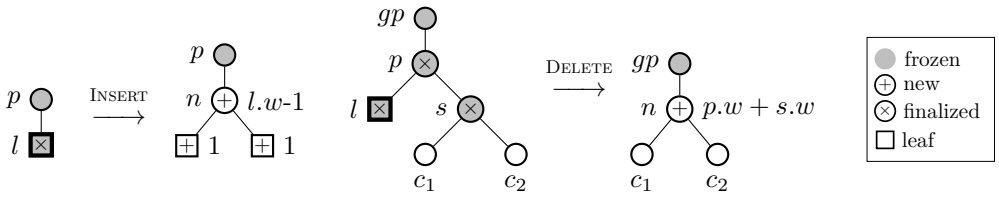

Figure 2 The update transformations of the chromatic tree.

$v \in V$, P attempts to freeze v by setting $v.info$ to point to U using a *freezing* CAS. This only succeeds if v has not been frozen since P's last LLX on v. If the freezing CAS fails and no other process has frozen v for U, then P performs an *abort step* and returns FALSE. An abort step atomically unfreezes all nodes frozen for U by setting U to the Aborted state.

Once all nodes in V are successfully frozen for U, the SCX can no longer fail. Then each node in $R \subseteq V$ is *marked* for removal and an *update* CAS sets fld to the value *new*. At this point, the nodes in R are no longer reachable from the root of the chromatic tree. Finally, a *commit step* is performed, which atomically unfreezes all nodes in $V \setminus R$ by setting U to the Committed state. The nodes in R remain frozen.

Consider an LLX(r) by a process P. If r is marked when the LLX is invoked, P will help the SCX that marked r (if it is not yet complete) and return FINALIZED. Otherwise P attempts to take a snapshot of the fields of r. If r is not frozen while P reads each of its fields, the LLX is successful and returns the fields of r. If a concurrent SCX operation freezes r sometime during P's LLX, P may help the concurrent SCX complete before returning FAIL.

LLX and SCX guarantee non-blocking progress. If SCX is performed infinitely often, then an infinite number succeed [4]. Much of our amortized analysis will require a mechanism to charge a failed LLX or SCX of one operation to a different operation that caused it to fail.

We give an overview of the algorithm to INSERT an element with key k into a chromatic tree. The algorithm for the DELETE operation is similar. The INSERT operation is divided into an *update phase* and a *cleanup phase*. An update phase repeatedly performs update attempts until one is successful.

An update attempt begins with a search through the tree for the key k, starting from the root, until it visits a leaf l. Let p denote the node visited immediately before visiting l. If $l.key = k$, then the key is already in the chromatic tree, so INSERT returns FALSE. Otherwise a subroutine TRYINSERT(p, l) is invoked, which attempts to perform an LLX on the nodes p and l, followed by a single SCX to apply the INSERT transformation shown in Figure 2. Gray nodes represent nodes on which LLXs and freezing CASs are performed. Nodes marked with $\times$ are removed from the chromatic tree, while nodes marked with $+$ are new nodes added to the tree. Any weight restrictions on a node are shown to its right. Notice that, to complete the transformation, an SCX is only required to update the pointer of a single node to point to a subtree of new nodes. The instance of TRYINSERT fails if any of its LLXs or SCXs are unsuccessful, in which case a new update attempt begins immediately following the failed LLX or SCX. If a violation is created during the insertion performed by a successful TRYINSERT, the operation enters its cleanup phase. Otherwise, the INSERT terminates.

Ellen, Fatourou, and Helga, and Ruppert [7] use a stack to recover from failed update attempts in their unbalanced binary search tree. Each time a process visits a node during its search, it pushes a pointer to this node onto a local stack. After a failed attempt, the process can pop nodes off its stack until an unmarked node is found. This allows the process

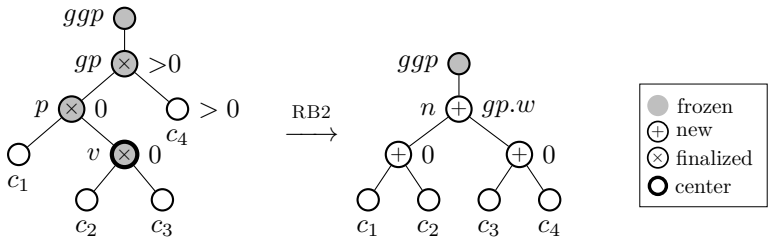

Figure 3 One of the 11 different rebalancing transformations.

to recover from a failed attempt without restarting its search from the root. A process is *backtracking* during the interval in which it pops nodes off its stack. As described in [7], update operations that restart searches from the root without using a stack have poor amortized cost. Consequently, we will consider an implementation of the chromatic tree using a stack to recover from failed attempts. The proof of correctness of this modification appears in the full version.

The cleanup phase is also divided into a series of attempts. At the start of the cleanup phase, the process empties its local stack and performs a new search for k starting from the root. When a new attempt begins, the process pops nodes off its stack until an unmarked node is found. Then it continues searching for k from this node. The cleanup phase terminates when the search reaches a leaf without encountering a violation. If a violation is encountered at a node v during the search, a subroutine TryRebalance(ggp, gp, p, v) attempts to remove the violation, where p, gp, and ggp are the last 3 nodes visited on the search to v. TryRebalance determines which of the 11 rebalancing transformations should be applied. It applies the rebalancing transformation by performing LLX on each node involved in the transformation and then using SCX to update one pointer. LLXs are performed from top to bottom. If two nodes involved in the transformation have the same height, then one of these is ggp, gp, p, or v, and an LLX is performed on this node first. If any LLX or SCX is unsuccessful, TryRebalance immediately returns. Whether or not TryRebalance successfully performs a rebalancing transformation, a new attempt begins. A cleanup attempt is successful if it performs a successful SCX during TryRebalance; otherwise it is unsuccessful.

For our analysis, we identify a single node for each rebalancing transformation as its *center*. In Figure 3, the node with a bold outline is the center node. The center node of a transformation always contains a violation. A rebalancing transformation is *centered* at a violation if the violation occurs at the center node of the transformation. We classify each node involved in an SCX as either a downwards node or a cross node.

▶ **Definition 5.** Consider an instance S of SCX(V, R, fld, new), where $V = \{v_1, \ldots, v_k\}$, enumerated in the order they are frozen. For $1 \leq i < k$, v_i is a *cross node* for S if v_i is the sibling of v_{i+1}, otherwise v_i is a *downwards node* for S. We define v_k to be a downwards node for S.

Only Insert transformations create red-red violations and only Delete transformations create overweight violations. For the Insert transformation shown in Figure 2, if $p.w = 0$ and $l.w = 1$, then a red-red violation is created at the new node n. The Delete transformation creates an overweight violation at the new node n if $p.w > 0$ and $s.w > 0$, and moves any overweight violations from p or s to n.

Rebalancing transformations remove at most 1 violation. Consider the rebalancing transformation RB2 shown in Figure 3. The red-red violation located at the center node v is removed. If $gp.w > 1$, then there are $gp.w - 1$ overweight violations at gp. We say that these overweight violations are moved to the new node n. Similar definitions can be made for the remaining 10 rebalancing transformations.

Some rebalancing transformations may move violations, without removing any violations. Thus, several rebalancing transformations may be centered at a violation before it is eventually removed.

▶ **Definition 6.** If the update phase of an operation creates a violation and, hence, begins a cleanup phase, cp, let $viol(cp)$ denote this violation. Let $rebal(viol(cp))$ be the number of times during the execution $viol(cp)$ is located at the center of a successful rebalancing transformation.

Intuitively, our analysis will charge cp for every successful rebalancing transformation centered at $viol(cp)$.

We can show that cp does not terminate until $viol(cp)$ is removed. Since an update transformation creates at most 1 violation, the number of violations in the chromatic tree is bounded by the number of incomplete Insert and Delete operations. This fact is used to show that Lemma 4 continues to hold when backtracking is used.

5 Chromatic Tree Amortized Analysis

In this section, we present an amortized step complexity of the chromatic tree. We use the following notation.

▶ **Definition 7.** For each configuration C and operation op with update phase up and cleanup phase cp of an execution α, let:

- $h(C)$ be the height of the chromatic tree in configuration C,
- $h(up)$ and $h(cp)$ be the height of the chromatic tree in the starting configurations of up and cp, respectively,
- $h(op)$ be the maximum height of the chromatic tree over all configurations in the execution interval of op, and
- $\dot{c}(up)$ and $\dot{c}(cp)$ be the maximum number of active operations in a single configuration in the execution interval of up and cp respectively.

We first give a sketch of our amortized analysis. Its structure is similar to the amortized analysis of a binary search tree by Ellen et al [7].

▶ **Definition 8.** For an update phase or cleanup phase xp, let $attempts(xp)$ be the number of attempts made by xp, and $pushes(xp)$ be the number of times xp pushes a node onto its stack.

The number of steps an operation takes during searches is a constant times the number of pushes it performs. Each attempt performs a constant number of LLXs and SCXs, each of which takes a constant number of steps. These observations imply the following result.

▶ **Lemma 9.** *The number of steps taken by an operation op with update phase up and cleanup phase cp is $steps(op) = O(attempts(up) + attempts(cp) + pushes(up) + pushes(cp))$. If op has no cleanup phase, then $steps(op) = O(attempts(up) + pushes(up))$.*

In the amortized analysis of the binary search tree [7], it is shown how the total number of pushes can be bounded by a function of the total number of attempts, the height of the tree at the beginning of each operation, and point contention. For the chromatic tree, we similarly prove that for any finite execution α,

$$\sum_{up\in\alpha} pushes(up) + \sum_{cp\in\alpha} pushes(cp) \le \sum_{up\in\alpha} [2\cdot attempts(up) + h(up) + 4\dot{c}(up)] + \sum_{cp\in\alpha} [3\cdot attempts(cp) + h(cp) + 10\dot{c}(cp)\cdot rebal(viol(cp))].$$

The majority of our amortized analysis is devoted to showing that

$$\sum_{up\in\alpha} attempts(up) \le \sum_{up\in\alpha} [30h(up) + 5594\dot{c}(up) + 159]$$

and

$$\sum_{cp\in\alpha} attempts(cp) \le \sum_{cp\in\alpha} [78h(cp) + 312\dot{c}(cp) + 5008\dot{c}(cp)\cdot rebal(viol(cp)) + 357].$$

Note that the constants used are larger than required to simplify the analysis. Combining these results with Lemma 9 gives the following lemma.

▶ **Lemma 10.** *For any finite execution α of* Insert, Delete, *and* Find *operations on the chromatic tree,*

$$\sum_{op\in\alpha} steps(op) = O\left(\sum_{up\in\alpha} [h(up) + \dot{c}(up)] + \sum_{cp\in\alpha} [h(cp) + rebal(viol(cp))\cdot\dot{c}(cp)]\right).$$

The height terms $h(up)$ and $h(cp)$ in Lemma 10 are the number of steps to perform a search in the update phase and cleanup phase, respectively, assuming there are no concurrent operations. The $\dot{c}(up)$ terms account for extra steps taken by operations concurrent with up due to the successful insertion or deletion made by up. Similarly, the $rebal(viol(cp))\cdot\dot{c}(cp)$ terms account for extra steps taken due to successful rebalancing transformations centered at $viol(cp)$.

Lemma 10 and Lemma 3 together prove the main theorem of our paper.

▶ **Theorem 11.** *The amortized number of steps made by any chromatic tree operation op in any finite execution α is $O(\dot{c}(\alpha) + \log n(op))$, where $n(op)$ is the maximum number of nodes in the chromatic tree during op's execution interval.*

Proof. For any operation op with update phase up and cleanup phase cp, $h(up) \le h(op)$ and $h(cp) \le h(op)$ since the starting configurations of up and cp are contained in op. Therefore,

$$\sum_{up\in\alpha} h(up) + \sum_{cp\in\alpha} h(cp) \le \sum_{op\in\alpha} 2h(op).$$

Similarly, $\dot{c}(up) \le \dot{c}(\alpha)$ and $\dot{c}(cp) \le \dot{c}(\alpha)$ since up and cp are intervals contained in α. Therefore, Lemma 10 can be expressed as

$$\sum_{op\in\alpha} steps(op) = O\left(\sum_{op\in\alpha} [h(op) + \dot{c}(\alpha)] + \dot{c}(\alpha)\cdot\sum_{cp\in\alpha} rebal(viol(cp))\right).$$

Suppose the execution α contains i INSERT operations and d DELETE operations. Then, by Lemma 3,

$$\begin{aligned}\dot{c}(\alpha) \cdot \sum_{cp \in \alpha} rebal(viol(cp)) &\leq \dot{c}(\alpha)(3i + d - 2) \\ &\leq \dot{c}(\alpha)(3i + 3d) \\ &\leq \sum_{op \in \alpha} 3\dot{c}(\alpha).\end{aligned}$$

So, the total number of steps taken by all operations in an execution α is

$$\sum_{op \in \alpha} steps(op) = O\Big(\sum_{op \in \alpha} [\dot{c}(\alpha) + h(op)] \Big).$$

It follows that the amortized cost of a single operation op is $O(\dot{c}(\alpha) + h(op)) = O(\dot{c}(\alpha) + \log n(op))$, by Lemma 4. ◀

Notice that the amortized cost of each operation has an additive term of $O(\dot{c}(\alpha))$, as opposed to $O(\dot{c}(op))$. This is because the operation with the largest contention in the execution may perform a lot of the rebalancing. Lemma 3 only gives an upper bound on the total number of rebalancing transformations that may occur in an execution, but does not state anything about which operations will perform these rebalancing transformations.

The analysis for bounding the total number of attempts made in update phases in the chromatic tree is similar to the analysis for bounding the total number of attempts made in the unbalanced binary search tree. In the remaining sections, we give an overview of the analysis for bounding the total number of attempts made in cleanup phases. The complete amortized analysis appears in the full version of the paper.

5.1 Bounding the Number of Failed Cleanup Attempts

In this section, we count the number of cleanup attempts that fail due to unsuccessful LLXs. Counting the number of cleanup attempts that fail due to unsuccessful SCXs is similar. Lemma 3 gives an upper bound on the number of successful cleanup attempts, so we focus on counting unsuccessful cleanup attempts.

Our analysis uses the accounting method. We define a number of bank accounts for each cleanup phase cp. The rules that deposit and withdraw dollars from bank accounts are designed so that the *bank* (i.e. the collection of all bank accounts) satisfies the following properties for any finite execution.

- Property P1: The total number of dollars deposited into the bank by an operation is $O(h(cp) + rebal(viol(cp)) \cdot \dot{c}(cp))$ during its cleanup phase cp.
- Property P2: Every failed attempt caused by a failed LLX during an operation's cleanup phase cp withdraws one dollar from one of its own accounts.
- Property P3: All bank accounts have non-negative balance.

Assuming these properties of the bank hold, the total number of attempts that fail due to LLXs is bounded by the total number of dollars deposited into the bank. Thus, the total number of attempts that fail due to LLXs is $\sum_{cp \in \alpha} O(h(cp) + rebal(viol(cp)) \cdot \dot{c}(cp))$.

Let $\#llx(cp)$ be the maximum number of LLXs performed by cp in a single attempt. It can be verified that $\#llx(cp) \leq 7$. For each node x in the chromatic tree at some point in the execution, and for each cleanup phase cp, we define the accounts $B(cp)$, $B_{llx}(cp, x)$, $S(cp, x)$ and $L_i(cp)$, for $1 \leq i \leq \#llx(cp)$. For each type of bank account $X(args)$, we let $X(args, C)$

Table 1 Rules for bank accounts owned by a cleanup phase cp.

D1-S	Consider a **successful update CAS** *ucas* for a rebalancing transformation centered at $viol(cp)$. For all nodes x removed from the chromatic tree by *ucas* and for all active cleanup phases cp', cp deposits 1 dollar into $S(cp', x)$.
D1-L	The **start of a cleanup phase** cp deposits 3 dollars into each of its own $L_i(cp)$ accounts (for $1 \leq i \leq \#llx(cp)$).
D2-L	A **successful update CAS** for a rebalancing transformation centered at $viol(cp)$ deposits 3 dollars into each active L_i account.
D1-B	The **start of a cleanup phase** cp deposits $78h(cp) + 312\dot{c}(cp) + 312$ dollars into its own $B(cp)$ account.
D2-B	A **successful update CAS** of a rebalancing transformation centered at $viol(cp)$ deposits 4934 dollars into each active B account.
D3-B	A **successful commit step** of a rebalancing transformation centered at $viol(cp)$ deposits 26 dollars into each active B account.
W-S	A **stale invocation** of TryRebalance for a cleanup phase cp that blames a node x withdraws 1 dollar from $S(cp, x)$.
W-L	Suppose that cp fails its ith LLX during a non-stale instance of TryRebalance. If $L_i(cp)$ is non-empty, the **failure step of the LLX** withdraws 1 dollar from $L_i(cp)$. If $L_i(cp)$ is empty, consider the node x on which this LLX is performed. If x is a downwards node for the SCX blamed by the LLX, then cp withdraws 1 dollar from $B_{llx}(cp, x)$; otherwise it withdraws from $B_{llx}(cp, p)$, where p is the parent of x.
T-B	Consider a **successful freezing CAS** from a configuration C performed by any process on a downwards node x for some SCX. Every cleanup phase cp, where $x \in targets(cp, C)$, transfers 1 dollar from $B(cp)$ to $B_{llx}(cp, x)$.

be the number of dollars in $X(args)$ in a configuration C. A bank account $X(cp)$ or $X(cp, x)$ is *active* if cp is active. Each bank account is initially empty.

The bank accounts $S(cp, x)$ and $L_i(cp)$ will serve two purposes. First, they pay for any failed attempts due to unsuccessful LLXs of cp due to steps that occur prior to the start of cp. Second, they will pay for cp's failed attempts due to unsuccessful LLXs of cp due to recent changes in the structure of the chromatic tree. Whenever the $S(cp, x)$ and $L_i(cp)$ accounts do not pay for a failed attempt, a dollar will be withdrawn from one of the $B_{llx}(cp, x)$ accounts instead. The $B(cp)$ account is only responsible for transferring dollars into the $B_{llx}(cp, x)$ accounts.

The rules for bank accounts owned by cp are summarized in Table 1. Some terms in the rules will be defined when introduced in the following sections. There are 6 deposit rules beginning with D. They indicate which accounts cp deposits dollars into. There are two withdraw rules beginning with W. They define when cp withdraws dollars from its accounts. There is one transfer rule T-B that transfers dollars from $B(cp)$ to $B_{llx}(cp, x)$.

We next show that Properties P1 and P2 are always satisfied. Recall that a successful SCX contains 1 successful update CAS, 1 successful commit step, and a number of successful freezing CASs.

▶ **Lemma 12.** *The total number of dollars deposited into the bank by an operation is* $O(h(cp) + rebal(viol(cp)) \cdot \dot{c}(cp))$ *during its cleanup phase* cp.

Proof. Consider a process in its cleanup phase cp. Rules D1-L and D1-B are each applied once, at the start of cp. Rule D1-L only deposits a constant number of dollars. Rule D1-B deposits $O(h(cp) + \dot{c}(cp))$ dollars. By definition, there are $rebal(viol(cp))$ successful

rebalancing transformations centered at $viol(cp)$, so each of rules D1-S, D2-L, D2-B, and D3-B are applied $rebal(viol(cp))$ times throughout cp. There are at most $\dot{c}(cp)$ active operations each time a rule is applied. It can be verified by inspection of the rebalancing transformations that at most 5 nodes are removed from the chromatic tree by a single update CAS, so rule D1-S deposits $O(\dot{c}(cp))$ dollars each time it is applied. Since $\#llx(cp) \leq 7$, rule D2-L deposits $O(\dot{c}(cp))$ dollars each time it is applied. By the rules in Table 1, cp deposits a total of $O(h(cp) + rebal(viol(cp)) \cdot \dot{c}(cp))$ dollars. Thus, Property P1 of the bank is satisfied. ◀

▶ **Lemma 13.** *Every failed attempt caused by a failed LLX during an operation's cleanup phase cp withdraws one dollar from one of its own accounts.*

Proof. Every cleanup attempt contains at most 1 unsuccessful LLX. The LLXs performed by cp only occur during an instance of TryRebalance. By rule W-S, if this instance is *stale* (which will be defined in the following section), a dollar is withdrawn from one of cp's S accounts. Otherwise, by rule W-L, a dollar is withdrawn from one of cp's L_i or B_{llx} accounts. Thus, Property P2 of the bank is satisfied. ◀

It remains to show that the balance of each bank account is non-negative. In Section 5.2, we consider the $S(cp, x)$, $L_i(cp)$, and $B_{llx}(cp, x)$ accounts, and in Section 5.3, we consider the $B(cp)$ accounts.

5.2 The $S(cp, x)$, $L_i(cp)$, and $B_{llx}(cp, x)$ Accounts

In this section, we describe the purpose of the $S(cp, x)$, $L_i(cp)$, and $B_{llx}(cp, x)$ accounts, and show that the balance in these accounts are non-negative.

▶ **Definition 14.** An instance I of TryRebalance(ggp, gp, p, v) starting from configuration C is called *stale* if one of ggp, gp, p, or v is not in the chromatic tree in C. If x is the first node in the sequence $\langle ggp, gp, p, v\rangle$ that is no longer in the chromatic tree in C, then I *blames* the node x.

The $S(cp, x)$ account pays for all failed attempts containing a stale instance of TryRebalance(ggp, gp, p, v) that blame a node x. According to D1-S, whenever cp is active and a node x is removed from the chromatic tree, 1 dollar is deposited into $S(cp, x)$. It can be shown that TryRebalance will blame a node x at most once sometime after x is removed from the chromatic tree, since in cp's following cleanup attempt, cp will restart its search from a node from which x is no longer reachable. It follows that $S(cp, x, C) \geq 0$.

The $L_i(cp)$ or $B_{llx}(cp, x)$ accounts are used to pay for each failed attempt due to an unsuccessful LLX during a non-stale instance of TryRebalance. We require a number of definitions to explain why at least one of these bank accounts has a positive balance when these LLXs occur. During a cleanup phase cp by a process P, P traverses from the root to a leaf, attempting to cleanup any violations it encounters during its traversal. When cp is performing a rebalancing transformation centered at a node v in the chromatic tree in a configuration C, we define $focalNode(cp, C) = v$. If not, but the node cp is currently visiting in its search is in the chromatic tree, then $focalNode(cp, C)$ is defined to be this node. Otherwise, $focalNode(cp, C)$ is the next node in the chromatic tree that cp will visit in a solo execution starting from C. Let $focalPath(cp, C)$ be the set of all nodes along the path from the root to $focalNode(cp, C)$, including both endpoints.

We define a set of nodes $rebalSet(v, C)$ so that any rebalancing transformation whose center node is v only involves nodes in $rebalSet(v, C)$. We use $rebalSet(v, C)$ to simplify

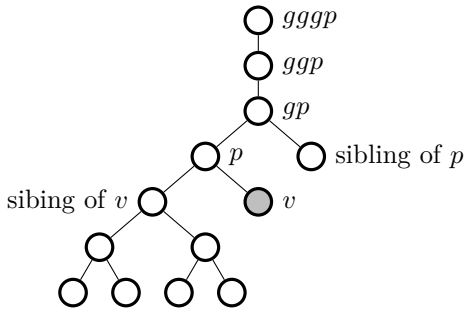

Figure 4 The set of nodes $rebalSet(v, C)$.

the amortized analysis so that each of the 11 rebalancing transformations do not have to be considered individually. This set of nodes is illustrated in Figure 4.

▶ **Definition 15.** For any node v in the chromatic tree in configuration C, let $rebalSet(v, C)$ be the set of nodes including the following 13 nodes:

- v and its four closest ancestors (labeled p, gp, ggp, and $gggp$),
- p's sibling, and
- all nodes with depth 2 or less in the subtree rooted at v's sibling.

For every cleanup phase cp and configuration C in which cp is active, let $targets(cp, C) = rebalSet(focalNode(cp, C), C)$. The following lemma concerning non-stale instances of TryRebalance is used to prove properties of the $L_i(cp)$ accounts.

▶ **Lemma 16.** *In the first configuration C of a non-stale* TryRebalance(ggp, gp, p, v) *performed by cp, $targets(cp, C) = rebalSet(v, C)$.*

For every unsuccessful LLX that occurs during a non-stale instance of TryRebalance, we choose an SCX which causes the LLX to fail.

▶ **Definition 17.** Let I be an LLX(r) that returns Fail or Finalized. Let the *failure step* of I be the last time I reads $r.info$, the pointer to r's SCX-record. Let U be the SCX-record pointed to by $r.info$ when I performs its failure step. We say that I *blames* the SCX that created U.

Consider a failed attempt by cp due to an unsuccessful LLX during a non-stale instance of TryRebalance. By rule W-L, if cp's ith LLX during this instance of TryRebalance is unsuccessful and $L_i(cp)$ is non-empty, then cp withdraws 1 dollar from $L_i(cp)$. Since this is the only rule that withdraws from $L_i(cp)$, the $L_i(cp)$ account always has a non-negative balance. Rules D1-L and D2-L show that $L_i(cp)$ is non-empty during cp's first 3 attempts, or when a successful update CAS (by some other process) has occurred during cp's previous 3 attempts. We can use these facts along with Lemma 16 to prove the following lemma.

▶ **Lemma 18.** *Suppose a cleanup phase cp fails its ith LLX during a non-stale instance of* TryRebalance*. Let x be the node on which this LLX is performed. If $L_i(cp)$ does not pay for the unsuccessful LLX(x), then $x \in targets(cp, C')$ in the configuration C' immediately before the successful freezing CAS on x during the SCX blamed by the LLX(x).*

If $L_i(cp)$ is empty, the failed attempt is paid for by a $B_{llx}(cp, x)$ account. No rules deposit directly into $B_{llx}(cp, x)$ accounts. Dollars are only transferred into $B_{llx}(cp, x)$ from $B(cp)$ by rule T-B. Lemma 18 can be used to show that each time rule W-L withdraws a dollar from $B_{llx}(cp, x)$, rule T-B has previously transferred a dollar into this account.

▶ **Lemma 19.** *Consider the failure step of an unsuccessful LLX(x) that withdraws 1 dollar from $B_{llx}(cp, x)$. In the configuration C immediately before the failure step, $B_{llx}(cp, x, C) > 0$.*

To see why Lemma 19 is true, let S be the SCX blamed by the unsuccessful LLX(x). Let C' be the configuration immediately before the successful freezing CAS on x during S. Since a dollar is withdrawn from $B_{llx}(cp, x)$ to pay for this LLX, Lemma 18 implies that $x \in \mathit{targets}(cp, C')$. Furthermore, by rule W-L, x is a downwards node for S. Therefore, by rule T-B, the successful freezing CAS that freezes x for S transfers 1 dollar from $B(cp)$ into $B_{llx}(cp, x)$. Using properties of LLX, we can prove that no other unsuccessful LLX by cp withdraws this dollar between C' and C. Therefore, $B_{llx}(cp, x, C) > 0$.

According to rule W-L, an unsuccessful LLX(x) may instead withdraw a dollar from $B_{llx}(cp, p)$, where p is the parent of x. Results similar to Lemma 18 and Lemma 19 show that, in this case, $B_{llx}(cp, p)$ is also non-empty. Therefore, the B_{llx} accounts always have non-negative balance.

5.3 The $B(cp)$ Accounts

Rule T-B is the only rule that reduces the number of dollars in $B(cp)$. Since unsuccessful SCXs may contain successful freezing CASs, the total number of dollars transferred from $B(cp)$ throughout cp may be large. We show that, whenever T-B is applied, the balance in $B(cp)$ remains non-negative. We will define a function $J(cp, C)$ and show that, for every configuration C, $B(cp, C) \geq J(cp, C) \geq 0$.

We require some definitions. For any variable $z(args, C)$, we use $z(args)$ to denote $z(args, C)$ as a function of C. For any node x in the chromatic tree in configuration C, let

- $\mathit{depth}(x, C)$ be the number of edges along the path from the root to x in C,
- $\mathit{isFrozen}(x, C)$ be 1 if x is frozen in C and 0 otherwise, and
- $\mathit{npa}(x, C)$ be the set of nodes that are not proper ancestors of x in C.

▶ **Definition 20.** For each node x in the chromatic tree at some point in an execution, define $\mathit{abort}(x, C)$, which is 0 in initial configurations and is updated by the following four rules:

- A1: A successful freezing CAS on a downwards node x sets $\mathit{abort}(p) = 1$, where p is the parent of x.
- A2: A successful freezing CAS on a cross node x sets $\mathit{abort}(x) = 1$.
- A3: A successful abort step that unfreezes node x sets $\mathit{abort}(x) = 0$.
- A4: The completion of an update or cleanup attempt sets $\mathit{abort}(x) = 0$ for all nodes x except for those on which LLX(x) has been performed in the latest attempt of an active operation.

Next we define $H(x, C)$, which maps node x in the chromatic tree in configuration C to an integer. A similar function is defined in the analysis of the unbalanced binary search tree [7]. For any node x in the chromatic tree in configuration C, let

$$H(x, C) = 3h(C) + 12 - 3\mathit{depth}(x, C) + 6\dot{c}(C) + \sum_{u \in \mathit{npa}(x,C)} [\mathit{abort}(u, C) - \mathit{isFrozen}(u, C)].$$

▶ **Lemma 21.** *For any node x in the chromatic tree in configuration C, $H(x, C)$ has the following properties.*

1. $0 \leq H(x, C) \leq 3h(C) + 12\dot{c}(C) + 12$.
2. *If p is the parent of x, $2 \leq H(p, C) - H(x, C) \leq 4$.*
3. *If s is the sibling of x, $H(s, C) = H(x, C)$.*

4. *A successful freezing CAS on a downwards node x of an SCX decreases $H(x)$ by 1, and does not increase $H(u)$ for any other node u in the chromatic tree.*
5. *A successful commit step of an SCX increases $H(x)$ by at most 1.*
6. *A successful abort step of an SCX does not increase $H(x)$.*

▶ **Definition 22.** For every cleanup phase cp and every node x in the chromatic tree at the beginning of cp, $backup(cp, x) = 0$. When an update CAS by any process adds a new node x into the chromatic tree during cp, $backup(cp, x)$ is initialized to 1 if x is on $focalPath(cp)$, and 0 otherwise. Finally, a step that removes a node x from $focalPath(cp)$ sets $backup(cp, x) = 0$.

If $backup(cp, x, C) = 1$, then x is a node that has been added as a proper ancestor of $focalNode(cp, C)$ since cp began. When cp backtracks up the tree, so that x is no longer on $focalPath(cp)$, the decrease in $backup(cp, x)$ will ensure that $J(cp)$ does not increase.

▶ **Definition 23.** For any cleanup phase cp and any configuration C in which cp is active,

$$J(cp, C) = \sum_{x \in targets(cp,C)} 2H(x, C) + \sum_{x \in focalPath(cp,C)} 576 \cdot backup(cp, x, C).$$

By Lemma 21.1, $H(x, C) \geq 0$ for all nodes x and configurations C. By definition, $backup(cp, x, C) \geq 0$. Hence $J(cp, C)$ is always non-negative.

▶ **Lemma 24.** *For all cleanup phases cp and configurations C in which cp is active, $B(cp, C) \geq J(cp, C) \geq 0$.*

Proof sketch. In the first configuration C' in which cp is active, $backup(cp, x, C') = 0$ for all nodes x. By Lemma 21.1, $2H(x, C') \leq 6h(C') + 24\dot{c}(C') + 24$. Since $|targets(cp, C')| = 13$, $J(cp, C') \leq 78h(C') + 312\dot{c}(C') + 312$. Thus, by rule D1-B, $B(cp, C') \geq J(cp, C')$. We show that, whenever a step subsequently decreases the balance in $B(cp)$, $J(cp)$ decreases by at least the same amount, and whenever a step subsequently increases $J(cp)$, the balance in $B(cp)$ increases by at least the same amount. This implies that, in all configurations C in which cp is active, $B(cp, C) \geq J(cp, C)$.

Rule T-B decreases the balance in $B(cp)$ by 1 when a successful freezing CAS is performed on a downwards node x in $targets(cp, C)$. Lemma 21.4 says that, when this happens, $H(x)$ decreases by 1 and the H values of all other nodes do not change, so $J(cp)$ decreases by 2. No other steps decrease the balance of $B(cp)$.

Only successful commit steps and update CASs increase $J(cp)$. For example, Lemma 21.2 and Lemma 21.3 can be used to show that, when $focalNode(cp)$ changes from a node p to one of its children x, $J(cp)$ decreases as a result of the changes to $targets(cp)$ and $focalPath(cp)$. Furthermore, by Lemma 21.6, abort steps do not increase $J(cp)$.

By Lemma 21.5, a commit step increases $H(x)$ by at most 1, for all nodes x in the chromatic tree. Thus, $J(cp)$ increases by at most $2|targets(cp, C)| = 26$. Rule D3-B deposits 26 dollars into $B(cp)$, and so $B(cp, C) \geq J(cp, C)$.

A successful update CAS can change $targets(cp)$. We use the fact that rebalancing transformations only change a localized section of the tree to calculate an upper bound on the difference between the H values of cp's new targets and old targets. This is done independently of the type of rebalancing transformation applied by the update CAS.

Because an update CAS may remove nodes from the chromatic tree that are on cp's stack, $focalNode(cp)$ may change to a new node with much smaller depth. As a result, each term in the first summation in $J(cp)$ may increase by an amount that is proportional to the change in depth of $focalNode(cp)$. The $backup(cp, x)$ variables are used to offset this increase. We

show that whenever $\mathit{focalNode}(cp)$ decreases in depth, a proportional number of nodes x with $\mathit{backup}(cp, x) = 1$ are removed from $\mathit{focalPath}(cp)$. This decreases the second summation in $J(cp)$ to offset the increase in the first summation in $J(cp)$. We show that $J(cp)$ increases by at most 4934 as a result of an update CAS, so the number of dollars deposited into $B(cp)$ by rule D2-B is sufficient to maintain that $B(cp, C) \geq J(cp, C)$. ◀

Since all bank accounts owned by cp have non-negative balance, Property P3 of the bank is satisfied. Thus, the total number of attempts that fail due to unsuccessful LLXs in an execution α is $\sum_{cp \in \alpha} O(h(cp) + \mathit{rebal}(\mathit{viol}(cp)) \cdot \dot{c}(cp))$.

6 Conclusion

We have shown that the amortized step complexity of an implementation of a non-blocking chromatic tree is $O(\dot{c}(\alpha) + \log n(op))$. It would be interesting to see if amortized step complexity $O(\dot{c}(op) + \log n(op))$ can be shown for the chromatic tree. This is more challenging because one must argue that an operation with high contention does not perform an excessive amount of rebalancing. This may require a more detailed analysis of the chromatic tree than what is shown by Lemma 3. Alternatively, one may try to show a lower bound of $\Omega(\dot{c}(\alpha) + \log n(op))$. Finally, we believe that the techniques presented here can be applied to other balanced binary search tree implementations using LLX and SCX.

References

1 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience, 2nd edition, 2004.

2 Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization Results for Chromatic Search Trees, with an Application to Priority Queues. In *Proceedings of Algorithms and Data Structures, 4th International Workshop (WADS)*, pages 270–281, 1995. doi:10.1007/3-540-60220-8_69.

3 Trevor Brown. *Techniques for Constructing Efficient Lock-Free Data Structures*. PhD thesis, Department of Computer Science, University of Toronto, 2017.

4 Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, pages 13–22, 2013. doi:10.1145/2484239.2484273.

5 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 329–342, 2014. doi:10.1145/2555243.2555267.

6 Bapi Chatterjee, Nhan Nguyen Dang, and Philippas Tsigas. Efficient lock-free binary search trees. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 322–331, 2014. doi:10.1145/2611462.2611500.

7 Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, pages 332–340, 2014. doi:10.1145/2611462.2611486.

8 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, 2010. doi:10.1145/1835698.1835736.

9 Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 50–59, 2004. doi:10.1145/1011767.1011776.

10 Joel Gibson and Vincent Gramoli. Why Non-blocking Operations Should be Selfish. In *Proceedings of the Distributed Computing - 29th International Symposium (DISC)*, pages 200–214, 2015. doi:10.1007/978-3-662-48653-5_14.

11 Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the Distributed Computing, 15th International Conference (DISC)*, pages 300–314, 2001. doi:10.1007/3-540-45414-4_21.

12 Otto Nurmi and Eljas Soisalon-Soininen. Chromatic Binary Search Trees. A Structure for Concurrent Rebalancing. *Acta Inf.*, 33(6):547–557, 1996. doi:10.1007/BF03036462.

13 Niloufar Shafiei. Non-Blocking Doubly-Linked Lists with Good Amortized Complexity. In *Proceedings of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 35:1–35:17, 2015. doi:10.4230/LIPIcs.OPODIS.2015.35.