A General Technique for Non-blocking Trees

Trevor Brown[1], Faith Ellen[1] and Eric Ruppert[2]

University of Toronto[1], York University[2]

Abstract. We describe a general technique for obtaining provably correct, non-blocking implementations of a large class of tree data structures where pointers are directed from parents to children. Updates are permitted to modify any contiguous portion of the tree atomically. Our non-blocking algorithms make use of the LLX, SCX and VLX primitives, which are multi-word generalizations of the standard LL, SC and VL primitives and have been implemented from single-word CAS [10].
To illustrate our technique, we describe how it can be used in a fairly straightforward way to obtain a non-blocking implementation of a chromatic tree, which is a relaxed variant of a red-black tree. The height of the tree at any time is $O(c + \log n)$, where n is the number of keys and c is the number of updates in progress. We provide an experimental performance analysis which demonstrates that our Java implementation of a chromatic tree rivals, and often significantly outperforms, other leading concurrent dictionaries.

1 Introduction

The binary search tree (BST) is among the most important data structures. Previous concurrent implementations of balanced BSTs without locks either used coarse-grained transactions, which limit concurrency, or lacked rigorous proofs of correctness. In this paper, we describe a general technique for implementing *any* data structure based on a down-tree (a directed acyclic graph of indegree one), with updates that modify any connected subgraph of the tree atomically. The resulting implementations are non-blocking, which means that some process is always guaranteed to make progress, even if processes crash. Our approach drastically simplifies the task of proving correctness. This makes it feasible to develop provably correct implementations of non-blocking balanced BSTs with fine-grained synchronization (i.e., with updates that synchronize on a small constant number of nodes).

As with all concurrent implementations, the implementations obtained using our technique are more efficient if each update to the data structure involves a small number of nodes near one another. We call such an update *localized*. We use *operation* to denote an operation of the abstract data type (ADT) being implemented by the data structure. Operations that cannot modify the data structure are called *queries*. For some data structures, such as Patricia tries and leaf-oriented BSTs, operations modify the data structure using a single localized update. In some other data structures, operations that modify the data structure can be split into several localized updates that can be freely interleaved.

A particularly interesting application of our technique is to implement *relaxed-balance* versions of sequential data structures efficiently. Relaxed-balance data structures decouple updates that rebalance the data structure from operations, and allow updates that accomplish rebalancing to be delayed and freely interleaved with other updates. For example, a chromatic tree is a relaxed-balance version of a red-black tree (RBT) which splits up the insertion or deletion of a key and any subsequent rotations into a sequence of localized updates. There is a rich literature of relaxed-balance versions of sequential data structures [22], and several papers (e.g., [24]) have described general techniques that can be used to easily produce them from large classes of existing sequential data structures. The small number of nodes involved in each update makes relaxed-balance data structures perfect candidates for efficient implementation using our technique.

Our Contributions

- We provide a simple template that can be filled in to obtain an implementation of any update for a data structure based on a down-tree. We prove that any data structure that follows our template for all of

its updates will automatically be linearizable and non-blocking. The template takes care of all process coordination, so the data structure designer is able to think of updates as atomic steps.
- To demonstrate the use of our template, we provide a complete, provably correct, non-blocking linearizable implementation of a chromatic tree [27], which is a relaxed-balanced version of a RBT. To our knowledge, this is the first provably correct, non-blocking balanced BST implemented using fine-grained synchronization. Our chromatic trees always have height $O(c + \log n)$, where n is the number of keys stored in the tree and c is the number of insertions and deletions that are in progress (Section 5.3).
- We show that sequential implementations of some queries are linearizable, even though they completely ignore concurrent updates. For example, an ordinary BST search (that works when there is no concurrency) also works in our chromatic tree. Ignoring updates makes searches very fast. We also describe how to perform successor queries in our chromatic tree, which interact properly with updates that follow our template (Section 5.5).
- We show experimentally that our Java implementation of a chromatic tree rivals, and often significantly outperforms, known highly-tuned concurrent dictionaries, over a variety of workloads, contention levels and thread counts. For example, with 128 threads, our algorithm outperforms Java's non-blocking skip-list by 13% to 156%, the lock-based AVL tree of Bronson et al. by 63% to 224%, and a RBT that uses software transactional memory (STM) by 13 to 134 times (Section 6).

2 Related Work

There are many lock-based implementations of search tree data structures. (See [1, 9] for state-of-the-art examples.) Here, we focus on implementations that do not use locks. Valois [32] sketched an implementation of non-blocking node-oriented BSTs from CAS. Fraser [17] gave a non-blocking BST using 8-word CAS, but did not provide a full proof of correctness. He also described how multi-word CAS can be implemented from single-word CAS instructions. Ellen et al. [15] gave a provably correct, non-blocking implementation of leaf-oriented BSTs directly from single-word CAS. A similar approach was used for k-ary search trees [11] and Patricia tries [28]. All three used the cooperative technique originated by Turek, Shasha and Prakash [31] and Barnes [4]. Howley and Jones [20] used a similar approach to build node-oriented BSTs. They tested their implementation using a model checker, but did not prove it correct. Natarajan and Mittal [25] give another leaf-oriented BST implementation, together with a sketch of correctness. Instead of marking nodes, it marks edges. This enables insertions to be accomplished by a single CAS, so they do not need to be helped. It also combines deletions that would otherwise conflict. All of these trees are not balanced, so the height of a tree with n keys can be $\Theta(n)$.

Tsay and Li [30] gave a general approach for implementing trees in a wait-free manner using LL and SC operations (which can, in turn be implemented from CAS, e.g., [3]). However, their technique requires every process accessing the tree (even for read-only operations such as searches) to copy an entire path of the tree starting from the root. Concurrency is severely limited, since every operation must change the root pointer. Moreover, an extra level of indirection is required for every child pointer.

Red-black trees [5, 18] are well known BSTs that have height $\Theta(\log n)$. Some attempts have been made to implement RBTs without using locks. It was observed that the approach of Tsay and Li could be used to implement wait-free RBTs [26] and, furthermore, this could be done so that only updates must copy a path; searches may simply read the path. However, the concurrency of updates is still very limited. Herlihy et al. [19] and Fraser and Harris [16] experimented on RBTs implemented using software transactional memory (STM), which only satisfied obstruction-freedom, a weaker progress property. Each insertion or deletion, together with necessary rebalancing is enclosed in a single large transaction, which can touch all nodes on a path from the root to a leaf.

Some researchers have attempted fine-grained approaches to build non-blocking balanced search trees, but they all use extremely complicated process coordination schemes. Spiegel and Reynolds [29] described a non-blocking data structure that combines elements of B-trees and skip lists. Prior to this paper, it was the leading implementation of an ordered dictionary. However, the authors provided only a brief justification of correctness. Braginsky and Petrank [8] described a B+tree implementation. Although they have posted a correctness proof, it is very long and complex.

In a balanced search tree, a process is typically responsible for restoring balance after an insertion or deletion by performing a series of rebalancing steps along the path from the root to the location where the insertion or deletion occurred. Chromatic trees, introduced by Nurmi and Soisalon-Soininen [27], decouple the updates that perform the insertion or deletion from the updates that perform the rebalancing steps. Rather than treating an insertion or deletion and its associated rebalancing steps as a single, large update, it is broken into smaller, localized updates that can be interleaved, allowing more concurrency. This decoupling originated in the work of Guibas and Sedgewick [18] and Kung and Lehman [21]. We use the leaf-oriented chromatic trees by Boyar, Fagerberg and Larsen [7]. They provide a family of local rebalancing steps which can be executed in any order, interspersed with insertions and deletions. Moreover, an amortized *constant* number of rebalancing steps per INSERT or DELETE is sufficient to restore balance for any sequence of operations. We have also used our template to implement a non-blocking version of Larsen's leaf-oriented relaxed AVL tree [23]. In such a tree, an amortized *logarithmic* number of rebalancing steps per INSERT or DELETE is sufficient to restore balance.

There is also a node-oriented relaxed AVL tree by Bougé et al. [6], in which an amortized *linear* number of rebalancing steps per INSERT or DELETE is sufficient to restore balance. Bronson et al. [9] developed a highly optimized fine-grained locking implementation of this data structure using optimistic concurrency techniques to improve search performance. Deletion of a key stored in an internal node with two children is done by simply marking the node and a later insertion of the same key can reuse the node by removing the mark. If all internal nodes are marked, the tree is essentially leaf-oriented. Crain et al. gave a different implementation using lock-based STM [12] and locks [13], in which *all* deletions are done by marking the node containing the key. Physical removal of nodes and rotations are performed by one separate thread. Consequently, the tree can become very unbalanced. Drachsler et al. [14] give another fine-grained lock-based implementation, in which deletion physically removes the node containing the key and searches are non-blocking. Each node also contains predecessor and successor pointers, so when a search ends at an incorrect leaf, sequential search can be performed to find the correct leaf. A non-blocking implementation of Bougé's tree has not appeared, but our template would make it easy to produce one.

3 LLX, SCX and VLX Primitives

The load-link extended (LLX), store-conditional extended (SCX) and validate-extended (VLX) primitives are multi-word generalizations of the well-known load-link (LL), store-conditional (SC) and validate (VL) primitives and they have been implemented from single-word CAS [10]. The benefit of using LLX, SCX and VLX to implement our template is two-fold: the template can be described quite simply, and much of the complexity of its correctness proof is encapsulated in that of LLX, SCX and VLX.

Instead of operating on single words, LLX, SCX and VLX operate on Data-records, each of which consists of a fixed number of mutable fields (which can change), and a fixed number of immutable fields (which cannot). LLX(r) attempts to take a snapshot of the mutable fields of a Data-record r. If it is concurrent with an SCX involving r, it may return FAIL, instead. Individual fields of a Data-record can also be read directly. An SCX(V, R, fld, new) takes as arguments a sequence V of Data-records, a subsequence R of V, a pointer fld to a mutable field of one Data-record in V, and a new value new for that field. The SCX tries to atomically store the value new in the field that fld points to and *finalize* each Data-record in R. Once a Data-record is finalized, its mutable fields cannot be changed by any subsequent SCX, and any LLX of the Data-record will return FINALIZED instead of a snapshot.

Before a process invokes SCX or VLX(V), it must perform an LLX(r) on each Data-record r in V. The last such LLX by the process is said to be *linked* to the SCX or VLX, and the linked LLX must return a snapshot of r (not FAIL or FINALIZED). An SCX(V, R, fld, new) by a process modifies the data structure only if each Data-record r in V has not been changed since its linked LLX(r); otherwise the SCX fails. Similarly, a VLX(V) returns TRUE only if each Data-record r in V has not been changed since its linked LLX(r) by the same process; otherwise the VLX fails. VLX can be used to obtain a snapshot of a set of Data-records. Although LLX, SCX and VLX can fail, their failures are limited in such a way that we can use them to build non-blocking data structures. See [10] for a more formal specification of these primitives.

These new primitives were designed to balance ease of use and efficient implementability using single-word CAS. The implementation of the primitives from CAS in [10] is more efficient if the user of the primitives can guarantee that two constraints, which we describe next, are satisfied. The first constraint prevents the ABA problem for the CAS steps that actually perform the updates.

Constraint 1: Each invocation of SCX(V, R, fld, new) tries to change fld to a value new that it never previously contained.

The implementation of SCX does something akin to locking the elements of V in the order they are given. Livelock can be easily avoided by requiring all V sequences to be sorted according to some total order on Data-records. However, this ordering is necessary only to guarantee that SCXs continue to succeed. Therefore, as long as SCXs are still succeeding in an execution, it does not matter how V sequences are ordered. This observation leads to the following constraint, which is much weaker.

Constraint 2: Consider each execution that contains a configuration C after which the value of no field of any Data-record changes. There is a total order of all Data-records created during this execution such that, for every SCX whose linked LLXs begin after C, the V sequence passed to the SCX is sorted according to the total order.

It is easy to satisfy these two constraints using standard approaches, e.g., by attaching a version number to each field, and sorting V sequences by any total order, respectively. However, we shall see that Constraints 1 and 2 are *automatically* satisfied in a natural way when LLX and SCX are used according to our tree update template.

Under these constraints, the implementation of LLX, SCX, and VLX in [10] guarantees that there is a linearization of all SCXs that modify the data structure (which may include SCXs that do not terminate because a process crashed, but *not* any SCXs that fail), and all LLXs and VLXs that return, but do not fail.

We assume there is a Data-record *entry* which acts as the entry point to the data structure and is never deleted. This Data-record points to the root of a down-tree. We represent an empty down-tree by a pointer to an empty Data-record. A Data-record is *in the tree* if it can be reached by following pointers from *entry*. A Data-record r is *removed from the tree* by an SCX if r is in the tree immediately prior to the linearization point of the SCX and is not in the tree immediately afterwards. Data structures produced using our template *automatically* satisfy one additional constraint:

Constraint 3: A Data-record is finalized when (and only when) it is removed from the tree.

Under this additional constraint, the implementation of LLX and SCX in [10] also guarantees the following three properties.

- If LLX(r) returns a snapshot, then r is in the tree just before the LLX is linearized.
- If an SCX(V, R, fld, new) is linearized and new is (a pointer to) a Data-record, then this Data-record is in the tree immediately after the SCX is linearized.
- If an operation reaches a Data-record r by following pointers read from other Data-records, starting from *entry*, then r was in the tree at some earlier time during the operation.

These properties are useful for proving the correctness of our template. In the following, we sometimes abuse notation by treating the sequences V and R as sets, in which case we mean the set of all Data-records in the sequence.

The memory overhead introduced by the implementation of LLX and SCX is fairly low. Each node in the tree is augmented with a pointer to a descriptor and a bit. Every node that has had one of its child pointers changed by an SCX points to a descriptor. (Other nodes have a NIL pointer.) A descriptor can be implemented to use only three machine words after the update it describes has finished. The implementation of LLX and SCX in [10] assumes garbage collection, and we do the same in this work. This assumption can be eliminated by using, for example, the new efficient memory reclamation scheme of Aghazadeh et al. [2].

4 Tree Update Template

Our tree update template implements updates that atomically replace an old connected subgraph in a down-tree by a new connected subgraph. Such an update can implement any change to the tree, such as an

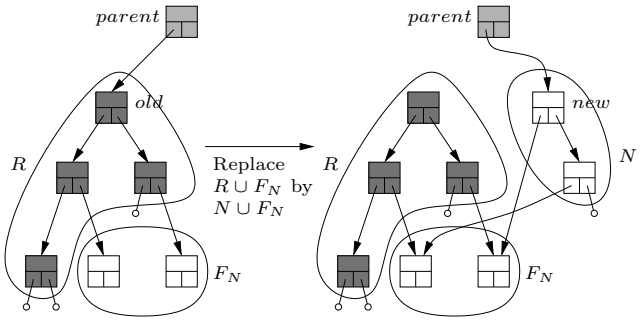


Fig. 1. Example of the tree update template. R is the set of nodes to be removed, N is a tree of new nodes that have never before appeared in the tree, and F_N is the set of children of N (and of R). Nodes in F_N may have children. The shaded nodes (and possibly others) are in the sequence V of the SCX that performs the update. The darkly shaded nodes are finalized by the SCX.

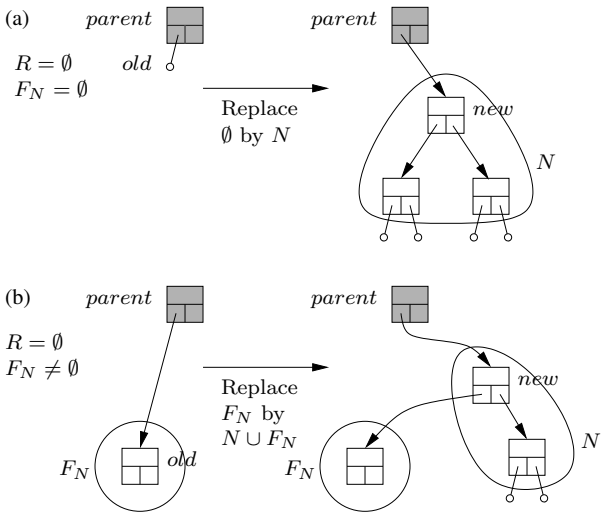


Fig. 2. Examples of two special cases of the tree update template when no nodes are removed from the tree. (a) Replacing a NIL child pointer: In this case, $R = F_N = \emptyset$. (b) Inserting new nodes in the middle of the tree: In this case, $R = \emptyset$ and F_N consists of a single node.

insertion into a BST or a rotation used to rebalance a RBT. The old subgraph includes all nodes with a field (including a child pointer) to be modified. The new subgraph may have pointers to nodes in the old tree. Since every node in a down-tree has indegree one, the update can be performed by changing a single child pointer of some node *parent*. (See Figure 1.) However, problems could arise if a concurrent operation changes the part of the tree being updated. For example, nodes in the old subgraph, or even *parent*, could be removed from the tree before *parent*'s child pointer is changed. Our template takes care of the process coordination required to prevent such problems.

Each tree node is represented by a Data-record with a fixed number of child pointers as its mutable fields (but different nodes may have different numbers of child fields). Each child pointer points to a Data-record or contains NIL (denoted by $\multimap$ in our figures). For simplicity, we assume that any other data in the node is stored in immutable fields. Thus, if an update must change some of this data, it makes a new copy of the node with the updated data.

Our template for performing an update to the tree is fairly simple: An update first performs LLXs on nodes in a contiguous portion of the tree, including *parent* and the set R of nodes to be removed from the

```
1  TEMPLATE(args)
2    follow zero or more pointers from entry to reach a node n_0
3    i := 0
4    loop
5      s_i := LLX(n_i)
6      if s_i ∈ {FAIL, FINALIZED} then return FAIL
7      s'_i := immutable fields of n_i
8      exit loop when CONDITION(s_0, s'_0, ..., s_i, s'_i, args)
           ▷ CONDITION must eventually return TRUE
9      n_{i+1} := NEXTNODE(s_0, s'_0, ..., s_i, s'_i, args)
           ▷ returns a non-NIL child pointer from one of s_0, ..., s_i
10     i := i + 1
11   end loop
12   if SCX(SCX-ARGUMENTS(s_0, s'_0, ..., s_i, s'_i, args)) then return RESULT(s_0, s'_0, ..., s_i, s'_i, args)
13   else return FAIL
```

Fig. 3. Tree update template. CONDITION, NEXTNODE, SCX-ARGUMENTS and RESULT can be filled in with any locally computable functions, provided that SCX-ARGUMENTS satisfies postconditions PC1 to PC8.

tree. Then, it performs an SCX that atomically changes the child pointer as shown in Figure 1 and finalizes nodes in R. Figure 2 shows two special cases where R is empty. An update that performs this sequence of steps is said to *follow* the template.

We now describe the tree update template in more detail. An update UP($args$) that follows the template shown in Figure 3 takes any arguments, $args$, that are needed to perform the update. UP first reads a sequence of child pointers starting from *entry* to reach some node n_0. Then, UP performs LLXs on a sequence $\sigma = \langle n_0, n_1, \ldots \rangle$ of nodes starting with n_0. For maximal flexibility of the template, the sequence σ can be constructed on-the-fly, as LLXs are performed. Thus, UP chooses a non-NIL child of one of the previous nodes to be the next node of σ by performing some deterministic local computation (denoted by NEXTNODE in Figure 3) using any information that is available locally, namely, the snapshots of mutable fields returned by LLXs on the previous elements of σ, values read from immutable fields of previous elements of σ, and $args$. (This flexibility can be used, for example, to avoid unnecessary LLXs when deciding how to rebalance a BST.) UP performs another local computation (denoted by CONDITION in Figure 3) to decide whether more LLXs should be performed. To avoid infinite loops, this function must eventually return TRUE in any execution of UP. If any LLX in the sequence returns FAIL or FINALIZED, UP also returns FAIL, to indicate that the attempted update has been aborted because of a concurrent update on an overlapping portion of the tree. If all of the LLXs successfully return snapshots, UP invokes SCX and returns a result calculated locally by the RESULT function (or FAIL if the SCX fails).

UP applies the function SCX-ARGUMENTS to use locally available information to construct the arguments V, R, fld and new for the SCX. The postconditions that must be satisfied by SCX-ARGUMENTS are somewhat technical, but intuitively, they are meant to ensure that the arguments produced describe an update as shown in Figure 1 or Figure 2. The update must remove a connected set R of nodes from the tree and replace it by a connected set N of newly-created nodes that is rooted at new by changing the child pointer stored in fld to point to new. In order for this change to occur atomically, we include R and the node containing fld in V. This ensures that if any of these nodes has changed since it was last accessed by one of UP's LLXs, the SCX will fail. The sequence V may also include any other nodes in σ.

More formally, we require SCX-ARGUMENTS to satisfy nine postconditions. The first three are basic requirements of SCX.

PC1: V is a subsequence of σ.
PC2: R is a subsequence of V.
PC3: The node *parent* containing the mutable field fld is in V.

Let G_N be the directed graph $(N \cup F_N, E_N)$, where E_N is the set of all child pointers of nodes in N when they are initialized, and $F_N = \{y : y \notin N \text{ and } (x, y) \in E_N \text{ for some } x \in N\}$. Let *old* be the value read from fld by the LLX on *parent*.

PC4: G_N is a non-empty down-tree rooted at *new*.

PC5: If $old = \text{NIL}$ then $R = \emptyset$ and $F_N = \emptyset$.

PC6: If $R = \emptyset$ and $old \neq \text{NIL}$, then $F_N = \{old\}$.

PC7: UP allocates memory for all nodes in N, including *new*.

Postcondition PC7 requires *new* to be a newly-created node, in order to satisfy Constraint 1. There is no loss of generality in using this approach: If we wish to change a child y of node x to NIL (to chop off the entire subtree rooted at y) or to a descendant of y (to splice out a portion of the tree), then, instead, we can replace x by a new copy of x with an updated child pointer. Likewise, if we want to delete the entire tree, then *entry* can be changed to point to a new, empty Data-record.

The next postcondition is used to guarantee Constraint 2, which is used to prove progress.

PC8: The sequences V constructed by all updates that take place entirely during a period of time when no SCXs change the tree structure must be ordered consistently according to a fixed tree traversal algorithm (for example, an in-order traversal or a breadth-first traversal).

Stating the remaining postcondition formally requires some care, since the tree may be changing while UP performs its LLXs. If $R \neq \emptyset$, let G_R be the directed graph $(R \cup F_R, E_R)$, where E_R is the union of the sets of edges representing child pointers read from each $r \in R$ when it was last accessed by one of UP's LLXs and $F_R = \{y : y \notin R \text{ and } (x, y) \in E_R \text{ for some } x \in R\}$. G_R represents UP's view of the nodes in R according to its LLXs, and F_R is the *fringe* of G_R. If other processes do not change the tree while UP is being performed, then F_R contains the nodes that should remain in the tree, but whose parents will be removed and replaced. Therefore, we must ensure that the nodes in F_R are reachable from nodes in N (so they are not accidentally removed from the tree). Let G_σ be the directed graph $(\sigma \cup F_\sigma, E_\sigma)$, where E_σ is the union of the sets of edges representing child pointers read from each $r \in \sigma$ when it was last accessed by one of UP's LLXs and $F_\sigma = \{y : y \notin \sigma \text{ and } (x, y) \in E_\sigma \text{ for some } x \in \sigma\}$. Since G_σ, G_R and G_N are not affected by concurrent updates, the following postcondition can be proved using purely sequential reasoning, ignoring the possibility that concurrent updates could modify the tree during UP.

PC9: If G_σ is a down-tree and $R \neq \emptyset$, then G_R is a non-empty down-tree rooted at *old* and $F_N = F_R$.

4.1 Correctness and Progress

For brevity, we only sketch the main ideas of the proof here. The full proof appears in Appendix B. Consider a data structure in which all updates follow the tree update template and SCX-ARGUMENTS satisfies postconditions PC1 to PC9. We prove, by induction on the sequence of steps in an execution, that the data structure is always a tree, each call to LLX and SCX satisfies its preconditions, Constraints 1 to 3 are satisfied, and each successful SCX atomically replaces a connected subgraph containing nodes $R \cup F_N$ with another connected subgraph containing nodes $N \cup F_N$, finalizing and removing the nodes in R from the tree, and adding the new nodes in N to the tree. We also prove no node in the tree is finalized, every removed node is finalized, and removed nodes are never reinserted.

We linearize each update UP that follows the template and performs an SCX that modifies the data structure at the linearization point of its SCX. We prove the following correctness properties.

C1: If UP were performed atomically at its linearization point, then it would perform LLXs on the same nodes, and these LLXs would return the same values.

This implies that UP's SCX-ARGUMENTS and RESULT computations must be the same as they would be if UP were performed atomically at its linearization point, so we obtain the following.

C2: If UP were performed atomically at its linearization point, then it would perform the same SCX (with the same arguments) and return the same value.

Additionally, a property is proved in [10] that allows some query operations to be performed very efficiently using only READs, for example, GET in Section 5.

C3: If a process p follows child pointers starting from a node in the tree at time t and reaches a node r at time $t' \geq t$, then r *was* in the tree at some time between t and t'. Furthermore, if p reads v from

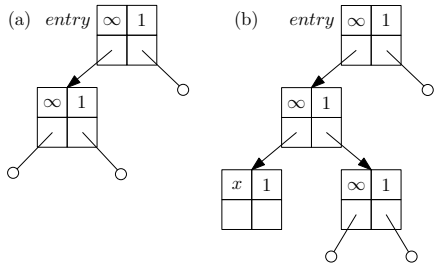


Fig. 4. (a) empty tree, (b) non-empty tree.

a mutable field of r at time $t'' \geq t'$ then, at some time between t and t'', node r was in the tree and this field contained v.

The following properties, which come from [10], can be used to prove non-blocking progress of queries.

P1: If LLXs are performed infinitely often, then they return snapshots or FINALIZED infinitely often.

P2: If VLXs are performed infinitely often, and SCXs are not performed infinitely often, then VLXs return TRUE infinitely often.

Each update that follows the template is wait-free. Since updates can fail, we also prove the following progress property.

P3: If updates that follow the template are performed infinitely often, then updates succeed infinitely often.

A successful update performs an SCX that modifies the tree. Thus, it is necessary to show that SCXs succeed infinitely often. Before an invocation of SCX(V, R, fld, new) can succeed, it must perform an LLX(r) that returns a snapshot, for each $r \in V$. Even if P1 is satisfied, it is possible for LLXs to always return FINALIZED, preventing any SCXs from being performed. We prove that any algorithm whose updates follow the template automatically guarantees that, for each Data-record r, each process performs at most one invocation of LLX(r) that returns FINALIZED. We use this fact to prove P3.

5 Application: Chromatic Trees

Here, we show how the tree update template can be used to implement an ordered dictionary ADT using chromatic trees. Due to space restrictions, we only sketch the algorithm and its correctness proof. All details of the implementation and its correctness proof appear in Appendix C. The ordered dictionary stores a set of keys, each with an associated value, where the keys are drawn from a totally ordered universe. The dictionary supports five operations. If *key* is in the dictionary, GET(*key*) returns its associated value. Otherwise, GET(*key*) returns $\perp$. SUCCESSOR(*key*) returns the smallest key in the dictionary that is larger than *key* (and its associated value), or $\perp$ if no key in the dictionary is larger than *key*. PREDECESSOR(*key*) is analogous. INSERT(*key*, *value*) replaces the value associated with *key* by *value* and returns the previously associated value, or $\perp$ if *key* was not in the dictionary. If the dictionary contains *key*, DELETE(*key*) removes it and returns the value that was associated immediately beforehand. Otherwise, DELETE(*key*) simply returns $\perp$.

A RBT is a BST in which the root and all leaves are coloured black, and every other node is coloured either red or black, subject to the constraints that no red node has a red parent, and the number of black nodes on a path from the root to a leaf is the same for all leaves. These properties guarantee that the height of a RBT is logarithmic in the number of nodes it contains. We consider search trees that are leaf-oriented, meaning the dictionary keys are stored in the leaves, and internal nodes store keys that are used only to direct searches towards the correct leaf. In this context, the BST property says that, for each node x, all descendants of x's left child have keys less than x's key and all descendants of x's right child have keys that are greater than *or equal to* x's key.

To decouple rebalancing steps from insertions and deletions, so that each is localized, and rebalancing steps can be interleaved with insertions and deletions, it is necessary to relax the balance properties of RBTs. A *chromatic tree* [27] is a relaxed-balance RBT in which colours are replaced by non-negative integer

weights, where weight zero corresponds to red and weight one corresponds to black. As in RBTs, the sum of the weights on each path from the root to a leaf is the same. However, RBT properties can be violated in the following two ways. First, a red child node may have a red parent, in which case we say that a *red-red violation* occurs at this child. Second, a node may have weight $w > 1$, in which case we say that $w - 1$ *overweight violations* occur at this node. The root always has weight one, so no violation can occur at the root.

To avoid special cases when the chromatic tree is empty, we add sentinel nodes at the top of the tree (see Figure 10). The sentinel nodes and *entry* have key ∞ to avoid special cases for SEARCH, INSERT and DELETE, and weight one to avoid special cases for rebalancing steps. Without having a special case for INSERT, we automatically get the two sentinel nodes in Figure 10(b), which also eliminate special cases for DELETE. The chromatic tree is rooted at the leftmost grandchild of *entry*. The sum of weights is the same for all paths from the root of the chromatic tree to its leaves, but not for paths that include *entry* or the sentinel nodes.

Rebalancing steps are localized updates to a chromatic tree that are performed at the location of a violation. Their goal is to eventually eliminate all red-red and overweight violations, while maintaining the invariant that the tree is a chromatic tree. If no rebalancing step can be applied to a chromatic tree (or, equivalently, the chromatic tree contains no violations), then it is a RBT. We use the set of rebalancing steps of Boyar, Fagerberg and Larsen [7], which have a number of desirable properties: No rebalancing step increases the number of violations in the tree, rebalancing steps can be performed in any order, and, after sufficiently many rebalancing steps, the tree will always become a RBT. Furthermore, in any sequence of insertions, deletions and rebalancing steps starting from an empty chromatic tree, the amortized number of rebalancing steps is at most three per insertion and one per deletion.

5.1 Implementation

We represent each node by a Data-record with two mutable child pointers, and immutable fields k, v and w that contain the node's key, associated value, and weight, respectively. The child pointers of a leaf are always NIL, and the value field of an internal node is always NIL.

GET, INSERT and DELETE each execute an auxiliary procedure, SEARCH(key), which starts at *entry* and traverses nodes as in an ordinary BST search, using READs of child pointers until reaching a leaf, which it then returns (along with the leaf's parent and grandparent). Because of the sentinel nodes shown in Figure 10, the leaf's parent always exists, and the grandparent exists whenever the chromatic tree is non-empty. If it is empty, SEARCH returns NIL instead of the grandparent. We define the *search path* for key at any time to be the path that SEARCH(key) would follow, if it were done instantaneously. The GET(key) operation simply executes a SEARCH(key) and then returns the value found in the leaf if the leaf's key is key, or $\perp$ otherwise.

At a high level, INSERT and DELETE are quite similar to each other. INSERT(key, $value$) and DELETE(key) each perform SEARCH(key) and then make the required update at the leaf reached, in accordance with the tree update template. If the modification fails, then the operation restarts from scratch. If it succeeds, it may increase the number of violations in the tree by one, and the new violation occurs on the search path to key. If a new violation is created, then an auxiliary procedure CLEANUP is invoked to fix it before the INSERT or DELETE returns.

Detailed pseudocode for GET, SEARCH, DELETE and CLEANUP is given in Figure 12 and 13. (The implementation of INSERT is similar to that of DELETE, and its pseudocode is omitted due to lack of space.) Note that an expression of the form $P\ ?\ A : B$ evaluates to A if the predicate P evaluates to true, and B otherwise. The expression $x.y$, where x is a Data-record, denotes field y of x, and the expression $\&x.y$ represents a pointer to field y.

DELETE(key) invokes TRYDELETE to search for a leaf containing key and perform the localized update that actually deletes key and its associated value. The effect of TRYDELETE is illustrated in Figure 7. There, nodes drawn as squares are leaves, shaded nodes are in V, $\otimes$ denotes a node in R to be finalized, and $\oplus$ denotes a new node. The name of a node appears below it or to its left. The weight of a node appears to its right.

```
1  GET(key)
2     ⟨−, −, l⟩ := SEARCH(key)
3     return (key = l.k) ? l.v : NIL
4  SEARCH(key)
5     n0 := NIL; n1 := entry; n2 := entry.left
6     while n2 is internal
7        n0 := n1; n1 := n2
8        n2 := (key < n1.k) ? n1.left : n1.right
9     return ⟨n0, n1, n2⟩
10 DELETE(key)
11    do
12       result := TRYDELETE(key)
13    while result = FAIL
14    ⟨value, violation⟩ := result
15    if violation then CLEANUP(key)
16    return value
17 CLEANUP(key)
18    ▷ Eliminates the violation created by an INSERT or DELETE of key
19    while TRUE
20       ▷ Save four last nodes traversed
21       n0 := NIL; n1 := NIL; n2 := entry; n3 := entry.left
22       while TRUE
23          if n3.w > 1 or (n2.w = 0 and n3.w = 0) then
24             ▷ Found a violation at node n3
25             TRYREBALANCE(n0, n1, n2, n3)                    ▷ Try to fix it
26             exit loop                                       ▷ Go back to entry and search again
27          else if n3 is a leaf then return
28             ▷ Arrived at a leaf without finding a violation
29          if key < n3.k then
30             n0 := n1; n1 := n2; n2 := n3; n3 := n3.left
31          else n0 := n1; n1 := n2; n2 := n3; n3 := n3.right
```

Fig. 5. GET, SEARCH, DELETE and CLEANUP.

```
1  TryDelete(key)
2    ▷ If successful, returns ⟨value, violation⟩, where value is the value associated with key, or Nil if key was not
         in the dictionary, and violation indicates whether the deletion created a violation. Otherwise, Fail is
         returned.
3    ⟨n0, −, −⟩ := Search(key)
4    ▷ Special case: there is no grandparent of the leaf reached
5    if n0 = Nil then return ⟨Nil, False⟩
6    ▷ Template iteration 0 (grandparent of leaf)
7    s0 := LLX(n0)
8    if s0 ∈ {Fail, Finalized} then return Fail
9    n1 := (key < s0.left.k) ? s0.left : s0.right
10   ▷ Template iteration 1 (parent of leaf)
11   s1 := LLX(n1)
12   if s1 ∈ {Fail, Finalized} then return Fail
13   n2 := (key < s1.left.k) ? s1.left : s1.right
14   ▷ Special case: key is not in the dictionary
15   if n2.k ≠ key then return ⟨⊥, False⟩
16   ▷ Template iteration 2 (leaf)
17   s2 := LLX(n2)
18   if s2 ∈ {Fail, Finalized} then return Fail
19   n3 := (key < s1.left.k) ? s1.right : s1.left
20   ▷ Template iteration 3 (sibling of leaf)
21   s3 := LLX(n3)
22   if s3 ∈ {Fail, Finalized} then return Fail
23   ▷ Computing SCX-Arguments from locally stored values
24   w := (n1.k = ∞ or n0.k = ∞) ? 1 : n1.w + n3.w
25   new := new node with weight w, key n3.k, value n3.v, and children s3.left, s3.right
26   V := (key < s1.left.k) ? ⟨n0, n1, n2, n3⟩ : ⟨n0, n1, n3, n2⟩
27   R := (key < s1.left.k) ? ⟨n1, n2, n3⟩ : ⟨n1, n3, n2⟩
28   fld := (key < s0.left.k) ? &n0.left : &n0.right
29   if SCX(V, R, fld, new) then return ⟨n2.v, (w > 1)⟩
30   else return Fail
```

Fig. 6. TryDelete.

TryDelete first invokes Search(key) to find the grandparent, n_0, of the leaf on the search path to key. If the grandparent does not exist, then the tree is empty (and it looks like Figure 10(a)), so TryDelete returns successfully at line 5. TryDelete then performs LLX(n_0), and uses the result to obtain a pointer to the parent, n_1, of the leaf to be deleted. Next, it performs LLX(n_1), and uses the result to obtain a pointer to the leaf, n_2, to be deleted. If n_2 does not contain key, then the tree does not contain key, and TryDelete returns successfully at line 15. So, suppose that n_2 does contain key. Then TryDelete performs LLX(n_2). At line 19, TryDelete uses the result of its previous LLX(n_1) to obtain a pointer to the sibling, n_3, of the leaf to be deleted. A final LLX is then performed on n_3. Over the next few lines, TryDelete computes SCX-Arguments. Line 24 computes the weight of the node new in the depiction of Delete in Figure 10, ensuring that it has weight one if it is taking the place of a sentinel or the root of the chromatic tree. Line 25 creates new, reading the key, and value directly from n_3 (since they are immutable) and the child pointers from the result of the LLX(n_3) (since they are mutable). Next, TryDelete uses locally stored values to construct the sequences V and R that it will use for its SCX, ordering their elements according to a breadth-first traversal, in order to satisfy PC8. Finally, TryDelete invokes SCX to perform the modification. If the SCX succeeds, then TryDelete returns a pair containing the value stored in node n_2 (which is immutable) and the result of evaluating the expression $w > 1$.

Delete can create an overweight violation (but not a red-red violation), so the result of $w > 1$ indicates whether TryDelete created a violation. If any LLX returns Fail or Finalized, or the SCX fails, TryDelete simply returns Fail, and Delete invokes TryDelete again. If TryDelete creates a new violation, then Delete invokes Cleanup(key) (described in Section 5.2) to fix it before Delete returns.

A simple inspection of the pseudocode suffices to prove that SCX-Arguments satisfies postconditions PC1 to PC9. TryDelete follows the template except when it returns at line 5 or line 15. In these cases, not following the template does not impede our efforts to prove correctness or progress, since TryDelete will not modify the data structure, and returning at either of these lines will cause Delete to terminate.

We now describe how rebalancing steps are implemented from LLX and SCX, using the tree update template. As an example, we consider one of the 22 rebalancing steps, named RB2 (shown in Figure 7), which eliminates a red-red violation at node n_3. The other 21 are implemented similarly. To implement RB2, a sequence of LLXs are performed, starting with node n_0. A pointer to node n_1 is obtained from the result of LLX(n_0), pointers to nodes n_2 and f_3 are obtained from the result of LLX(n_1), and a pointer to node n_3 is obtained from the result of LLX(n_2). Since node n_3 is to be removed from the tree, an LLX is performed on it, too. If any of these LLXs returns Fail or Finalized, then this update fails. For RB2 to be applicable, n_1 and f_3 must have positive weights and n_2 and n_3 must both have weight 0. Since weight fields are immutable, they can be read any time after the pointers to n_1, f_3, n_2, and n_3 have been obtained. Next, new and its two children are created. N consists of these three nodes. Finally, SCX(V, R, fld, new) is invoked, where fld is the child pointer of n_0 that pointed to n_1 in the result of LLX(n_0).

If the SCX modifies the tree, then no node $r \in V$ has changed since the update performed LLX(r). In this case, the SCX replaces the directed graph G_R by the directed graph G_N and the nodes in R are finalized. This ensures that other updates cannot erroneously modify these old nodes after they have been replaced. The nodes in the set $F_R = F_N = \{f_0, f_1, f_2, f_3\}$ each have the same keys, weights, and child pointers before and after the rebalancing step, so they can be reused. $V = \langle n_0, n_1, n_2, n_3\rangle$ is simply the sequence of nodes on which LLX is performed, and $R = \langle n_1, n_2, n_3\rangle$ is a subsequence of V, so PC1, PC2 and PC3 are satisfied. Clearly, we satisfy PC4 and PC7 when we create new and its two children. It is easy to verify that PC5, PC6 and PC9 are satisfied. If the tree does not change during the update, then the nodes in V are ordered consistently with a breadth-first traversal of the tree. Since this is true for all updates, PC8 is satisfied.

One might wonder why f_3 is not in V, since RB2 should be applied only if n_1 has a right child with positive weight. Since weight fields are immutable, the only way that this can change after we check $f_3.w > 0$ is if the right child field of n_1 is altered. If this happens, the SCX will fail.

5.2 The rebalancing algorithm

Since rebalancing is decoupled from updating, there must be a scheme that determines when processes should perform rebalancing steps to eliminate violations. In [7], the authors suggest maintaining one or

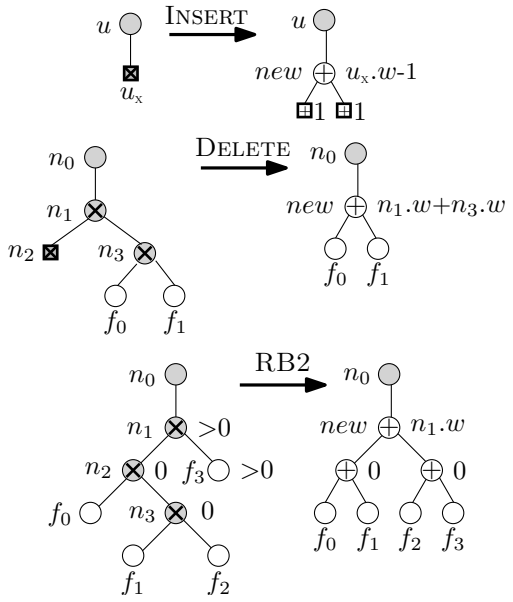


Fig. 7. Examples of chromatic tree updates.

more *problem queues* which contain pointers to nodes that contain violations, and dedicating one or more *rebalancing processes* to simply perform rebalancing steps as quickly as possible. This approach does not yield a bound on the height of the tree, since rebalancing may lag behind insertions and deletions. It is possible to obtain a height bound with a different queue based scheme, but we present a way to bound the tree's height without the (significant) overhead of maintaining any auxiliary data structures. The linchpin of our method is the following claim concerning violations.

VIOL: If a violation is on the search path to *key* before a rebalancing step, then the violation is still on the search path to *key* after the rebalancing step, or it has been eliminated.

While studying the rebalancing steps in [7], we realized that most of them satisfy VIOL. Furthermore, any time a rebalancing step would violate VIOL another rebalancing step that satisfies VIOL can be applied instead. Hence, we always choose to perform rebalancing so that each violation created by an INSERT(*key*) or DELETE(*key*) stays on the search path to *key* until it is eliminated. In our implementation, each INSERT or DELETE that increases the number of violations cleans up after itself. It does this by invoking a procedure CLEANUP(*key*), which behaves like SEARCH(*key*) until it finds the first node n_3 on the search path where a violation occurs. Then, CLEANUP(*key*) attempts to eliminate or move the violation at n_3 by invoking another procedure TRYREBALANCE which applies one localized rebalancing step at n_3, following the tree update template. (TRYREBALANCE is similar to DELETE, and pseudocode is omitted, due to lack of space.) CLEANUP(*key*) repeats these actions, searching for *key* and invoking TRYREBALANCE to perform a rebalancing step, until the search goes all the way to a leaf without finding a violation.

In order to prove that each INSERT or DELETE cleans up after itself, we must prove that while an invocation of CLEANUP(*key*) searches for *key* by reading child pointers, it does not somehow miss the violation it is responsible for eliminating, even if a concurrent rebalancing step moves the violation upward in the tree, above where CLEANUP is currently searching. To see why this is true, consider any rebalancing step that occurs while CLEANUP is searching. The rebalancing step is implemented using the tree update template, and looks like Figure 1. It takes effect at the point it changes a child pointer fld of some node *parent* from a node *old* to a node *new*. If CLEANUP reads fld while searching, we argue that it does not matter whether fld contains *old* or *new*. First, suppose the violation is at a node that is removed from the tree by the rebalancing step, or a child of such a node. If the search passes through *old*, it will definitely reach the violation, since nodes do not change after they are removed from the tree. If the search passes through *new*, VIOL implies that the rebalancing step either eliminated the violation, or moved it to a new

node that will still be on the search path through *new*. Finally, if the violation is further down in the tree, below the section modified by the concurrent rebalancing step, a search through either *old* or *new* will reach it.

Showing that TryRebalance follows the template (i.e., by defining the procedures in Figure 3) is complicated by the fact that it must decide which of the chromatic tree's 22 rebalancing steps to perform. It is more convenient to unroll the loop that performs LLXs, and write TryRebalance using conditional statements. A helpful technique is to consider each path through the conditional statements in the code, and check that the procedures Condition, NextNode, SCX-Arguments and Result can be defined to produce this single path. It is sufficient to show that this can be done for each path through the code, since it is always possible to use conditional statements to combine the procedures for each path into procedures that handle all paths.

5.3 Proving a bound on the height of the tree

Since we always perform rebalancing steps that satisfy VIOL, if we reach a leaf without finding the violation that an Insert or Delete created, then the violation has been eliminated. This allows us to prove that the number of violations in the tree at any time is bounded above by c, the number of insertions and deletions that are currently in progress. Further, since removing all violations would yield a red-black tree with height $O(\log n)$, and eliminating each violation reduces the height by at most one, the height of the chromatic tree is $O(c + \log n)$.

5.4 Correctness and Progress

As mentioned above, Get(key) invokes Search(key), which traverses a path from *entry* to a leaf by reading child pointers. Even though this search can pass through nodes that have been removed by concurrent updates, we prove by induction that every node visited *was* on the search path for key at some time during the search. Get can thus be linearized when the leaf it reaches is on the search path for key (and, hence, this leaf is the only one in the tree that could contain key).

Every Delete operation that performs an update, and every Insert operation, is linearized at the SCX that performs the update. Other Delete operations (that return at line 5 or 15) behave like queries, and are linearized in the same way as Get. Because no rebalancing step modifies the set of keys stored in leaves, the set of leaves always represents the set of dictionary entries.

The fact that our chromatic tree is non-blocking follows from P1 and the fact that at most $3i + d$ rebalancing steps can be performed after i insertions and d deletions have occurred (proved in [7]).

5.5 Successor queries

Successor(key) runs an ordinary BST search algorithm, using LLXs to read the child fields of each node visited, until it reaches a leaf. If the key of this leaf is larger than key, it is returned and the operation is linearized at any time during the operation when this leaf was on the search path for key. Otherwise, Successor finds the next leaf. To do this, it remembers the last time it followed a left child pointer and, instead, follows one right child pointer, and then left child pointers until it reaches a leaf, using LLXs to read the child fields of each node visited. If any LLX it performs returns Fail or Finalized, Successor restarts. Otherwise, it performs a validate-extended (VLX), which returns True only if all nodes on the path connecting the two leaves have not changed. If the VLX succeeds, the key of the second leaf found is returned and the query is linearized at the linearization point of the VLX. If the VLX fails, Successor restarts.

5.6 Allowing more violations

Forcing insertions and deletions to rebalance the chromatic tree after creating only a single violation can cause unnecessary rebalancing steps to be performed, for example, because an overweight violation created

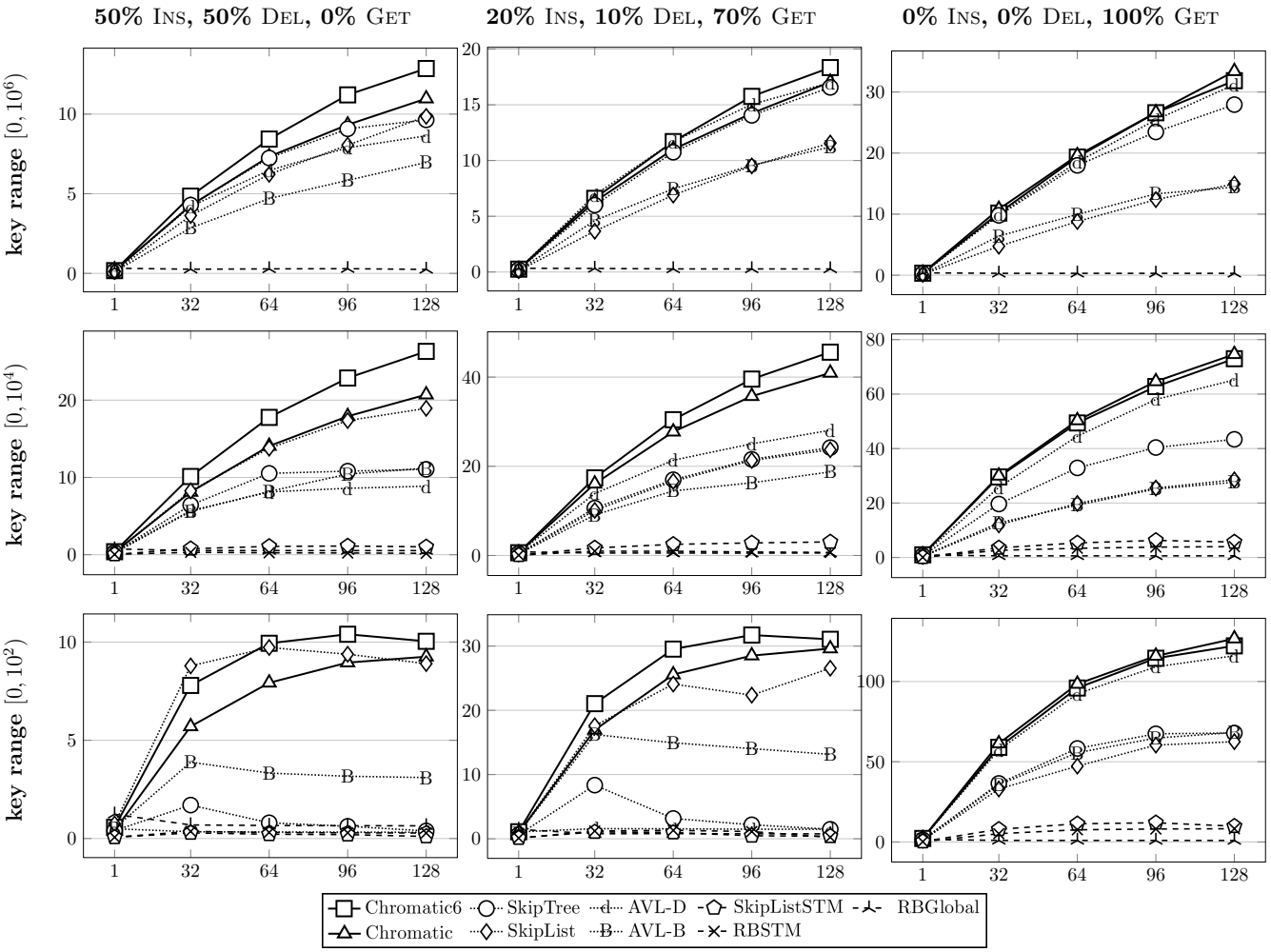


Fig. 8. *Multithreaded* throughput (millions of operations/second) for 2-socket SPARC T2+ (128 hardware threads) on y-axis versus number of threads on x-axis.

by a deletion might be eliminated by a subsequent insertion. In practice, we can reduce the total number of rebalancing steps that occur by modifying our Insert and Delete procedures so that Cleanup is invoked only once the number of violations on a path from *entry* to a leaf exceeds some constant k. The resulting data structure has height $O(k + c + \log n)$. Since searches in the chromatic tree are extremely fast, slightly increasing search costs to reduce update costs can yield significant benefits for update-heavy workloads.

6 Experimental Results

We compared the performance of our chromatic tree (Chromatic) and the variant of our chromatic tree that invokes Cleanup only when the number of violations on a path exceeds six (Chromatic6) against several leading data structures that implement ordered dictionaries: the non-blocking skip-list (SkipList) of the Java Class Library, the non-blocking multiway search tree (SkipTree) of Spiegel and Reynolds [29], the lock-based relaxed-balance AVL tree with non-blocking searches (AVL-D) of Drachsler et al. [14], and the lock-based relaxed-balance AVL tree (AVL-B) of Bronson et al. [9]. Our comparison also includes an STM-based red-black tree optimized by Oracle engineers (RBSTM) [19], an STM-based skip-list (SkipListSTM), and the highly optimized Java red-black tree, `java.util.TreeMap`, with operations protected by a global lock

(RBGlobal). The STM data structures are implemented using DeuceSTM 1.3.0, which is one of the fastest STM implementations that does not require modifications to the Java virtual machine. We used DeuceSTM's offline instrumentation capability to eliminate any STM instrumentation at running time that might skew our results. All of the implementations that we used were made publicly available by their respective authors. For a fair comparison between data structures, we made slight modifications to RBSTM and SkipListSTM to use generics, instead of hardcoding the type of keys as `int`, and to store values in addition to keys. Java code for Chromatic and Chromatic6 is available from http://implementations.tbrown.pro.

We tested the data structures for three different operation mixes, 0i-0d, 20i-10d and 50i-50d, where xi-yd denotes x% INSERTs, y% DELETEs, and $(100 - x - y)$% GETs, to represent the cases when all of the operations are queries, when a moderate proportion of the operations are INSERTs and DELETEs, and when all of the operations are INSERTs and DELETEs. We used three key ranges, $[0, 10^2), [0, 10^4)$ and $[0, 10^6)$, to test different contention levels. For example, for key range $[0, 10^2)$, data structures will be small, so updates are likely to affect overlapping parts of the data structure.

For each data structure, each operation mix, each key range, and each thread count in {1, 32, 64, 96, 128}, we ran five trials which each measured the total throughput (operations per second) of all threads for five seconds. Each trial began with an untimed prefilling phase, which continued until the data structure was within 5% of its expected size in the steady state. For operation mix 50i-50d, the expected size is half of the key range. This is because, eventually, each key in the key range has been inserted or deleted at least once, and the last operation on any key is equally likely to be an insertion (in which case it is in the data structure) or a deletion (in which case it is not in the data structure). Similarly, 20i-10d yields an expected size of two thirds of the key range since, eventually, each key has been inserted or deleted and the last operation on it is twice as likely to be an insertion as a deletion. For 0i-0d, we prefilled to half of the key range.

We used a Sun SPARC Enterprise T5240 with 32GB of RAM and two UltraSPARC T2+ processors, for a total of 16 1.2GHz cores supporting a total of 128 hardware threads. The Sun 64-bit JVM version 1.7.0_03 was run in server mode, with 3GB minimum and maximum heap sizes. Different experiments run within a single instance of a Java virtual machine (JVM) are not statistically independent, so each batch of five trials was run in its own JVM instance. Prior to running each batch, a fixed set of three trials was run to cause the Java HotSpot compiler to optimize the running code. Garbage collection was manually triggered before each trial. The heap size of 3GB was small enough that garbage collection was performed regularly (approximately ten times) in each trial. We did not pin threads to cores, since this is unlikely to occur in practice.

Figure 8 shows our experimental results. Our algorithms are drawn with solid lines. Competing hand-crafted implementations are drawn with dotted lines. Implementations with coarse-grained synchronization are drawn with dashed lines. Error bars are not drawn because they are mostly too small to see: The standard deviation is less than 2% of the mean for half of the data points, and less than 10% of the mean for 95% of the data points. The STM data structures are not included in the graphs for key range $[0, 10^6)$, because of the enormous length of time needed just to perform prefilling (more than 120 seconds per five second trial).

Chromatic6 nearly always outperforms Chromatic. The only exception is for an all query workload, where Chromatic performs slightly better. Chromatic6 is prefilled with the Chromatic6 insertion and deletion algorithms, so it has a slightly larger average leaf depth than Chromatic; this accounts for the performance difference. In every graph, Chromatic6 rivals or outperforms the other data structures, even the highly optimized implementations of SkipList and SkipTree which were crafted with the help of Doug Lea and the Java Community Process JSR-166 Expert Group. Under high contention (key range $[0, 10^2)$), Chromatic6 outperforms every competing data structure except for SkipList in case 50i-50d and AVL-D in case 0i-0d. In the former case, SkipList approaches the performance of Chromatic6 when there are many INSERTs and DELETEs, due to the simplicity of its updates. In the latter case, the non-blocking searches of AVL-D allow it to perform nearly as well as Chromatic6; this is also evident for the other two key ranges. SkipTree, AVL-D and AVL-B all experience negative scaling beyond 32 threads when there are updates. For SkipTree, this is because its nodes contain many child pointers, and processes modify a node by replacing it (severely limiting concurrency when the tree is small). For AVL-D and AVL-B, this is likely because processes waste time waiting for locks to be released when they perform updates. Under moderate contention (key range $[0, 10^4)$),

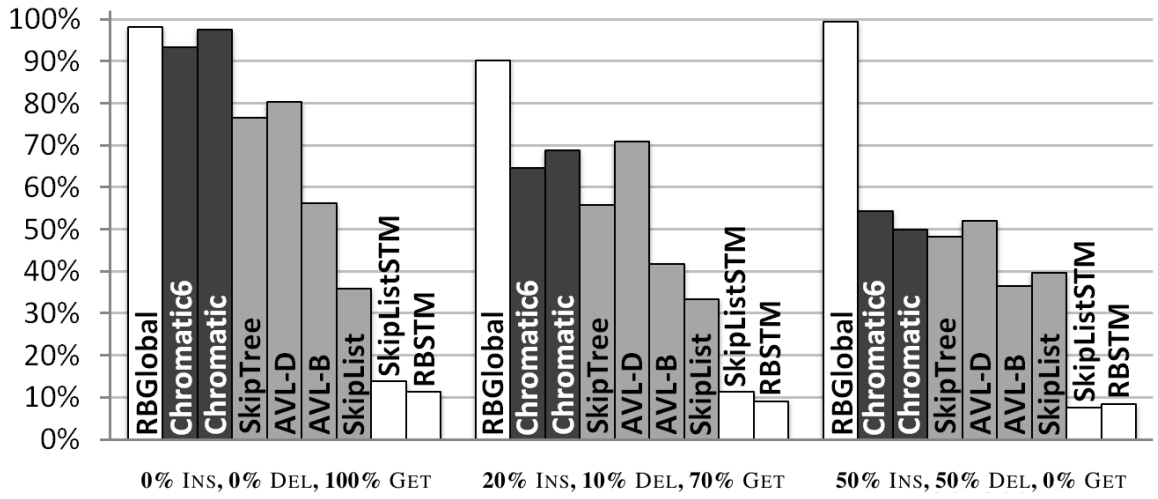


Fig. 9. *Single threaded* throughput of the data structures relative to Java's sequential RBT for key range $[0, 10^6)$.

in cases 50i-50d and 20i-10d, Chromatic6 significantly outperforms the other data structures. Under low contention, the advantages of a non-blocking approach are less pronounced, but Chromatic6 is still at the top of each graph (likely because of low overhead and searches that ignore updates).

Figure 9 compares the single-threaded performance of the data structures, relative to the performance of the sequential RBT, `java.util.TreeMap`. This demonstrates that the overhead introduced by our technique is relatively small.

Although balanced BSTs are designed to give performance guarantees for worst-case sequences of operations, the experiments are performed using random sequences. For such sequences, BSTs without rebalancing operations are balanced with high probability and, hence, will have better performance because of their lower overhead. Better experiments are needed to evaluate balanced BSTs.

7 Conclusion

In this work, we presented a template that can be used to obtain non-blocking implementations of any data structure based on a down-tree, and demonstrated its use by implementing a non-blocking chromatic tree. To the authors' knowledge, this is the first provably correct, non-blocking balanced BST with fine-grained synchronization. Proving the correctness of a direct implementation of a chromatic tree from hardware primitives would have been completely intractable. By developing our template abstraction and our chromatic tree in tandem, we were able to avoid introducing any extra overhead, so our chromatic tree is very efficient.

Given a copy of [23], and this paper, a first year undergraduate student produced our Java implementation of a relaxed-balance AVL tree in less than a week. Its performance was slightly lower than that of Chromatic. After allowing more violations on a path before rebalancing, its performance was indistinguishable from that of Chromatic6.

We hope that this work sparks interest in developing more relaxed-balance sequential versions of data structures, since it is now easy to obtain efficient concurrent implementations of them using our template.

Acknowledgements

This work was supported by NSERC. We thank Oracle for providing the machine used for our experiments.

References

1. Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. Tarjan. CBTree: A practical concurrent self-adjusting search tree. In *Proc. 26th International Symposium on Distributed Computing*, volume 7611 of *LNCS*, pages 1–15, 2012.
2. Z. Aghazadeh, W. Golab, and P. Woelfel. Brief announcement: Resettable objects and efficient memory reclamation for concurrent algorithms. In *Proc. 32nd ACM Symposium on Principles of Distributed Computing*, pages 322–324, 2013.
3. J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.
4. G. Barnes. A method for implementing lock-free data structures. In *Proc. 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.
5. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.
6. L. Bougé, J. Gabarró, X. Messeguer, and N. Schabanel. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries. Technical Report LSI-98-12-R, Universitat Politècnica de Catalunya, 1998. Available from http://www.lsi.upc.edu/dept/techreps/llistat_detallat.php?id=307.
7. J. Boyar, R. Fagerberg, and K. S. Larsen. Amortization results for chromatic search trees, with an application to priority queues. *J. Comput. Syst. Sci.*, 55(3):504–521, Dec. 1997.
8. A. Braginsky and E. Petrank. A lock-free B+tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 58–67, 2012.
9. N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming*, pages 257–268, 2010.
10. T. Brown, F. Ellen, and E. Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 13–22, 2013. Full version available from http://tbrown.pro.
11. T. Brown and J. Helga. Non-blocking k-ary search trees. In *Proc. 15th International Conf. on Principles of Distributed Systems*, volume 7109 of *LNCS*, pages 207–221, 2011.
12. T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *Proc. 17th ACM Symp. on Principles and Practice of Parallel Programming*, pages 161–170, 2012.
13. T. Crain, V. Gramoli, and M. Raynal. A contention-friendly binary search tree. In *Euro-Par*, pages 229–240, 2013.
14. D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In these proceedings, 2014.
15. F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010. Full version available as Technical Report CSE-2010-04, York University.
16. K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. on Computer Systems*, 25(2):5, 2007.
17. K. A. Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, 2003.
18. L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.
19. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proc. 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.
20. S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 161–171, 2012.
21. H. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, 1980.
22. K. S. Larsen. Amortized constant relaxed rebalancing using standard rotations. *Acta Informatica*, 35(10):859–874, 1998.
23. K. S. Larsen. AVL trees with relaxed balance. *J. Comput. Syst. Sci.*, 61(3):508–522, Dec. 2000.
24. K. S. Larsen, T. Ottmann, and E. Soisalon-Soininen. Relaxed balance for search trees with local rebalancing. *Acta Informatica*, 37(10):743–763, 2001.
25. A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In these proceedings, 2014.
26. A. Natarajan, L. Savoie, and N. Mittal. Concurrent wait-free red black trees. In *Proc. 15th International Symposium on Stabilization, Safety and Security of Distributed Systems*, volume 8255 of *LNCS*, pages 45–60, 2013.

27. O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica*, 33(6):547–557, 1996.
28. N. Shafiei. Non-blocking Patricia tries with replace operations. In *Proc. 33rd International Conference on Distributed Computing Systems*, pages 216–225, 2013.
29. M. Spiegel and P. F. Reynolds, Jr. Lock-free multiway search trees. In *Proc. 39th International Conference on Parallel Processing*, pages 604–613, 2010.
30. J.-J. Tsay and H.-C. Li. Lock-free concurrent tree structures for multiprocessor systems. In *Proc. International Conference on Parallel and Distributed Systems*, pages 544–549, 1994.
31. J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proc. 11th ACM Symposium on Principles of Database Systems*, pages 212–222, 1992.
32. J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

A Primitives

Since the definition of LLX, VLX and SCX has not yet appeared in print, we include the specification of the primitives here for completeness. However, a much fuller discussion of the primitives can be found in [10].

The VLX primitive is an extension of the usual validate primitive: it takes as an argument a sequence V of Data-records. Intuitively, a VLX(V) by a process p returns true only if no SCX has modified any record r in V since the last LLX by p.

Definition 1 *Let I' be an invocation of* SCX(V, R, fld, new) *or* VLX(V) *by a process p, and r be a Data-record in V. We say an invocation I of* LLX(r) *is* ***linked to*** *I' if and only if:*

1. *I returns a value different from* Fail *or* Finalized*, and*
2. *no invocation of* LLX(r)*,* SCX(V', R', fld', new')*, or* VLX(V')*, where V' contains r, is performed by p between I and I'.*

Before calling SCX or VLX, p must perform a linked LLX(r) on each Data-record r in V.

For every execution, there is a linearization of all successful LLXs, all successful SCXs, a subset of the non-terminating SCXs, all successful VLXs, and all reads, such that the following conditions are satisfied.

- Each read of a field f of a Data-record r returns the last value stored in f by an SCX linearized before the read (or f's initial value, if no such SCX has modified f).
- Each linearized LLX(r) that does not return Finalized returns the last value stored in each mutable field f of r by an SCX linearized before the LLX(or f's initial value, if no such SCX has modified f).
- Each linearized LLX(r) returns Finalized if and only if it is linearized after an SCX(V, R, fld, new) with r in R.
- For each linearized invocation I of SCX(V, R, fld, new) or VLX(V), and for each r in V, no SCX(V', R', fld', new') with r in V' is linearized between the LLX(r) linked to I and I.

Moreover, we have the following progress properties.

- Each terminating LLX(r) returns Finalized if it begins after the end of a successful SCX(V, R, fld, new) with r in R or after another LLX(r) has returned Finalized.
- If operations are performed infinitely often, then operations succeed infinitely often.
- If SCX and VLX operations are performed infinitely often, then SCX or VLX operations succeed infinitely often.
- If SCX operations are performed infinitely often, then SCX operations succeed infinitely often.

B Proof for Tree Update Template

B.1 Additional properties of LLX/SCX/VLX

We first import some useful definitions and properties of LLX/SCX/VLX that were proved in the appendix of the companion paper [10].

Definition 2 *A process p* ***sets up*** *an invocation of* SCX(V, R, fld, new) *by invoking* LLX(r) *for each $r \in V$, and then invoking* SCX(V, R, fld, new) *if none of these* LLX*s return* Fail *or* Finalized*.*

Definition 3 *A Data-record r is* ***in the data structure*** *in some configuration C if and only if r is reachable by following pointers from an entry point. We say r is* ***initiated*** *if it has ever been in the data structure. We say r is* ***removed (from the data structure) by*** *some step s if and only if r is in the data structure immediately before s, and r is not in the data structure immediately after s. We say r is* ***added (to the data structure) by*** *some step s if and only if r is not in the data structure immediately before s, and r is in the data structure immediately after s.*

Note that a Data-record can be removed from or added to the data structure only by a linearized invocation of SCX. The results of this section holds only if Constraint 3 (from section 3) is satisfied.

Lemma 4 [10] *If an invocation I of* LLX(r) *returns a value different from* FAIL *or* FINALIZED*, then r is in the data structure just before I is linearized.*

Lemma 5 [10] *If S is a linearized invocation of* SCX(V, R, fld, new)*, where new is a Data-record, then new is in the data structure just after S.*

Let C_1 and C_2 be configurations in the execution. We use $C_1 < C_2$ to mean that C_1 precedes C_2 in the execution. We say $C_1 \leq C_2$ precisely when $C_1 = C_2$ or $C_1 < C_2$. We denote by $[C_1, C_2]$ the set of configurations $\{C \mid C_1 \leq C \leq C_2\}$.

Lemma 6 [10] *Let $r_1, r_2, ..., r_l$ be a sequence of Data-records, where r_1 is an entry point, and $C_1, C_2, ..., C_{l-1}$ be a sequence of configurations satisfying $C_1 < C_2 < ... < C_{l-1}$. If, for each $i \in \{1, 2, ..., l-1\}$, a field of r_i points to r_{i+1} in configuration C_i, then r_{i+1} is in the data structure in some configuration in $[C_1, C_i]$. Additionally, if a mutable field f of r_l contains a value v in some configuration C_l after C_{l-1} then, in some configuration in $[C_1, C_l]$, r_l is in the data structure and f contains v.*

B.2 Correctness of the tree update template

In this section, we refer to an operation that follows the tree update template simply as a *tree update operation*. The results of this section apply only if the following constraint is satisfied.

Constraint 7 *Every invocation of* SCX *is performed by a tree update operation.*

In the proofs that follow, we sometimes argue about when invocations of LLX, SCX and VLX are linearized. However, the arguments we make do not require any knowledge of the linearization points chosen by any particular implementation of these primitives. We do this because the behavior of LLX, SCX and VLX is defined in terms of when operations are linearized, relative to one another. Similarly, we frequently refer to *linearized* invocations of SCX, rather than *successful* invocations, because it is possible for a non-terminating invocation of SCX to modify the data structure, and we linearize such invocations.

Since we refer to the preconditions of LLX and SCX in the following, we reproduce them here.

- LLX(r): r has been initiated
- SCX(V, R, fld, new):
  1. for each $r \in V$, p has performed an invocation I_r of LLX(r) linked to this SCX
  2. new is not the initial value of fld
  3. for each $r \in V$, no SCX(V', R', fld, new) was linearized before I_r was linearized

The following lemma establishes Constraint 3, and some other properties that will be useful when proving linearizability.

Lemma 8 *The following properties hold in any execution of tree update operations.*

1. *Let S be a linearized invocation of* SCX(V, R, fld, new)*, and G be the directed graph induced by the edges read by the* LLX*s linked to S. G is a sub-graph of the data structure at all times after the last* LLX *linked to S and before S is linearized, and no node in the N set of S is in the data structure before S is linearized.*
2. *Every invocation of* LLX *or* SCX *performed by a tree update operation has valid arguments, and satisfies its preconditions.*
3. *Let S be a linearized invocation of* SCX(V, R, fld, new)*, where fld is a field of parent, and old is the value read from fld by the* LLX$(parent)$ *linked to S. S changes fld from old to new, replacing a connected subgraph containing nodes $R \cup F_N$ with another connected subgraph containing nodes $N \cup F_N$. Further, the Data-records added by S are precisely those in N, and the Data-records removed by S are precisely those in R.*
4. *At all times, root is the root of a tree of Node-records. (We interpret $\perp$ as the empty tree.)*

Proof. We prove these claims by induction on the sequence of steps taken in the execution. Clearly, these claims hold initially. Suppose they hold before some step s. We prove they hold after s. Let O be the operation that performs s.

Proof of Claim 1. To affect this claim, s must be a linearized invocation of SCX(V, R, fld, new). Since s is linearized, the semantics of SCX imply that, for each $r \in V$, no SCX(V', R', fld', new') with $r \in V'$ is linearized between the linearization point of the invocation I of LLX(r) linked to s and the linearization point of s. Thus, for each $r \in V$, no mutable field of r changes between when I and s are linearized. We now show that all nodes and edges of G are in the data structure at all times after the last LLX linked to s is linearized, and before s is linearized. Fix any arbitrary $r \in V$. By inductive Claim 2, I satisfies its precondition, so r was initiated when I started and, hence, was in the data structure before I was linearized. By the semantics of SCX, since I returns a value different from FAIL or FINALIZED, no invocation of SCX(V'', R'', fld'', new'') with $r \in R''$ is linearized before I is linearized. By inductive Claim 3, Constraint 3 is satisfied at all times before s. Thus, r is not removed before I. Since s is linearized, no invocation of SCX(V'', R'', fld'', new'') with $r \in R''$ is linearized between the linearization points of I and s. (If such an invocation were to occur then, since r would also be in V'', the semantics of SCX would imply that s could not be linearized.) Since the linearization point of I is before that of s, r is not removed before s is linearized. When I is linearized, since r is in the data structure, all of its children are also in the data structure. Since no mutable field of r changes between the linearization points of I and s, all of r's children read by I are in the data structure throughout this time. Thus, each node and edge in G is in the data structure at all times after the last LLX linked to s, and before s.

Finally, we prove that no node in N is in the data structure before s is linearized. Since O follows the tree update template, s is its only modification to shared memory. Since each $r' \in N$ is newly created by O, it is clear that r' can only be in the data structure after s is linearized.

Proof of Claim 2. Suppose s is invocation of LLX(r). Then, $r \neq$ NIL (by the discussion in Sec. 4). By the code in Figure 3, either $r = top$, or r was obtained from the return value of some invocation L of LLX(r') previously performed by O. If r was obtained from the return value of L, then Lemma 4 implies that r' is in the data structure when L is linearized. Hence, r is in the data structure when L is linearized, which implies that r is initiated when s occurs. Now, suppose $r = top$. By the precondition of O, r was reached by following child pointers from *root* since the last operation by p. By inductive Claim 3, Constraint 3 is satisfied at all times before s. Therefore, we can apply Lemma 6, which implies that r was in the data structure at some point before the start of O (and, hence, before s). By Definition 3, r is initiated when s begins.

Suppose s is an invocation of SCX(V, R, fld, new). By Condition C3, fld is a mutable (child) field of some node $parent \in V$. By Condition C2, R is a subsequence of V. Therefore, the arguments to s are valid. By Condition C1 and the definition of σ, for each $r \in V$, O performs an invocation I of LLX(r) before s that returns a value different from FAIL or FINALIZED (and, hence, is linked to s), so s satisfies Precondition 1 of SCX.

We now prove that s satisfies SCX Precondition 2. Let $parent.c_i$ be the field pointed to by fld. If $parent.c_i$ initially contains $\perp$ then, by Condition C4, new is a Node-record, and we are done. Suppose $parent.c_i$ initially points to some Node-record r. We argued in the previous paragraph that O performs an LLX(*parent*) linked to s before s is linearized. By inductive Claim 3, Constraint 3 is satisfied at all times before s. Therefore, we can apply Lemma 6 to show that r was in the data structure at some point before the start of O (and, hence, before s). However, by inductive Claim 1 (which we have proved for s), new cannot be initiated before s, so $new \neq r$.

Finally, we show s satisfies SCX Precondition 3. Fix any $r' \in V$, and let L be the LLX(r') linked to s performed by O. To derive a contradiction, suppose an invocation S' of SCX(V', R', fld, new) is linearized before L (which is before s). By Lemma 5 (which we can apply since Constraint 3 is satisfied at all times before s), new would be in the data structure (and, hence, initiated) before s is linearized. However, this contradicts our argument that new cannot be initiated before s occurs.

Proof of Claim 3 and Claim 4. To affect these claims, s must be a linearized invocation of SCX(V, R, fld, new). Let t be when s is linearized. The semantics of SCX and the fact that s is linearized imply that, for each $r \in V$, no SCX(V', R', fld', new') with $r \in V'$ is linearized after the invocation I of LLX(r)

linked to s is linearized and before t. Thus, O satisfies tree update template Condition C5, C9, C4 and C6. By inductive Claim 1 (which we have proved for s), all nodes and edges in G are in the data structure just before t, and no node in N is in the data structure before t. Let $parent.c_i$ be the mutable (child) field changed by s, and *old* be the value read from $parent.c_i$ by the LLX(*parent*) linked to s.

Suppose $R \neq \emptyset$ (as in Fig. 1). Then, by Condition C9, G_R is a tree rooted at *old* and $F_N = F_R$. Since G_R is a sub-graph of G, inductive Claim 1 implies that each node and edge of G_R is in the data structure just before t. Further, since O performs an LLX(r) linked to s for each $r \in R$, and no child pointer changes between this LLX and time t, G_R contains every node that was a child of a node in R just before t. Thus, F_R contains every node $r \notin R$ that was a child of a node in R just before t. This implies that, just before t, for each node $r \notin R$ in the sub-tree rooted at *old*, F_R contains r or an ancestor of r. By inductive Claim 4, just before t, every path from *root* to a descendent of *old* passes through *old*. Therefore, just before t, every path from *root* to a node in $\{\text{descendents of } old\} - R$ passes through a node in F_R. Just before t, by the definition of F_R, and the fact that the nodes in R form a tree, $R \cap F_R$ is empty and no node in R is a descendent of a node in F. By Condition C4, G_N is a non-empty tree rooted at *new* with node set $N \cup F_N = N \cup F_R$, where N contains nodes that have not been in the data structure before t. Since $parent.c_i$ is the only field changed by s, s replaces a connected sub-graph with node set $R \cup F_R$ by a connected sub-graph with node set $N \cup F_R$. We prove that *parent* was in the data structure just before t. Since s modifies $parent.c_i$, just before t, *parent* must not have been finalized. Thus, no SCX(V', R', fld', new') with $parent \in R'$ can be linearized before t. By inductive Claim 3, Constraint 3 is satisfied at all times before t, so *parent* cannot be removed from the data structure before t. By inductive Claim 2, the precondition of the LLX(*parent*) linked to s implies that *parent* was initiated, so *parent* was in the data structure just before t. Since no node in N is in the data structure before t, the Data-records added by s are precisely those in N. Since, just before t, no node in R is in F, or a descendent of a node in F, and every $r \in \{\text{descendents of } old\} - R$ is reachable from a node in F_R, the Data-records removed by s are precisely those in R.

Now, to prove Claim 4, we need only show that $parent.c_i$ is the root of a sub-tree just after t. We have argued that *old* is the root of G_R just before t. Since $parent.c_i$ points to *old* just before t, the inductive hypothesis implies that *old* is the root of a subtree, and *parent* is not a descendent of *old*. Therefore, just before t, *parent* is not in any sub-tree rooted at a node in F_R. This implies that no descendent of *old* is changed by s. By inductive Claim 4, each $r \in F$ is the root of a sub-tree just before t, so each $r \in F$ is the root of a sub-tree just after t. Finally, since Condition C4 states that G_N is a non-empty down-tree rooted at *new*, and we have argued that G_N has node set $N \cup F_R$, *parent* is the root of a sub-tree just after t.

The two other cases, where $R = \emptyset$ and $old = \text{NIL}$ (as in Fig. 2(a)), and where $R = \emptyset$ and $old \neq \text{NIL}$ (as in Fig. 2(b)), are similar (and substantially easier to prove).

Observation 9 *Constraint 3 is implied by Lemma 8.3.*

We call a tree update operation **successful** if it performs a linearized invocation of SCX (which either returns TRUE, or does not terminate). We linearize each successful tree update operation at its linearized invocation of SCX. Clearly, each successful tree update operation is linearized during that operation.

Theorem 10 *At every time t, the tree T rooted at root is the same as the tree T_L that would result from the atomic execution of all tree update operations linearized up until t, at their linearization points. Moreover, the return value of each tree update operation is the same as it would be if it were performed atomically at its linearization point.*

Proof. The steps that affect T are linearized invocations of SCX. The steps that affect T_L are successful tree update operations, each of which is linearized at the linearized invocation of SCX that it performs. Thus, the steps that affect T and T_L are the same. We prove this claim by induction on the sequence of linearized invocations of SCX. Clearly, the claim holds before any linearized invocation of SCX has occurred. We suppose it holds just before some linearized invocation S of SCX(V, R, fld, new), and prove it holds just after S. Consider the tree update operation $OP(top, args)$ that performs S. Let O_L be the tree update operation $OP(top, args)$ in the linearized execution that occurs at S. By the inductive hypothesis, before S, $T = T_L$. To show that $T = T_L$ holds just after S, we must show that O and O_L perform

invocations of SCX with exactly the same arguments. In order for the arguments of these SCXs to be equal, the SCX-ARGUMENTS$(s_0, ..., s_i, args)$ computations performed by O and O_L must have the same inputs. Similarly, in order for the return values of O and O_L to be equal, their RESULT$(s_0, ..., s_i, args)$ computations must have the same inputs. Observe that O and O_L have the same $args$, and $s_0, ..., s_i$ consist of the return values of the LLXs performed by the operation, and the immutable fields of the nodes passed to these LLXs. Therefore, it suffices to prove that the arguments and return values of the LLXs performed by O and O_L are the same.

We prove that each LLX performed by O returns the same value as it would if it were performed atomically at S (when O_L is atomically performed). Since S is linearized, for each LLX(r) performed by O, no invocation of SCX(V', R', fld', new') with $r \in V'$ is linearized between when this LLX is linearized and when S is linearized. Thus, the LLX(*parent*) performed by O returns the same result that it would if it were performed atomically when S occurs.

Let I^k and I_L^k be the kth LLXs by O and O_L, respectively, a^k and a_L^k be the respective arguments to I^k and I_L^k, and v^k and v_L^k be the respective return values of I^k and I_L^k. We prove by induction that $a^k = a_L^k$ and $v^k = v_L^k$ for all $k \geq 1$.

Base case: Since O and O_L have the same *top* argument, $a^1 = a_L^1 = top$. Since each LLX performed by O returns the same value as it would if it were performed atomically at S, $v^1 = v_L^1$.

Inductive step: Suppose the inductive hypothesis holds for $k-1$ ($k > 1$). The NEXTNODE computation from which O obtains a^k depends only on $v^1, ..., v^{k-1}$ and the immutable fields of nodes $a^1, ..., a^{k-1}$. Similarly, the NEXTNODE computation from which O_L obtains a_L^k depends only on $v_L^1, ..., v_L^{k-1}$ and the immutable fields of nodes $a_L^1, ..., a_L^{k-1}$. Thus, by the inductive hypothesis, $a^k = a_L^k$. Since each LLX performed by O returns the same value as it would if it were performed atomically at S, $v^k = v_L^k$. Therefore, the inductive hypothesis holds for k, and the claim is proved.

Lemma 11 *After a Data-record r is removed from the data structure, it cannot be added back into the data structure.*

Proof. Suppose r is removed from the data structure. The only thing that can add r back into the data structure is a linearized invocation S of SCX(V, R, fld, new). By Constraint 7, such an SCX must occur in a tree update operation O. By Lemma 8.3, every Data-record added by S is in O's N set. By Lemma 8.1, no node in N is in the data structure at any time before S is linearized. Thus, r cannot be added to the data structure by S.

Recall that a process p *sets up* an invocation S of SCX(V, R, fld, new) if it performs a LLX(r) for each $r \in V$, which each return a snapshot, and then invokes S. In the companion paper [10], we make a general assumption that there is a bound on the number of times that any process will perform an LLX(r) that returns FINALIZED, for any Data-record r. Under this assumption, we prove that if a process sets up invocations of SCX infinitely often, then invocations of SCX will succeed infinitely often. We would like to use this fact to prove progress. Thus, we must show that our algorithm respects this assumption.

Lemma 12 *No process performs more than one invocation of* LLX(r) *that returns* FINALIZED*, for any r, during tree update operations.*

Proof. Fix any Data-record r. Suppose, to derive a contradiction, that a process p performs two different invocations L and L' of LLX(r) that return FINALIZED, during tree update operations. Then, since each tree update operation returns FAIL immediately after performing an LLX(r) that returns FINALIZED, L and L' must occur in different tree update operations O and O'. Without loss of generality, suppose O occurs before O'. Since L returns FINALIZED, it occurs after an invocation S of SCX(σ, R, fld, new) with $r \in R$ (by the semantics of SCX). By Lemma 8.3, r is removed from the data structure by S. By Lemma 11, r cannot be added back into the data structure. Thus, r is not in the data structure at any time after S occurs. By the precondition of the tree update template, the argument *top* to O' was obtained by following child pointers from the *root* entry point since O. By Observation 9, Constraint 3 is satisfied. Thus, Lemma 6 implies that *top* is in the data structure at some point between O and O'. This implies that $r \neq top$. Since O' performs L',

and $r \neq top$, O' must have obtained r from the return value of an invocation of LLX(r') that O' performed before L'. By Lemma 4, r' must be in the data structure just before L'' is linearized. Since L'' returns r, r' also points to r just before L'' is linearized. Therefore, r is in the data structure just before L'', which contradicts the fact that r is not in the data structure at any time after S occurs.

Lemma 13 *Tree update operations are wait-free.*

Proof. A tree update operation consists of a loop in which LLX, CONDITION and NEXTNODE are invoked (see Fig. 3), an invocation of SCX, and some other constant, local work. The implementations of LLX and SCX given in [10] are wait-free. Similarly, NEXTNODE and CONDITION must perform finite computation, and CONDITION must eventually return TRUE in every tree update operation, causing the operation to exit the loop.

Theorem 14 *If tree update operations are performed infinitely often, then tree update operations succeed infinitely often.*

Proof. Suppose, to derive a contradiction, that tree update operations are performed infinitely often but, after some time t_0, no tree update operation succeeds. Then, after some time $t_1 \geq t_0$, no tree update operation is successful. Since a tree update operation is successful if and only if it performs a linearized SCX, no invocation of SCX is linearized after t_1. Therefore, after t_1, the data structure does not change, so only a finite number of Data-records are ever initiated in the execution. By Lemma 12, after some time $t_2 \geq t_1$, no invocation of LLX performed by a tree update operation will return FINALIZED. Consider any tree update operation O (see Figure 3). First, LLXs are performed on a sequence of Data-records. If these LLXs all return values different from FAIL or FINALIZED, then an invocation of SCX is performed. If this invocation of SCX is successful, then O will be successful. Since tree update operations are performed infinitely often after t_2, Definition 2 implies that invocations of SCX are set up infinitely often. Thus, invocations of SCX succeed infinitely often. Finally, Constraint 7 implies that tree update operations succeed infinitely often, which is a contradiction.

C Chromatic search trees

C.1 Pseudocode

To avoid special cases when applying operations near the root, we add sentinel nodes with a special key ∞ that is larger than any key in the dictionary. When the dictionary is empty, it looks like Fig. 10(a). When non-empty, it looks like Fig. 10(b), with all dictionary keys stored in a chromatic tree rooted at x. The nodes shown in Fig. 10 always have weight one.

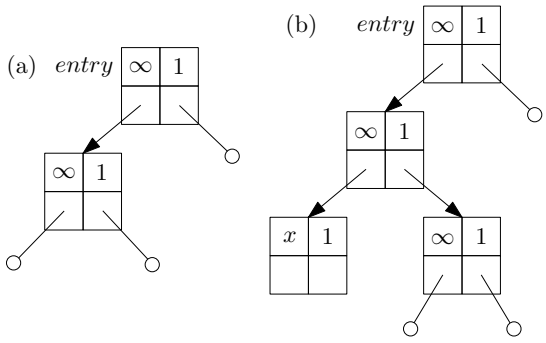


Fig. 10. (a) initial tree; (b) non-empty tree. Weights are on the right.

Pseudocode for the dictionary operations on chromatic search trees are given in Figure 12 and 13. As described in Section 5, a search for a key simply uses reads of child pointers to locate a leaf, as in an ordinary (non-concurrent) BST. INSERT and DELETE search for the location to perform the update and then call TRYINSERT or TRYDELETE, which each follow the tree update template to swing a pointer that accomplishes the update. This is repeated until the update is successful, and the update then calls CLEANUP, if necessary, to remove the violation the update created. The tree transformations that accomplish the updates are the first three shown in Figure 11. INSERT1 adds a key that was not previously present in the tree; one of the two new leaves contains the new key, and the other contains the key (and value) that were stored in u_x. INSERT2 updates the value

associated with a key that is already present in the tree by creating a new copy of the leaf containing the key. The DELETE transformation deletes the key stored in the left child of u_{x}. (There is a symmetric version for deleting keys stored in the right child of a node.)

The CLEANUP repeatedly searches for the key that was inserted or deleted, fixing any violation it encounters, until it can traverse the entire path from *root* to a leaf without seeing any violations. When a violation is found, it calls TRYREBALANCE, which uses the decision tree given in Figure 14 to decide which rebalancing step should be applied to fix the violation. At each node of the decision tree, the process decides which child to proceed to by looking at the weight of one node, as indicated in the child. The leaves of the decision tree are labelled by the rebalancing operation to apply. The code to implement the decision tree is given in Figure 15 and 16. Note that this decision tree is a component of the sequential chromatic tree algorithm that was left to the implementer in [7].

The rebalancing steps, which are shown in Figure 11, are a slight modification of those in [7].[1] Each also has a symmetric mirror-image version, denoted by an S after the name, except BLK, which is its own mirror image. We use a simple naming scheme for the nodes in the diagram. Consider the node u_{x}. We denote its left child by u_{xL}, and its right child by u_{xR}. Similarly, we denote the left child of u_{xL} by u_{xLL}, and so on. (The subscript X indicates that we do not care whether it is a left or right child.) For each transformation shown in Figure 11, the transformation is achieved by an SCX that swings a child pointer of u and depends on LLXs of all of the shaded nodes. The nodes marked with $\times$ are finalized (and removed from the data structure). The nodes marked by a + are newly created nodes. The nodes with no marking may be internal nodes, leaves or NIL. Weights of all newly created nodes are shown. The keys stored in newly created nodes are the same as in the removed nodes (so that an in-order traversal encounters them in the same order). Figure 17 implements one of the rebalancing steps. The others can be generated from their diagrams in a similar way.

The SUCCESSOR(*key*) function uses an ordinary BST search for *key* to find a leaf. If this leaf's key is bigger than *key*, it is returned. Otherwise, SUCCESSOR finds the leaf that would be reached next by an in-order traversal of the BST and then performs a VLX that verifies the path connecting the two leaves has not changed. The PREDECESSOR function can be implemented similarly.

C.2 Correctness of chromatic trees

The following lemma proves that TRYINSERT, TRYDELETE and TRYREBALANCE follow the tree update template, and that SCXs are performed only by tree update operations, so that we can invoke results from Appendix B.2 to argue that the transformations in Fig. 11 are performed atomically. Our proof that TRYREBALANCE follows the tree update template is complicated slightly by the fact that its subroutine, OVERWEIGHTLEFT (or OVERWEIGHTRIGHT), can effectively follow the template for a while, and then return early at line 155 or line 166 if it sees a NIL child pointer. (We later prove this can happen only if a tree update has changed a node since we last performed LLX on it, so that our SCX would be unsuccessful, anyway.) We thus divide the invocations of TRYREBALANCE into those that follow the template, and those that follow the template until just before they return (at one of these two lines), and show that the latter do not invoke SCX (which we must show because they are not technically tree update operations). We prove these claims together inductively with an invariant that the top of the tree is as shown in Fig. 10.

Lemma 15 *Our implementation of a chromatic search tree satisfies the following.*

1. TRYINSERT *and* TRYDELETE *follow the tree update template and satisfy all constraints specified by the template. If an invocation of* TRYREBALANCE *does not return at line 155 or line 166, then it follows the tree update template and satisfies all constraints specified by the template. Otherwise, it follows the tree update template up until it returns without performing an* SCX*, and it satisfies all constraints specified by the template.*

[1] Specifically, we do not allow W1, W2, W3 or W4 to be applied when the node labeled u_{x} has weight 0. Under this restriction, this set of rebalancing steps has the desirable property that when a violation moves, it remains on the search path to the key whose insertion or deletion originally caused the violation. It is easy to verify that an alternative rebalancing step can always be performed when $u_{\text{x}}.w = 0$, so this modification does not affect the chromatic tree's convergence to a RBT.

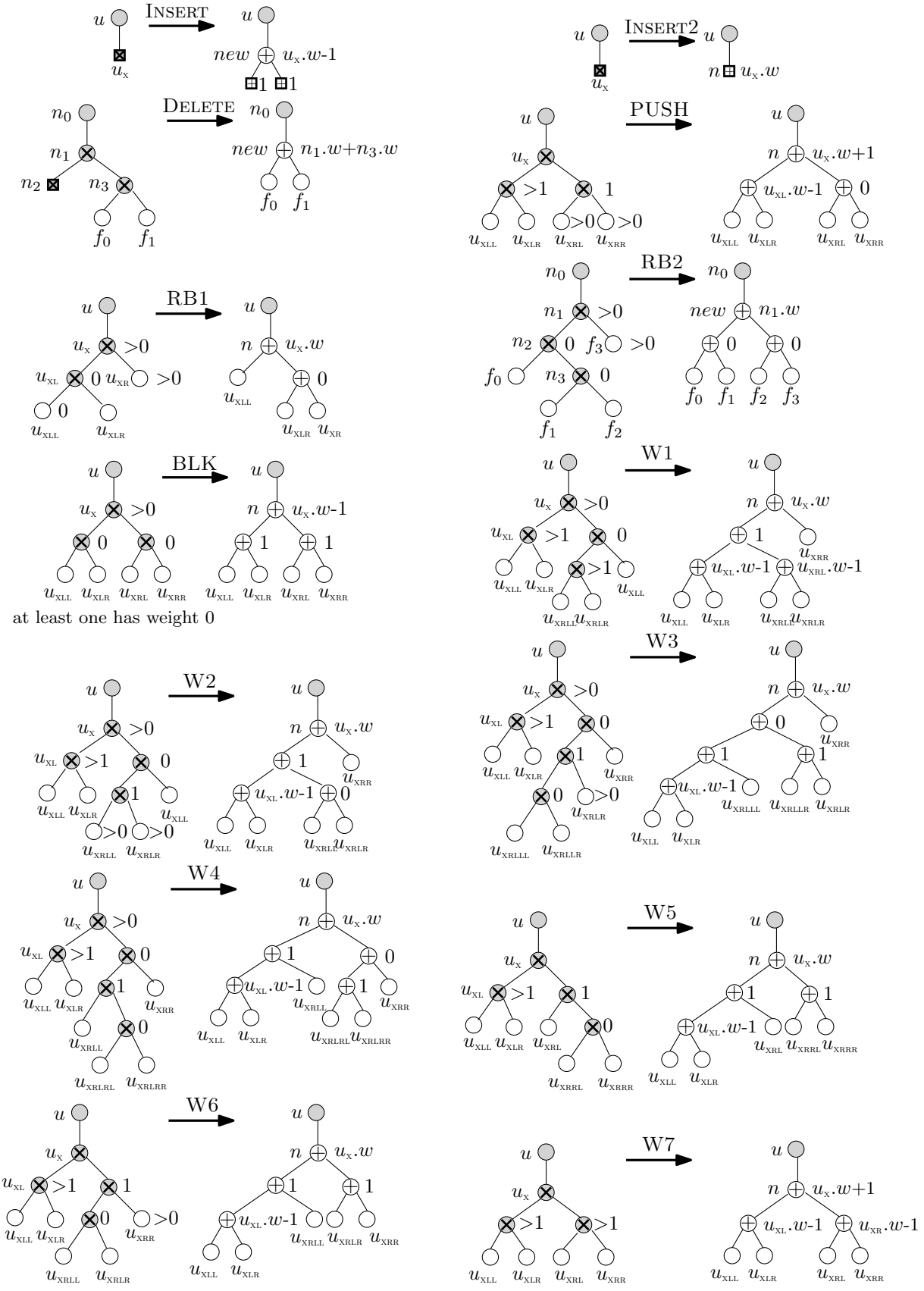


Fig. 11. Transformations for chromatic search trees. Each transformation also has a mirror image.

```
type Node-record
    ▷ User-defined fields
    left, right     ▷ child pointers (mutable)
    k, v, w         ▷ key, value, weight (immutable)
    ▷ Fields used by LLX/SCX algorithm
    info            ▷ pointer to SCX-record
    marked          ▷ Boolean
```

```
1  GET(key)
2    ▷ Returns the value associated with key, or ⊥ if no value is associated with key
3    do a standard BST search for key using reads, ending at a leaf l
4    if l.k = key then return l.v
5    else return ⊥
```

```
6  INSERT(key, value)
7    ▷ Associates value with key in the dictionary and returns the old associated value, or ⊥ if none existed
8    do
9      do a standard BST search for key using reads, ending at a leaf l with parent p
10     result := TRYINSERT(p, l, key, value)
11   while result = FAIL
12   ⟨createdViolation, value⟩ := result
13   if createdViolation then CLEANUP(key)
14   return value
```

```
15 TRYINSERT(p, l, key, value)
16   ▷ Returns ⟨TRUE, ⊥⟩ if key was not in the dictionary and inserting it caused a violation,
         ⟨FALSE, ⊥⟩ if key was not in the dictionary and inserting it did not cause a violation,
         ⟨FALSE, oldValue⟩ if ⟨key, oldValue⟩ was in the dictionary, and FAIL if we should try again
17   if (result := LLX(p)) ∈ {FAIL, FINALIZED} then return FAIL else ⟨p_L, p_R⟩ := result
18   if p_L = l then ptr := &p.left
19   else if p_R = l then ptr := &p.right
20   else return FAIL
21   if LLX(l) ∈ {FAIL, FINALIZED} then return FAIL
22   newLeaf := pointer to a new Node-record⟨NIL, NIL, key, value, 1⟩
23   if l.k = key then
24     oldValue := l.v
25     new := newLeaf
26   else
27     oldValue := ⊥
28     if l is a sentinel node then newWeight := 1 else newWeight := l.w − 1
29     if key < l.k then new := pointer to a new Node-record⟨newLeaf, l, l.k, NIL, newWeight⟩
30     else new := pointer to a new Node-record⟨l, newLeaf, key, NIL, newWeight⟩
31   if SCX(⟨p, l⟩, ⟨l⟩, ptr, new) then
32     return ⟨(new.w = p.w = 0), oldValue⟩
33   else return FAIL
```

Fig. 12. Data structure, and pseudocode for GET, INSERT and TRYINSERT (which follows the tree update template).

```
34 DELETE(key)
35   ▷ Deletes key and returns its associated value, or returns ⊥ if key was not in the dictionary
36   do
37     do a standard BST search for key using reads, ending at a leaf l with parent p and grandparent gp
38     result := TRYDELETE(gp, p, l, key)
39   while result = FAIL
40   ⟨createdViolation, value⟩ := result
41   if createdViolation then CLEANUP(key)
42   return value

43 TRYDELETE(gp, p, l, key)
44   ▷ Returns ⟨TRUE, value⟩ if ⟨key, value⟩ was in the dictionary and deleting key caused a violation,
         ⟨FALSE, value⟩ if ⟨key, value⟩ was in the dictionary and deleting key did not cause a violation,
         ⟨FALSE, ⊥⟩ if key was not in the dictionary, and FAIL if we should try again
45   if l.k ≠ key then return ⟨FALSE, ⊥⟩
46   if (result := LLX(gp)) ∈ {FAIL, FINALIZED} then return FAIL else ⟨gp_L, gp_R⟩ := result
47   if gp_L = p then ptr := &gp.left
48   else if gp_R = p then ptr := &gp.right
49   else return FAIL
50   if (result := LLX(p)) ∈ {FAIL, FINALIZED} then return FAIL else ⟨p_L, p_R⟩ := result
51   if p_L = l then s := p_R
52   else if p_R = l then s := p_L
53   else return FAIL
54   if LLX(l) ∈ {FAIL, FINALIZED} then return FAIL
55   if LLX(s) ∈ {FAIL, FINALIZED} then return FAIL
56   if p is a sentinel node then newWeight := 1 else newWeight := p.w + s.w
57   if SCX(⟨gp, p, l⟩, ⟨p, l⟩, ptr, new copy of s with weight newWeight) then
58     return ⟨(newWeight > 1), l.v⟩
59   else return FAIL

60 SUCCESSOR(key)
61   ▷ Returns the successor of key and its associated value (or ⟨⊥, ⊥⟩ if there is no such successor)
62   l := root
63   loop until l is a leaf
64     if LLX(l) ∈ {FAIL, FINALIZED} then retry SUCCESSOR(key) from scratch
65     if key < l.key then
66       lastLeft := l
67       l := l.left
68       V := ⟨lastLeft⟩
69     else
70       l := l.right
71       add l to end of V
72   if lastLeft = root then return ⟨⊥, ⊥⟩                    ▷ Dictionary is empty
73   else if key < l.k then return ⟨l.k, l.v⟩
74   else                                  ▷ Find next leaf after l in in-order traversal
75     succ := lastLeft.right
76     loop until succ is a leaf
77       if LLX(succ) ∈ {FAIL, FINALIZED} then retry SUCCESSOR(key) from scratch
78       add succ to end of V
79       succ := succ.left
80     if succ.key = ∞ then result := ⟨⊥, ⊥⟩ else result := ⟨succ.k, succ.v⟩
81     if VLX(V) then return result
82     else retry SUCCESSOR(key) from scratch
```

Fig. 13. Code for DELETE, TRYDELETE (which follows the tree update template), and SUCCESSOR.

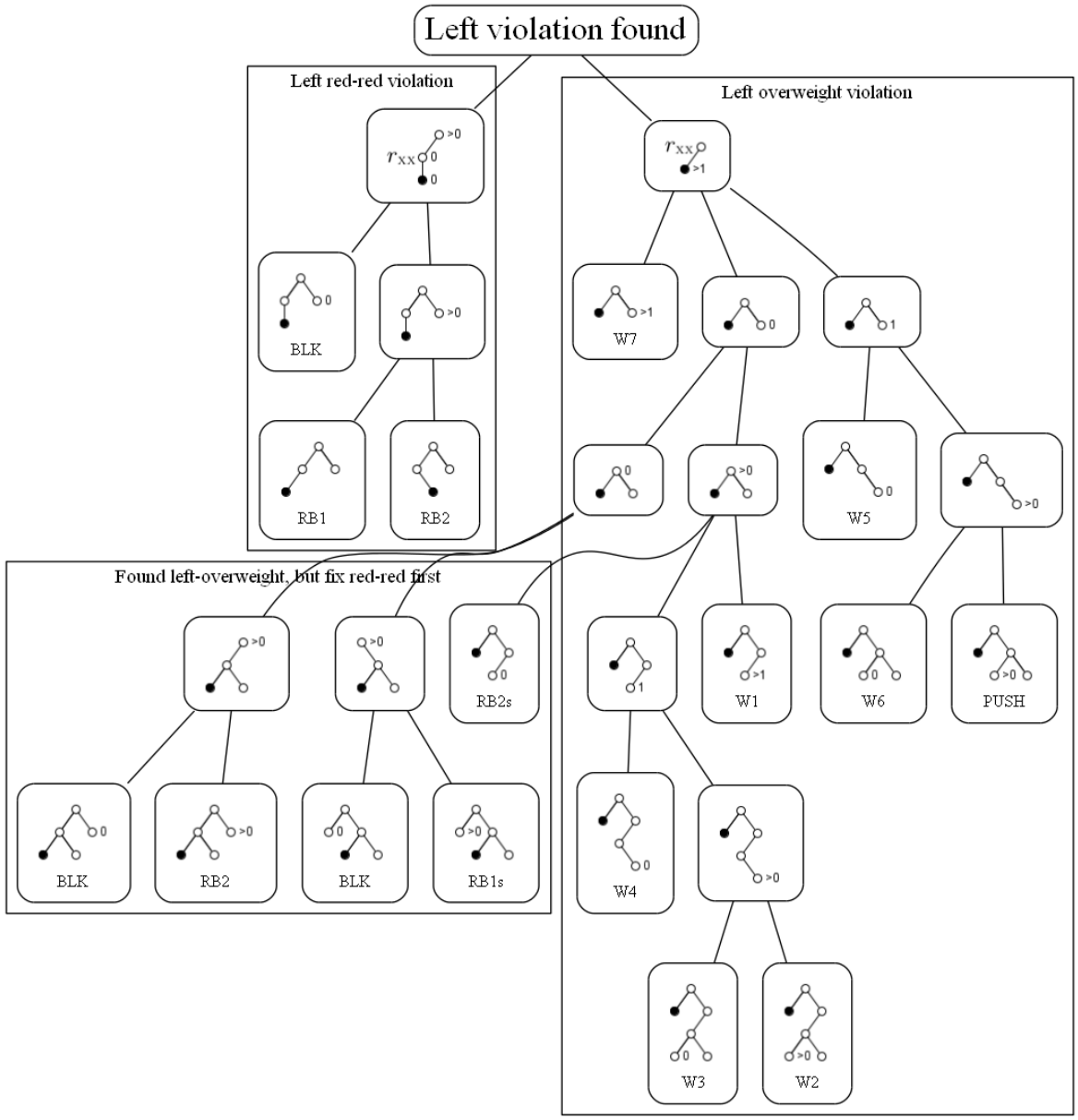


Fig. 14. Decision tree used by the algorithm to determine which rebalancing operation to apply when a violation is encountered at a node (highlighted in black). The corresponding diagram to cover right violations can be obtained by: horizontally flipping each miniature tree diagram, changing each rebalancing step DoX(·) to its symmetric version DoXs(·), and changing each symmetric rebalancing step DoXs(·) back to its original version DoX(·).

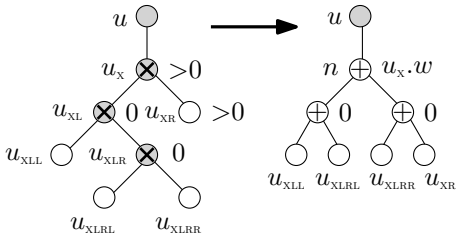
```
83  CLEANUP(key)
84    ▷ Ensures the violation created by an INSERT or DELETE of key gets eliminated
85    while TRUE
86      ggp := NIL; gp := NIL; p := NIL; l := root                ▷ Save four last nodes traversed
87      while TRUE
88        if l is a leaf then return                              ▷ Arrived at leaf without finding a violation
89        if key < l.key then {ggp := gp; gp := p; p := l; l := l.left}
90        else {ggp := gp; gp := p; p := l; l := l.right}
91        if l.w > 1 or (p.w = 0 and l.w = 0) then                ▷ Found a violation
92          TRYREBALANCE(ggp, gp, p, l)                           ▷ Try to fix it
93          exit loop                                             ▷ Go back to root and traverse again

94  TRYREBALANCE(ggp, gp, p, l)
95  ▷ Precondition: l.w > 1 or l.w = p.w = 0 ≠ gp.w
96    r := ggp
97    if (result := LLX(r)) ∈ {FAIL, FINALIZED} then return else ⟨r_L, r_R⟩ := result

99    r_X := gp
100   if r_X ∉ {r_L, r_R} then return
101   if (result := LLX(r_X)) ∈ {FAIL, FINALIZED} then return else ⟨r_XL, r_XR⟩ := result

103   r_XX := p
104   if r_XX ∉ {r_XL, r_XR} then return
105   if (result := LLX(r_XX)) ∈ {FAIL, FINALIZED} then return else ⟨r_XXL, r_XXR⟩ := result

107   if l.w > 1 then                                             ▷ Overweight violation at l
108     if l = r_XXL then                                         ▷ Left overweight violation (l is a left child)
109       if (result := LLX(r_XXL)) ∈ {FAIL, FINALIZED} then return
110       OVERWEIGHTLEFT(all r variables)
111     else if l = r_XXR then                                    ▷ Right overweight violation (l is a right child)
112       if (result := LLX(r_XXR)) ∈ {FAIL, FINALIZED} then return
113       OVERWEIGHTRIGHT(all r variables)
114   else                                                        ▷ Red-red violation at l
115     if r_XX = r_XL then                                       ▷ Left red-red violation (p is a left child)
116       if r_XR.w = 0 then
117         if (result := LLX(r_XR)) ∈ {FAIL, FINALIZED} then return else ⟨r_XRL, r_XRR⟩ := result
118         DOBLK(⟨r, r_X, r_XX, r_XR⟩, all r variables)
119       else if l = r_XXL then return DORB1(⟨r, r_X, r_XX⟩, all r variables)
120       else if l = r_XXR then
121         if (result := LLX(r_XXR)) ∈ {FAIL, FINALIZED} then return
122         DORB2(⟨r, r_X, r_XX, r_XXR⟩, all r variables)
123     else ▷ r_XX = r_XR                                        ▷ Right red-red violation (p is a right child)
124       if r_XL.w = 0 then
125         if (result := LLX(r_XL)) ∈ {FAIL, FINALIZED} then return else ⟨r_XLL, r_XLR⟩ := result
126         DOBLK(⟨r, r_X, r_XL, r_XX⟩, all r variables)
127       else if l = r_XXR then return DORB1S(⟨r, r_X, r_XX⟩, all r variables)
128       else if l = r_XXL then
129         if (result := LLX(r_XXL)) ∈ {FAIL, FINALIZED} then return
130         DORB2S(⟨r, r_X, r_XX, r_XXL⟩, all r variables)
```

Fig. 15. Pseudocode for TRYREBALANCE (which follows the tree update template) and CLEANUP.

```
131 OverweightLeft(r, r_X, r_XX, r_XXL, r_L, r_R, r_XL, r_XR, r_XXR)
132   if r_XXR.w = 0 then
133     if r_XX.w = 0 then
134       if r_XX = r_XL then
135         if r_XR.w = 0 then
136           if (result := LLX(r_XR)) ∈ {Fail, Finalized} then return else ⟨r_XRL, r_XRR⟩ := result
137           DoBLK(⟨r, r_X, r_XX, r_XR⟩, all r variables)
138         else ▷ r_XR.w > 0
139           if (result := LLX(r_XXR)) ∈ {Fail, Finalized} then return else ⟨r_XXRL, r_XXRR⟩ := result
140           DoRB2(⟨r, r_X, r_XX, r_XXR⟩, all r variables)
141       else ▷ r_XX = r_XR
142         if r_XL.w = 0 then
143           if (result := LLX(r_XL)) ∈ {Fail, Finalized} then return else ⟨r_XLL, r_XLR⟩ := result
144           DoBLK(⟨r, r_X, r_XL, r_XX⟩, all r variables)
145         else DoRB1s(⟨r, r_X, r_XX⟩, all r variables)
146     else ▷ r_XX.w > 0
147       if (result := LLX(r_XXR)) ∈ {Fail, Finalized} then return else ⟨r_XXRL, r_XXRR⟩ := result
148       if (result := LLX(r_XXRL)) ∈ {Fail, Finalized} then return
149       if r_XXRL.w > 1 then DoW1(⟨r_X, r_XX, r_XXL, r_XXR, r_XXRL⟩, result, all r variables)
150       else if r_XXRL.w = 0 then
151         ⟨r_XXRLL, r_XXRLR⟩ := result
152         DoRB2s(⟨r_X, r_XX, r_XXR, r_XXRL⟩, all r variables)
153       else ▷ r_XXRL.w = 1
154         ⟨r_XXRLL, r_XXRLR⟩ := result
155         if r_XXRLR = Nil then return                    ▷ a node we performed LLX on was modified
156         if r_XXRLR.w = 0 then
157           if (res := LLX(r_XXRLR)) ∈ {Fail, Finalized} then return else ⟨r_XXRLRL, r_XXRLRR⟩ := res
158           DoW4(⟨r_X, r_XX, r_XXL, r_XXR, r_XXRL, r_XXRLR⟩, all r variables)
159         else ▷ r_XXRLR.w > 0
160           if r_XXRLL.w = 0 then
161             if (res := LLX(r_XXRLL)) ∈ {Fail, Finalized} then return else ⟨r_XXRLLL, r_XXRLLR⟩ := res
162             DoW3(⟨r_X, r_XX, r_XXL, r_XXR, r_XXRL, r_XXRLL⟩, all r variables)
163           else DoW2(⟨r_X, r_XX, r_XXL, r_XXR, r_XXRL⟩, all r variables)
164   else if r_XXR.w = 1 then
165     if (result := LLX(r_XXR)) ∈ {Fail, Finalized} then return else ⟨r_XXRL, r_XXRR⟩ := result
166     if r_XXRR = Nil then return                         ▷ a node we performed LLX on was modified
167     if r_XXRR.w = 0 then
168       if (result := LLX(r_XXRR)) ∈ {Fail, Finalized} then return else ⟨r_XXRRL, r_XXRRR⟩ := result
169       DoW5(⟨r_X, r_XX, r_XXL, r_XXR, r_XXRR⟩, all r variables)
170     else if r_XXRL.w = 0 then
171       if (result := LLX(r_XXRL)) ∈ {Fail, Finalized} then return else ⟨r_XXRLL, r_XXRLR⟩ := result
172       DoW6(⟨r_X, r_XX, r_XXL, r_XXR, r_XXRL⟩, all r variables)
173     else DoPush(⟨r_X, r_XX, r_XXL, r_XXR⟩, all r variables)
174   else
175     if (result := LLX(r_XXR)) ∈ {Fail, Finalized} then return else ⟨r_XXRL, r_XXRR⟩ := result
176     DoW7(⟨r_X, r_XX, r_XXL, r_XXR⟩, all r variables)
```

```
177 OverweightRight(r, r_X, r_XX, r_XXR, r_L, r_R, r_XL, r_XR, r_XXL)
178 ▷ Obtained from OverweightLeft by flipping each R in the subscript of an r variable to an L (and vice versa),
      and by flipping each rebalancing step DoX(·) to its symmetric version DoXs(·) (and vice versa).
```

Fig. 16. Pseudocode for OverweightLeft.




```
179  DoRB2(u, u_x, u_xL, u_xR, u_xLL, u_xLR, u_xLRL, u_xLRR)
180      ▷ Create new nodes according to the right-hand diagram
181      create node n_L with k = u_xL.k, w = 0, left = u_xLL, right = u_xLRL
182      create node n_R with k = u_x.k, w = 0, left = u_xLRR, right = u_xR
183      create node n with k = u_xLR.k, w = u_x.w, left = n_L, right = n_R
184      ▷ Perform the SCX to swing the child pointer of node u
185      if u_x = u_L then ptr := &u.left else ptr := &u.right
186      SCX(⟨u, u_x, u_xx, u_xLR⟩, ⟨u_x, u_xL, u_xLR⟩, ptr, n)
```

Fig. 17. Implementing rebalancing step RB2. Other rebalancing steps are handled similarly using the diagrams shown in Figure 11.

2. The tree rooted at root always looks like Fig. 10(a) if it is empty, and Fig. 10(b) otherwise.

Proof. We proceed by induction on the sequence of steps in the execution.

Claim 1 follows almost immediately from inspection of the code. The only non-trivial part of the proof is showing that these algorithms never invoke LLX(r) where r = Nil, and the only step that can affect this sub-claim is an invocation of LLX. Suppose the inductive hypothesis holds just before an invocation LLX(r). For Insert1, Insert2 and Delete, $r \neq$ Nil follows from inspection of the code, inductive Claim 2, and the fact that every key inserted or deleted from the dictionary is less than ∞ (so every key inserted or deleted minimally has a parent and a grandparent). For rebalancing steps, $r \neq$ Nil follows from inspection of the code and the decision tree in Fig. 14, using a few facts about the data structure. TryRebalance performs LLXs on its arguments ggp, gp, p, l, and then on a sequence of nodes reachable from l, as it follows the decision tree. From Fig. 10(b), it is easy to see that any node with weight $w \neq 1$ minimally has a parent, grandparent, and great-grandparent. Thus, the arguments to TryRebalance are all non-Nil. By inspection of the transformations in Fig. 11, each leaf has weight $w \geq 1$, every node has zero or two children, and the child pointers of a leaf do not change. This is enough to argue that all LLXs performed by TryRebalance, and nearly all LLXs performed by OverweightLeft and OverweightRight, are passed non-Nil arguments. Without loss of generality, we restrict our attention to LLXs performed by OverweightLeft. The argument for OverweightRight is symmetric. The only LLXs that require different reasoning are performed at lines 157, 161, 168 and 171. For lines 157 and 168, the claim follows immediately from lines 155 and line 166, respectively. Consider line 161. If $r_{\text{xxrll}} =$ Nil then, since every node has zero or two children, and the child pointers of a leaf do not change, r_{xxrl} is a leaf, so $r_{\text{xxrlr}} =$ Nil. Therefore, OverweightLeft will return before it reaches line 161. By the same argument, $r_{\text{xxrl}} \neq$ Nil when line 171 is performed. Thus, $r \neq$ Nil no matter where the LLX occurs in the code.

We now prove Claim 2. The only step that can modify the tree is an SCX. Suppose the inductive hypothesis holds just before an invocation S of SCX. By inductive Claim 1, the algorithm that performed S followed the tree update template up until it performed S. Therefore, Theorem 10 implies that S atomically performs one of the transformations in Fig. 11. By inspection of these transformations, when the tree is empty, Insert1 at the left child of *root* changes the tree from looking like Fig. 10(a) to looking like Fig. 10(b) and, otherwise, does not affect the claim. When the tree has only one node with $key \neq \infty$, Delete at the leftmost grandchild of *root* changes the tree back to looking like Fig. 10(a) and, otherwise, does not affect the claim. Clearly, Insert2 does not affect the claim. Without loss of generality, let S be a left rebalancing step. Each rebalancing step in {BLK, RB1, RB2} applies only if $u_{\text{xl}}.w = 0$, and every other rebalancing step applies only if $u_{\text{xl}}.w > 1$. Therefore, S must change a child pointer of a descendent of the left child of *root* (by the inductive hypothesis). Since the child pointer changed by S was traversed while a process was searching for a key that it inserted or deleted, and ∞ is greater than any such key, S can replace only nodes with $key < \infty$.

Using the same reasoning as in the proof of Lemma 15.1, it is easy to verify that, each time the chromatic tree algorithm accesses a field of r, $r \neq$ Nil.

Definition 16 *The **search path to** key **starting at** a node r is the path that an ordinary BST search starting at r would follow. If $r = root$, then we simply call this the search path to key.*

Note that this search is well-defined even if the data structure is not a BST. Moreover, the search path starting at a node is well-defined, even if the node has been removed from the tree. In any case, we simply look at the path that an ordinary BST search would follow, if it were performed on the data structure.

The following few lemmas establish that the tree remains a BST at all times and that searches are linearizable. They are proved in a way similar to [15], although the proofs here must deal with the additional complication of rebalancing operations occuring while a search traverses the tree.

Lemma 17 *If a node v is in the data structure in some configuration C and v was on the search path for key k in some earlier configuration C', then v is on the search path for k in C.*

Proof. Since v is in the data structure in both C' and C, it must be in the data structure at all times between C' and C by Lemma 11. Consider any successful SCX S that occurs between C' and C. We show that it preserves the property that v is on the search path for k. Suppose that the SCX changes a child pointer of a node u from *old* to *new*. If v is not a descendant of *old* immediately before S, then this change cannot remove v from the search path for k. So, suppose v is a descendant of *old* immediately prior to S.

Since S does not remove v from the data structure, v must be a descendant of a node f in the fringe set F of S (see Figure 1 and Figure 2). Moreover, f must be on the search path for k before S. It is easy to check by inspection of each possible tree modification in Figure 11 that if the node $f \in F$ is on the search path for k from u prior to the modification, then it is still on the search path for k from u after the modification. So f and v are still on the search path for k after S.

Since a search only reads child pointers, and the tree may change as the search traverses the tree, we must show that it still ends up at the correct leaf. In other words, we must show that the search is linearizable even if it traverses some nodes that are no longer in the tree.

Lemma 18 *If an ordinary BST search for key k starting from the root reaches a node v, then there was some earlier configuration during the search when v was on the search path for k.*

Proof. We prove this by induction on the number of nodes visited so far by the search.

Base case: *root* is always on the search path for k.

Inductive step: Suppose that some node v that is visited by the search was on the search path for k in some configuration C between the beginning of the search and the time that the search reached v. Let v' be the next node visited by the search after v. We prove that there is a configuration C' between C and the time the search reaches v' when v' is on the search path for k. Without loss of generality, assume $k < v.key$. (The argument when $k \geq v.key$ is symmetric.) Then, when the search reaches v', v' is the left child of v. We consider two cases.

Case 1 When the search reaches v', v is in the data structure: Let C' be the configuration immediately before the search reads v' from $v.left$. Then, by Lemma 17, v is still on the search path for k in C'. Since $k < v.key$, $v' = v.left$ is also on the search path for k in C'.

Case 2 When the search reaches v', v is not in the data structure: If $v' = v.left$ at C, then v' is in the data structure at C. Otherwise, $v.left$ is changed to v' some time after C, and when that change occurs, v has not been finalized, and therefore v' is in the data structure after that change (by Constraint 3). Either way, v' is in the data structure some time at or after C.

Let C' be the last configuration at or after C when v is in the data structure. By Lemma 17, v is on the search path for k in C'. Since the only steps that can modify child pointers are successful SCXs, the next step after C' must be a successful SCX S. Since all updates to the tree satisfy Constraint 3, v must be in the R-sequence of S. Thus, v is finalized by S and its left child pointer never changes again after C'. So, in C', $v.left = v'$. Since v is on the search path for k in C' and $k < v.key$, $v.left = v'$ is also on the search path for k in C'.

Lemma 19 *At all times, the tree rooted at the left child of root is a BST.*

Proof. Initially, the left child of *root* is a leaf node, which is a BST.

Keys of nodes are immutable, and the only operation that can change child pointers are successful SCXs, so we prove that every successful SCX preserves the invariant. Each SCX atomically implements one of the changes shown in Figure 11.

The only SCX that can change a child of the *root* is Insert1 (when the tree contains no keys) and Delete (when the tree contains exactly one key). By inspection, both of these changes preserve the invariant.

So, for the remainder of the proof consider an SCX that changes a child pointer of some descendant of the left child of the *root*. By inspection, the invariant is preserved by each rebalancing step, Delete and Insert2.

It remains to consider an SCX that performs Insert1 to insert a new key k. Let u be the node whose child pointer is changed by the insertion. The node u was reached by a search for k, so u was on the search path for k at some earlier time, by Lemma 18. Since u cannot have been finalized prior to the SCX that changes its child pointer, it is still in the data structure, by Constraint 3. Thus, by Lemma 17, u is still on the search path for k when the SCX occurs. Hence, the SCX preserves the BST property.

We define the linearization points for chromatic tree operations as follows.

- Get(key) is linearized at the time during the operation when the leaf reached was on the search path for key. (This time exists, by Lemma 18.)
- An Insert is linearized at its successful execution of SCX inside TryInsert (if such an SCX exists).
- A Delete that returns $\perp$ is linearized at the time the leaf reached during the last execution of line 37 was on the search path for key. (This time exists, by Lemma 18.)
- A Delete that does not return $\perp$ is linearized at its successful execution of SCX inside TryDelete (if such an SCX exists).
- A Successor query that returns at line 72 is linearized when it performs LLX($root$).
- A Successor query that returns at line 73 at the time during the operation that l was on the search path for key. (This time exists, by Lemma 18.)
- A Successor query that returns at line 81 is linearized when it performs its successful VLX.

It is easy to verify that every operation that terminates is assigned a linearization point during the operation.

Theorem 20 *The chromatic search tree is a linearizable implementation of an ordered dictionary with the operations* Get, Insert, Delete, Successor.

Proof. Theorem 10 asserts that the SCXs implement atomic changes to the tree as shown in Figure 11. By inspection of these transformations, the set of keys and associated values stored in leaves are not altered by any rebalancing steps. Moreover, the transformations performed by each linearized Insert and Delete maintain the invariant that the set of keys and associated values stored in leaves of the tree is exactly the set that should be in the dictionary.

When a Get(key) is linearized, the search path for key ends at the leaf found by the traversal of the tree. If that leaf contains key, Get returns the associated value, which is correct. If that leaf does not contain key, then, by Lemma 19, it is nowhere else in the tree, so Get is correct to return $\perp$.

If Successor(key) returns $\langle \perp, \perp \rangle$ at line 72, then at its linearization point, the left child of the *root* is a leaf. By Lemma 15.2, the dictionary is empty.

If Successor(key) returns $\langle l.k, l.v \rangle$ at line 72, then at its linearization point, l is the leaf on the search path for key. So, l contains either key or its predecessor or successor at the linearization point. Since $key < l.k$, l is key's successor.

Finally, suppose Successor(key) returns $\langle succ.k, succ.v \rangle$ at line 81. Then l was on the search path for key at some time during the search. Since l is among the nodes validated by the VLX, it is not finalized, so it is still on the search path for key at the linearization point, by Lemma 17. Since $key \geq l.k$, the successor of key is the next leaf after l in an in-order traversal of the tree. Leaf l is the rightmost leaf in the subtree rooted at the left child of $lastLeft$ and the key returned is the leftmost leaf in the subtree rooted at the right child of $lastLeft$. The paths from $lastLeft$ to these two leaves are not finalized and therefore are in the tree. Thus, the correct result is returned.

C.3 Progress

Lemma 21 *If* TryRebalance *is invoked infinitely often, then it follows the tree update template infinitely often.*

Proof. We first prove that each invocation I' of TryRebalance that returns at line 155 or line 166 is concurrent with a tree update operation that changes the tree during I'. Suppose not, to derive a contradiction. Then, since the tree is only changed by SCXs, which are only performed by tree update operations, there is some invocation I of TryRebalance during which the tree does not change. Suppose I returns at line 155. Then, by inspection of OverweightLeft, $r_{\text{XXRL}}.w = 1$, and $r_{\text{XXRLR}} = \text{Nil}$, so r_{XXRL} is a leaf. Furthermore, since the tree does not change during I, r_{XXR} is the parent of r_{XXRL} and $r_{\text{XXR}}.w = 0$, and r_{XXL} is the sibling of r_{XXR} and $r_{\text{XXL}}.w > 1$. Therefore, the sum of weights on a path from *root* to a leaf in the sub-tree rooted at r_{XXR} is different from *root* to a leaf in the sub-tree rooted at r_{XXL}, so the tree is not a chromatic search tree, which is a contradiction. The proof when I returns at line 166 is similar.

We now prove the stated claim. To derive a contradiction, suppose TryRebalance is invoked infinitely often, but only finitely many invocations of TryRebalance follow the tree update template. Then, Lemma 15.1 implies that, after some time t, every invocation of TryRebalance returns at line 155 or line 166. Let I be an invocation of TryRebalance that returns at line 155 or line 166, and that starts after t. However, we have just argued that I must be concurrent with a tree update operation that changes the tree during I and, hence, returns at a line different from line 155 or line 166 (by inspection of the code), which is a contradiction.

Theorem 22 *The chromatic tree operations are non-blocking.*

Proof. To derive a contradiction, suppose there is some time T after which some processes continue to take steps but none complete an operation. We consider two cases.

If the operations that take steps forever do not include Insert or Delete operations, then eventually no process performs any SCXs (since the queries get, Successor and Predecessor do not perform SCXs). Thus, eventually all LLXs and VLXs succeed, and therefore all queries terminate, a contradiction.

If the operations that take steps forever include Insert or Delete operations, then they repeatedly invoke TryInsert, TryDelete or TryRebalance. By Lemma 15.1 and Lemma 21, infinitely many invocations of TryInsert, TryDelete or TryRebalance follow the tree update template. By Theorem 14, infinitely many of these calls will succeed. There is only one successful TryInsert or TryDelete performed by each process after T. So, there must be infinitely many successful calls to TryRebalance. Boyar, Fagerberg and Larsen proved [7] proved that after a bounded number of rebalancing steps, the tree becomes a RBT, and then no further rebalancing steps can be applied, a contradiction.

C.4 Bounding the chromatic tree's height

We now show that the height of the chromatic search tree is $O(\log n + c)$ where n is the number of keys stored in the tree and c is the number of incomplete Insert and Delete operations. Each Insert or Delete can create one new violation. We prove that no Insert or Delete terminates until the violation it created is destroyed. Thus, the number of violations in the tree at any time is bounded by c, and the required bound follows.

Definition 23 *Let x be a node that is in the data structure. We say that $x.w - 1$* ***overweight violations occur at*** *x if $x.w > 1$. We say that a* ***red-red violation occurs at*** *x if x and its parent in the data structure both have weight 0.*

The following lemma says that red-red violations can never be created at a node, except when the node is first added to the data structure.

Lemma 24 *Let v be a node with weight 0. Suppose that when v is added to the data structure, its (unique) parent has non-zero weight. Then v is never the child of a node with weight 0.*

Proof. Node weights are immutable. It is easy to check by inspection of each transformation in Figure 11 that if v is not a newly created node and it acquires a new parent in the transformation with weight 0, then v had a parent of weight 0 prior to the transformation.

Definition 25 *A process P is **in a cleanup phase for** k if it is executing an* INSERT(k) *or a* DELETE(k) *and it has performed a successful* SCX *inside a* TRYINSERT *or* TRYDELETE *that returns created$Violation$ =* TRUE. *If P is between line 86 and 93, location(P) and parent(P) are the values of P's local variables l and p; otherwise location(P) is the root node and parent(P) is* NIL.

We use the following invariant to show that each violation in the data structure has a pending update operation that is responsible for removing it before terminating: either that process is on the way towards the violation, or it will find another violation and restart from the top of the tree, heading towards the violation.

Lemma 26 *In every configuration, there exists an injective mapping ρ from violations to processes such that, for every violation x,*

- (A) *process $\rho(x)$ is in a cleanup phase for some key k_x and*
- (B) *x is on the search path from root for k_x and*
- (C) *either*
  (C1) *the search path for k_x from location($\rho(x)$) contains the violation x, or*
  (C2) *location($\rho(x)$).$w = 0$ and parent($\rho(x)$).$w = 0$, or*
  (C3) *in the prefix of the search path for k_x from location($\rho(x)$) up to and including the first non-finalized node (or the entire search path if all nodes are finalized), there is a node with weight greater than 1 or two nodes in a row with weight 0.*

Proof. In the initial configuration, there are no violations, so the invariant is trivially satisfied. We show that any step S by any process P preserves the invariant. We assume there is a function ρ satisfying the claim for the configuration C immediately before S and show that there is a function ρ' satisfying the claim for the configuration C' immediately after S. The only step that can cause a process to leave its cleanup phase is the termination of an INSERT or DELETE that is in its cleanup phase. The only steps that can change *location*(P) and *parent*(P) are P's execution of line 93 or the read of the child pointer on line 89 or 90. (We think of all of the updates to local variables in the braces on those lines as happening atomically with the read of the child pointer.) The only steps that can change child pointers or finalize nodes are successful SCXs. No other steps S can cause the invariant to become false.

Case 1 S is the termination of an INSERT or DELETE that is in its cleanup phase: We choose $\rho' = \rho$. S happens when the test in line 88 is true, meaning that *location*(P) is a leaf. Leaves always have weight greater than 0. The weight of the leaf cannot be greater than 1, because then the process would have exited the loop in the previous iteration after the test at line 91 returned true (since weights of nodes never change). Thus, *location*(P) is a leaf with weight 1. So, P cannot be $\rho(x)$ for any violation x, so S cannot make the invariant become false.

Case 2 S is an execution of line 93: We choose $\rho' = \rho$. Step S changes *location*(P) to *root*. If $P \neq \rho(x)$ for any violation x, then this step cannot affect the truth of the invariant. Now suppose $P = \rho(x_0)$ for some violation x_0. The truth of properties (A) and (B) are not affected by a change in *location*(P) and property (C) is not affected for any violation $x \neq x_0$. Since ρ satisfies property (B) for violation x_0 before S, it will satisfy property (C1) for x_0 after S.

Case 3 S is a read of the left child pointer on line 89: We choose $\rho' = \rho$. Step S changes *location*(P) from some node v to node v_L, which is v's left child when S is performed. If $P \neq \rho(x)$ for any violation x, then this step cannot affect the truth of the invariant. So, suppose $P = \rho(x_0)$ for some violation x_0. By (A), P is in a cleanup phase for k_{x_0}. The truth of (A) and (B) are not affected by a change in *location*(P) and property (C) is not affected for any violation $x \neq x_0$. So it remains to prove that (C) is true for violation x_0 in C'.

First, we prove $v.w \leq 1$, and hence there is never an overweight violation at v. If v is the *root*, then $v.w = 1$. Otherwise, S does not occur during the first iteration of CLEANUP's inner loop. In the previous iteration, $v.w \leq 1$ at line 91 (otherwise, the loop would have terminated).

Next, we prove that there is no red-red violation at v when S occurs. If v is the root or is not in the data structure when S occurs, then there cannot be a red-red violation at v when S occurs, by definition. Otherwise, node v was read as the child of some other node u in the previous iteration of CLEANUP's inner loop and line 91 found that $u.w \neq 0$ or $v.w \neq 0$ (otherwise the loop would have terminated). So, at some time before S (and when v was in the data structure), there was no red-red violation at v. By Lemma 24, there is no red-red violation at v when S is performed.

Next, we prove that (C2) cannot be true for x_0 in configuration C. If S is in the first iteration of CLEANUP's inner loop, then $location(P) = root$, which has weight 1. If S is not in the first iteration of CLEANUP's inner loop, then the previous iteration found $parent(P).w \neq 0$ or $location(P).w \neq 0$ (otherwise the loop would have terminated).

So we consider two cases, depending on whether (C1) or (C3) is true in configuration C.

Case 3a (C1) is true in configuration C: Thus, when S is performed, the violation x_0 is on the search path for k_{x_0} from v, but it is not at v (as argued above). S reads the *left* child of v, so $k_{x_0} < v.k$ (since the key of node v never changes). So, x_0 must be on the search path for k_{x_0} from v_L. This means (C1) is satisfied for x_0 in configuration C'.

Case 3b (C3) is true in configuration C: We argued above that $v.w \leq 1$, so the prefix must contain two nodes in a row with weight 0. If they are the first two nodes, v and v_L, then (C2) is true after S. Otherwise, (C3) is still true after S.

Case 4 S is a read of the right child pointer on line 90: The argument is symmetric to Case 3.

Case 5 S is a successful SCX: We must define the mapping ρ' for each violation x in configuration C'. By Lemma 24 and the fact that node weights are immutable, no transformation in Figure 11 can create a new violation at a node that was already in the data structure in configuration C. So, if x is at a node that was in the data structure in configuration C, x was a violation in configuration C, and $\rho(x)$ is well-defined. In this case, we let $\rho'(x) = \rho(x)$.

If x is at a node that was added to the data structure by S, then we must define $\rho(x)$ on a case-by-case basis for all transformations described in Figure 11. (The symmetric operations are handled symmetrically.)

If x is a red-red violation at a newly added node, we define $\rho'(x)$ according to the following table.

Transformation	Red-red violations x created by S	$\rho'(x)$
RB1	none created	–
RB2	none created	–
BLK	at n (if $u_{\text{x}}.w = 1$ and $u.w = 0$)	ρ(red-red violation at one of $u_{\text{xLL}}, u_{\text{xLR}}, u_{\text{xRL}}, u_{\text{xRR}}$)[2]
PUSH	none created	–
W1,W2,W3,W4	none created[3]	–
W5	at n (if $u_{\text{x}}.w = u.w = 0$)	ρ(red-red violation at u_{x})
W6	at n (if $u_{\text{x}}.w = u.w = 0$)	ρ(red-red violation at u_{x})
W7	none created	–
INSERT1	at n (if $u_{\text{x}}.w = 1$ and $u.w = 0$)	process performing the INSERT
INSERT2	none created	–
DELETE	at n (if $u_{\text{x}}.w = u_{\text{xR}}.w = u.w = 0$)	ρ(red-red violation at u_{x})

For each newly added node that has k overweight violations after S, ρ' maps them to the k distinct processes $\{\rho(q) : q \in Q\}$, where Q is given by the following table.

[1] By inspection of the decision tree in Figure 14, BLK is only applied if one of $u_{\text{xLL}}, u_{\text{xLR}}, u_{\text{xRL}}$ or u_{xRR} has weight 0, and therefore a red-red violation, in configuration C, and this red-red violation is eliminated by the transformation.

[2] By inspection of the decision tree in Figure 14, W1, W2, W3 and W4 are applied only if $u_{\text{x}}.w > 0$.

Transformation	Overweight violations created by S		Set Q of overweight violations before S
RB1	$u_{\text{x}}.w-1$ at n	(if $u_{\text{x}}.w>1$)	$u_{\text{x}}.w-1$ at u_{x}
RB2	$u_{\text{x}}.w-1$ at n	(if $u_{\text{x}}.w>1$)	$u_{\text{x}}.w-1$ at u_{x}
BLK	$u_{\text{x}}.w-2$ at n	(if $u_{\text{x}}.w>2$)	$u_{\text{x}}.w-2$ of the $u_{\text{x}}.w-1$ at u_{x}
PUSH	$u_{\text{x}}.w$ at n	(if $u_{\text{x}}.w>0$)	$u_{\text{x}}.w-1$ at u_{x}, and 1 at u_{xL}
PUSH	$u_{\text{xL}}.w-2$ at n_{L}	(if $u_{\text{xL}}.w>2$)	$u_{\text{xL}}.w-2$ of the $u_{\text{xL}}.w-1$ at u_{xL}
W1	$u_{\text{x}}.w-1$ at n	(if $u_{\text{x}}.w>1$)	$u_{\text{x}}.w-1$ at u_{x}
W1	$u_{\text{xL}}.w-2$ at n_{LL}	(if $u_{\text{xL}}.w>2$)	$u_{\text{xL}}.w-2$ of the $u_{\text{xL}}.w-1$ at u_{xL}
W1	$u_{\text{xRL}}.w-2$ at n_{LR}	(if $u_{\text{xRL}}.w>2$)	$u_{\text{xRL}}.w-2$ of the $u_{\text{xRL}}.w-1$ at u_{xL}
W2	$u_{\text{x}}.w-1$ at n	(if $u_{\text{x}}.w>1$)	$u_{\text{x}}.w-1$ at u_{x}
W2	$u_{\text{xL}}.w-2$ at n_{LL}	(if $u_{\text{xL}}.w>2$)	$u_{\text{xL}}.w-2$ of the $u_{\text{xL}}.w-1$ at u_{xL}
W3	$u_{\text{x}}.w-1$ at n	(if $u_{\text{x}}.w>1$)	$u_{\text{x}}.w-1$ at u_{x}
W3	$u_{\text{xL}}.w-2$ at n_{LLL}	(if $u_{\text{xL}}.w>2$)	$u_{\text{xL}}.w-2$ of the $u_{\text{xL}}.w-1$ at u_{xL}
W4	$u_{\text{x}}.w-1$ at n	(if $u_{\text{x}}.w>1$)	$u_{\text{x}}.w-1$ at u_{x}
W4	$u_{\text{xL}}.w-2$ at n_{LL}	(if $u_{\text{xL}}.w>2$)	$u_{\text{xL}}.w-2$ of the $u_{\text{xL}}.w-1$ at u_{xL}
W5	$u_{\text{x}}.w-1$ at n	(if $u_{\text{x}}.w>1$)	$u_{\text{x}}.w-1$ at u_{x}
W5	$u_{\text{xL}}.w-2$ at n_{LL}	(if $u_{\text{xL}}.w>2$)	$u_{\text{xL}}.w-2$ of the $u_{\text{xL}}.w-1$ at u_{xL}
W6	$u_{\text{x}}.w-1$ at n	(if $u_{\text{x}}.w>1$)	$u_{\text{x}}.w-1$ at u_{x}
W6	$u_{\text{xL}}.w-2$ at n_{LL}	(if $u_{\text{xL}}.w>2$)	$u_{\text{xL}}.w-2$ of the $u_{\text{xL}}.w-1$ at u_{xL}
W7	$u_{\text{x}}.w$ at n	(if $u_{\text{x}}.w>0$)	$u_{\text{x}}.w-1$ at u_{x}, and 1 at u_{xL}
W7	$u_{\text{xL}}.w-2$ at n_{L}	(if $u_{\text{xL}}.w>2$)	$u_{\text{xL}}.w-2$ of the $u_{\text{xL}}.w-1$ at u_{xL}
W7	$u_{\text{xR}}.w-2$ at n_{R}	(if $u_{\text{xR}}.w>2$)	$u_{\text{xR}}.w-2$ of the $u_{\text{xR}}.w-1$ at u_{xR}
INSERT1	$u_{\text{x}}.w-2$ at n	(if $u_{\text{x}}.w>2$)	$u_{\text{x}}.w-2$ of the $u_{\text{x}}.w-1$ at u_{x}
INSERT2	$u_{\text{x}}.w-1$ at n	(if $u_{\text{x}}.w>1$)	$u_{\text{x}}.w-1$ at u_{x}
DELETE	$u_{\text{x}}.w+u_{\text{xR}}.w-1$ at n	(if $u_{\text{x}}.w+u_{\text{xR}}.w>1$)	$\max(0,u_{\text{x}}.w-1)$ at u_{x} and $\max(0,u_{\text{xR}}.w-1)$ at u_{xR}[4]

The function ρ' is injective, since ρ' maps each violation created by S to a distinct process that ρ assigned to a violation that has been removed by S, with only two exceptions: for red-red violations caused by INSERT1 and one overweight violation caused by DELETE, ρ' maps the red-red violation to the process that has just begun its cleanup phase (and therefore was not assigned any violation by ρ).

Let x be any violation in the tree in configuration C'. We show that ρ' satisfies properties (A), (B) and (C) for x in configuration C'.

Property (A): Every process in the image of ρ' was either in the image of ρ or a process that just entered its cleanup phase at step S, so every process in the image of ρ' is in its cleanup phase.

Property (B) and (C): We consider several subcases.

Subcase 5a Suppose S is an INSERT1's SCX, and x is the red-red violation created by S. Then, P is in its cleanup phase for the inserted key, which is one of the children of the node containing the red-red violation x. Since the tree is a BST, x is on the search path for this key, so (B) holds.

In this subcase, $\mathit{location}(\rho'(x)) = \mathit{root}$ since $P = \rho'(x)$ has just entered its cleanup phase. So property (B) implies property (C1).

Subcase 5b Suppose S is a DELETE's SCX, and x is the overweight violation assigned to P by ρ'. Then, P is in a cleanup phase for the deleted key, which was in one of the children of u_{x} before S. Therefore, x (at the root of the newly inserted subtree) is on the search path for this key, so (B) holds.

As in the previous subcase, $\mathit{location}(\rho'(x)) = \mathit{root}$ since $P = \rho'(x)$ has just entered its cleanup phase. So property (B) implies property (C1).

Subcase 5c If x is at a node that was added to the data structure by S (and is not covered by the above two cases), then $\rho'(x)$ is $\rho(y)$ for some violation y that has been removed from the tree by S, as described in the above two tables. Let k be the key such that process $\rho(y) = \rho'(x)$ is in the cleanup phase for k. By

[4] In this case, the number of violations in Q is one too small if both $u_{\text{x}}.w$ and $u_{\text{xR}}.w$ are greater than 0, so the remaining violation is assigned to the process that performed the DELETE's SCX.

property (B), y was on the search path for k before S. It is easy to check by inspection of the tables and Figure 11 that any search path that went through y's node in configuration C goes through x's node in configuration C'. (We designed the tables to have this property.) Thus, since y was on the search path for k in configuration C, x is on the search path for k in configuration C', satisfying property (B).

If (C2) is true for violation y in configuration C, then (C2) is true for x in configuration C' (since S does not affect $location()$ or $parent()$ and $\rho(y) = \rho'(x)$). If (C3) is true for violation y in configuration C, then (C3) is true for x in configuration C' (since any node that is finalized remains finalized forever, and its child pointers do not change).

So, for the remainder of the proof of subcase 5c, suppose (C1) is true for y in configuration C. Let $l = location(\rho(y))$ in configuration C. Then y is on the search path for k from l in configuration C.

First, suppose S removes l from the data structure.

- If y is a red-red violation at node l in configuration C, then the red-red violation was already there when process $\rho(y)$ read l as the child of some other node (by Lemma 24) and (C2) is true for x in configuration C'.
- If y is an overweight violation at node l in configuration C, then it makes (C3) true for x in configuration C'.
- Otherwise, since both l and its descendant, the parent of the node that contains y, are removed by S, the entire path between these two nodes is removed from the data structure by S. So, all nodes along this path are finalized by S because Constraint 3 is satisfied. Thus, the violation y makes (C3) true for x in configuration C'.

Now, suppose S does not remove l from the data structure. In configuration C, the search path from l for k contains y. It is easy to check by inspection of the tables defining ρ' and Figure 11 that any search path from l that went through y's node in configuration C goes through x's node in configuration C'. So, (C1) is true in configuration C'.

Subcase 5d If x is at a node that was in the data structure in configuration C, then $\rho'(x) = \rho(x)$. Let k be the key such that this process is in the cleanup phase for k. Since x was on the search path for k in configuration C and S did not remove x from the data structure, x is still on the search path for k in configuration C' (by inspection of Figure 11). This establishes property (B).

If (C2) or (C3) is true for x in configuration C, then it is also true for x in configuration C', for the same reason as in Subcase 5c.

So, suppose (C1) is true for x in configuration C. Let $l = location(\rho(x))$ in configuration C. Then, (C1) says that x is on the search path for k from l in configuration C. If S does not change any of the child pointers on this path between l and x, then x is still on the search path from $location(\rho'(x)) = l$ in configuration C', so property (C1) holds for x in C'. So, suppose S does change the child pointer of some node on this path from *old* to *new*. Then the search path from l for k in configuration C goes through *old* to some node f in the Fringe set F of S and then onward to the node containing violation x. By inspection of the transformations in Figure 11, the search path for k from l in configuration C' goes through *new* to the same node f, and then onward to the node containing the violation x. Thus, property (C1) is true for x in configuration C'.

Corollary 27 *The number of violations in the data structure is bounded by the number of incomplete* Insert *and* Delete *operations.*

In the following discussion, we are discussing "pure" chromatic trees, without the dummy nodes with key ∞ that appear at the top of our tree. The sum of weights on a path from the root to a leaf of a chromatic tree is called the *path weight* of that leaf. The *height* of a node v, denoted $h(v)$ is the maximum number of nodes on a path from v to a leaf descendant of v. We also define the *weighted height* of a node v as follows.

$$wh(v) = \begin{cases} v.w & \text{if } v \text{ is a leaf} \\ \max(wh(v.left), wh(v.right)) + v.w & \text{otherwise} \end{cases}$$

Lemma 28 *Consider a chromatic tree rooted at root that contains n nodes and c violations. Suppose T is any red black tree rooted at $root_T$ that results from performing a sequence of rebalancing steps on the tree rooted at root to eliminate all violations. Then, the following claims hold.*

1. $h(root) \leq 2wh(root) + c$
2. $wh(root) \leq wh(root_T) + c$
3. $wh(root_T) \leq h(root_T)$

Proof. **Claim 1:** Consider any path from *root* to a leaf. It has at most $wh(root)$ non-red nodes. So, there can be at most $wh(root)$ red nodes that do not have red parents on the path (since *root* has weight 1). There are at most c red nodes on the path that have red parents. So the total number of nodes on the path is at most $2wh(root) + c$.

Claim 2: Consider any rebalancing step that is performed by replacing some node u_x by n (using the notation of Figure 11). If u_x is not the root of the chromatic tree, then $wh(u_x) = wh(n)$, since the path weights of all leaves in a chromatic tree must be equal. (Otherwise, the path weight to a leaf in the subtree rooted at n would become different from the path weight to a leaf outside this subtree.)

Thus, the only rebalancing steps that can change the weighted height of the root are those where u_x is the root of the tree. Recall that the weight of the root is always one. If u_x and n are supposed to have different weights according to Figure 11, then blindly setting the weight of the n to one will have the effect of changing the weighted height of the root. By inspection of Figure 11, the only transformation that increases the weighted height of the root is BLK, because it is the only transformation where the weight of n is supposed to be less than the weight of u_x. Thus, each application of BLK at the root increases the weighted height of the root by one, but also eliminates at least one red-red violation at a grandchild of the root (without introducing any new violations). Since none of the rebalancing steps increases the number of violations in the tree, performing any sequence of steps that eliminates c violations will change the weighted height of the root by at most c. The claim then follows from the fact that T is produced by eliminating c violations from the chromatic tree rooted at *root*.

Claim 3: Since T is a RBT, it contains no overweight violations. Thus, the weighted height of the tree is a sum of zeros and ones. It follows that $wh(root_T) \leq h(root_T)$.

Corollary 29 *If there are c incomplete* INSERT *and* DELETE *operations and the data structure contains n keys, then its height is $O(log\ n + c)$.*

Proof. Let *root*, T, $root_T$ be defined as in Lemma 28. We immediately obtain $h(root) \leq 2h(root_T) + 3c$ from Corollary 27 and Lemma 28. Since the height of a RBT is $O(\log n)$, it follows that the height of our data structure is $O(\log n + c)$ (including the two dummy nodes at the top of the tree with key ∞).