View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Deletion without Rebalancing in Non-Blocking Binary Search Trees*

Meng He[1] **and Mengdu Li**[2]

1 **Faculty of Computer Science, Dalhousie University, Halifax, Canada**
mhe@cs.dal.ca
2 **Dark Matter LLC, Mississauga, Canada**
meng.du.li@dal.ca

Abstract

We present a provably linearizable and lock-free relaxed AVL tree called the *non-blocking ravl tree*. At any time, the height of a non-blocking ravl tree is upper bounded by $\log_\phi(2m) + c$, where ϕ is the golden ratio, m is the total number of successful INSERT operations performed so far and c is the number of active concurrent processes that have inserted new keys and are still rebalancing the tree at this time. The most significant feature of the non-blocking ravl tree is that it does not rebalance itself after DELETE operations. Instead, it performs rebalancing only after INSERT operations. Thus, the non-blocking ravl tree is much simpler to implement than other self-balancing concurrent *binary search trees* (BSTs) which typically introduce a large number of rebalancing cases after DELETE operations, while still providing a provable non-trivial bound on its height. We further conduct experimental studies to compare our solution with other state-of-the-art concurrent BSTs using randomly generated data sequences under uniform distributions, and find that our solution achieves the best performance among concurrent self-balancing BSTs. As the keys in access sequences are likely to be partially sorted in system software, we also conduct experiments using data sequences with various degrees of presortedness to better simulate applications in practice. Our experimental results show that, when there are enough degrees of presortedness, our solution achieves the best performance among all the concurrent BSTs used in our studies, including those that perform self-balancing operations and those that do not, and thus is potentially the best candidate for many real-world applications.

1998 ACM Subject Classification E.1 [Data Structures] Distributed Data Structures

Keywords and phrases concurrent data structures, non-blocking data structures, lock-free data structures, self-balancing binary search trees, relaxed AVL trees

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.34

1 Introduction

Concurrent data structures play an important role in modern multi-core and multi-processor systems, and extensive research has been done to design data structures that support efficient concurrent operations. As BSTs are fundamental data structures, many researchers have studied the problem of designing concurrent BSTs. Lock-based BSTs are the most intuitive solutions and have been shown to be efficient [23, 1, 18, 3, 9, 7, 8]. There is, however, a potential issue: if a process holding a lock on an object is halted by the operating system, all the other processes requiring access to the same object will be prevented from making any

* This work was supported by NSERC. The work was done when the second author was in Dalhousie University.

© Meng He and Mengdu Li;
licensed under Creative Commons License CC-BY
20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 34; pp. 34:1–34:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

further progress. Non-blocking (lock-free) BSTs have thus been proposed in recent years to overcome this limitation [11, 10, 5, 24, 6, 21, 19].

Despite the extensive work on lock-based and non-blocking BSTs, only a few solutions support self-balancing [3, 7, 8, 5, 9, 18]. Self-balancing BSTs are important in both theory and practice: In theory, they have better bounds on query and update time than BSTs that do not support self-balancing; for real-world applications, studies [22] have shown that self-balancing BSTs outperform BSTs without self-rebalancing greatly in sequential settings. However, the current status of the research on concurrent BSTs is that there is much less work that provides non-trivial bounds on access time than the research on the sequential counterparts. In addition, almost all existing empirical studies [1, 3, 5, 9, 16, 19, 21, 24, 23] are performed using randomly generated data under uniform distributions. This approach, however, has some drawbacks. As mentioned in [22, 5, 1], such experimental settings favor BSTs without self-balancing greatly: the expected heights of these trees are logarithmic with high probability under these settings [15], while they avoid the costs of rebalancing. Studies [22] have also shown that it is common that in real-world applications, the keys in an access sequence are partially sorted, i.e., have some degree of presortedness, instead of being completely random. As an example, when entering student grades of a course into a database, the data are likely to be entered in the order of student IDs or names. Thus, random data do not simulate these scenarios well. Therefore, much work is needed to provide more theoretical results on concurrent self-balancing BSTs, and better designed experimental studies are also needed to evaluate their performance.

While many researchers are designing concurrent BSTs, some significant progress has also been made recently in the study of self-balancing BSTs in sequential settings. In particular, Sen and Tarjan [26] proposed a solution to address the issue that self-balancing BSTs introduce so many cases when performing rebalancing after Delete operations that many developers resort to alternative solutions. These alternative solutions, however, may have inferior performance. To provide developers with a viable self-balancing BST solution for the fast development required in industry, Sen and Tarjan came up with a relaxed AVL tree called the *ravl tree*. A ravl tree only rebalances itself after Insert operations, while its height is still bounded by $O(\log m)$, where m is the number of Insert operations performed so far. The total number of rebalancing cases in the ravl tree is incredibly few, posing a great advantage in software development.

Based on the state of the art of the research on concurrent and sequential BSTs as described above, we study the problem of designing a non-blocking self-balancing BST that only rebalances itself after Insert operations, while still providing a non-trivial provable bound on its height in terms of the total number of successful Insert operations performed so far and the number of active concurrent processes. As in sequential settings, such a solution will decrease the development time greatly in practice. Furthermore, it may even potentially improve throughput in concurrent settings: If threads performing Delete operations do not rebalance the tree after removing items, they can terminate sooner so that there are fewer concurrent threads in the system.

1.1 Our Work

We design a concurrent self-balancing BST called non-blocking ravl tree that only rebalances itself after Insert operations for asynchronous systems where shared memory locations can be accessed by multiple processes. The number of rebalancing cases introduced is much fewer than other non-blocking self-balancing BSTs such as the non-blocking chromatic tree proposed by Brown et al. [5]. More precisely, the non-blocking ravl tree only has 5 rebalancing

cases, while 22 cases have to be considered for the non-blocking chromatic tree. We prove the linearizability and progress property of a non-blocking ravl tree, and bound its height. The theoretical results of our research are summarized in the following theorem:

▶ **Theorem 1.** *The non-blocking ravl tree is linearizable and lock-free, and it only rebalances itself after* Insert *operations. For a non-blocking ravl tree built via a sequence of arbitrarily intermixed* Insert *and* Delete *operations from an empty tree, at any time during the execution, the height of the tree is bounded by* $\log_\phi(2m) + c$*, where* $\phi = \frac{1+\sqrt{5}}{2}$ *is the golden ratio,* m *is the number of* Insert *operations that have successfully inserted new keys into the tree so far and* c *is the number of* Insert *operations that have inserted a new item but not yet terminated at this time.*

We further conduct experimental studies to evaluate the performance of our solution by comparing it against other state-of-the-art concurrent BSTs. We first use randomly generated data under uniform distributions, and show that non-blocking ravl trees outperform other concurrent self-balancing BSTs. We then use sequences with different degrees of presortedness. Experimental results show that our solution achieves the best performance among all concurrent BSTs with or without self-balancing when the data sequences have enough degree of presortedness, so that the average heights of BSTs without self-balancing are approximately 4-5 times greater than the average heights of self-balancing BSTs. Considering that previous studies [22] have shown that, when implemented in system software, it is very common that BSTs without self-balancing are more than five times taller than self-balancing BSTs, we believe that our solution is the best candidate for many real-world applications.

When designing the non-blocking ravl tree, our main strategy is to apply the general approach proposed by Brown et al. [5] for developing lock-free BSTs to design a concurrent version of Sen and Tarjan's ravl tree, and thus existing techniques are borrowed. Due to the special nature of the problem studied, we develop a number of new twists on these ideas: The three rebalancing cases of the original ravl tree are no longer sufficient to cover all possibilities in concurrent settings, and thus, we consider five rebalancing cases. Furthermore, a correctness proof is provided to show that our approach indeed covers all possibilities, while no similar proofs are needed in Sen and Tarjan's work where the correctness is obvious. To bound the tree height, we still use the exponential potential function approach in Sen and Tarjan's work which was originally proposed by Haeupler et al. [17], but we develop a more complex potential function and prove more properties of rebalancing to complete the analysis. It is interesting to see that the exponential potential function approach can still be made to work despite the more complex cases in concurrent settings.

We also would like to point out that in sequential settings, ravl trees are one type of *rank-balanced trees* [17], in which balancing information called ranks are assigned to tree nodes, and by maintaining different invariants called *rank rules*, different results can be achieved. Certain rank rules can yield the classic AVL trees and red-black trees, and creative rules can be invented to design data structures with new properties, including the ravl tree and the *weak AVL* tree [17] which uses fewer rotations than previous work while ensuring that its height is never worse than a red-black tree. Our work can provide a general guideline for those who wish to design and study the concurrent versions of this new class of trees.

2 Related Work

A number of lock-based and non-blocking BSTs have been proposed. In this section, we introduce some state-of-the-art concurrent BSTs. We say that a BST is *unbalanced* if it does

not support self-balancing, and call a BST *external* if values are only stored in its leaves, while an *internal* BST uses all nodes to store values.

Lock-based search trees. Bronson et al. [3] proposed a lock-based variant of AVL trees based on *hand-over-hand optimistic validation* adapted from *Software Transactional Memory* (STM). It reduces the complexity of deleting a node with two children by simply setting the value stored in this node to *NULL* without physically removing it from the data structure. This is the only case in which nodes that do not store values may be created, and thus they call their trees *partially external*. The speculation-friendly BST proposed by Crain et al. [7] is an internal tree in which updates are split into transactions that modify the abstraction state and those that restructure the tree implementation. Drachsler et al. [9] proposed a partially non-blocking internal BST supporting lock-free Get operations and lock-based update operations via *logic ordering*. This technique applies to both unbalanced BSTs and AVL trees. The CITRUS tree proposed by Arbel and Attiya [1] is an unbalanced internal BST offering wait-free Get operations and lock-based updates. It allows concurrent updates on BSTs based on *Read Copy Update* (RCU) synchronization and fine-grained locking. Ramachandran and Mittal [23] proposed the CASTLE tree, an unbalanced lock-based internal BST which locks edges instead of nodes to achieve higher concurrency.

Lock-free search trees. The first non-blocking BST that has been theoretically proven to be linearizable and non-blocking was proposed by Ellen et al. [11]. It is an unbalanced external BST which implements *Compare-And-Set* (CAS) to perform update operations. In their solution, processes performing update operations help each other so the entire system is guaranteed to make progress. Ellen et al. [10] also proposed a variant of [11] in which the amortized cost of an update operation, op, is $O(h(op) + \dot{c}(op))$, where $h(op)$ is the height of the tree when op is invoked, and $\dot{c}(op)$ is the maximum number of active processes during the execution of op. An unbalanced non-blocking internal BST was proposed by Howley and Jones [19], in which both processes performing Get and processes performing update operations help other processes. In the unbalanced non-blocking external BST proposed by Natarajan and Mittal [21], each process performing update operations operates based on marking edges. Ramachandran and Mittal [24] proposed an internal unbalanced lock-free BST by combining ideas from [19] and [21]. Brown et al. [5] proposed a general technique for developing lock-free BSTs using non-blocking primitive operations [4]. They not only presented a framework for researchers to design new non-blocking BSTs from existing sequential or lock-based BSTs, but also provided guidelines to prove the correctness and progress properties of the new solutions.

3 Preliminaries

In this section, we describe the previous results that are used in our solution. Non-blocking ravl trees use primitive operations *Load-Linked Extended* (LLX) and *Store-Condition Extended* (SCX) proposed by Brown et al. [4] to carry out update steps. LLX and SCX operations are performed on *Data-record*s consisting of mutable and immutable user-defined fields. Immutable fields of a Data-record cannot be further changed after initialization. The mutable fields of a Data-record cannot be further changed after it has been finalized. A successful LLX operation performed on a Data-record r reads r's mutable fields and returns a snapshot of the values of these fields. If an LLX operation on r is concurrent with any SCX operation that modifies r, it is allowed to return *fail*. An LLX operation performed on a finalized Data-

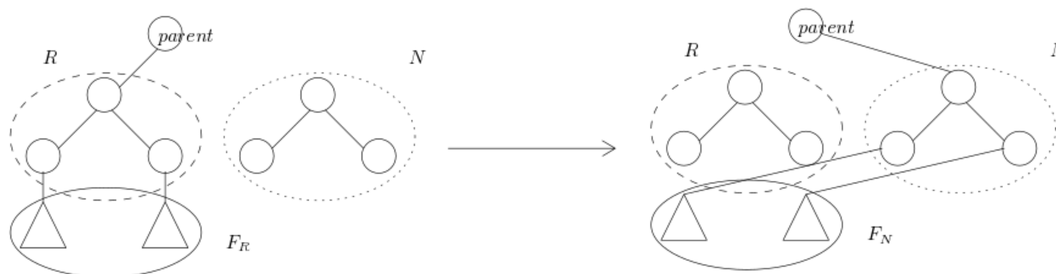

Figure 1 An example of a tree update operation following the template in [5].

record returns *finalized*. An SCX operation requires the following arguments: a sequence of Data-records V, a sequence of Data-records R which is a subset of V, a pointer *fld* pointing to a mutable field of a Data-record in V, and a new value *new*. A successful SCX operation atomically stores *new* into the mutable field pointed to by *fld* and finalizes all Data-records in R. An SCX operation can fail if it is concurrent with any other SCX operation performed by another process that modifies the Data-records in V.

Non-blocking ravl trees use the template proposed by Brown et al. [5] to perform updates. This template provides a framework to design a non-blocking *down tree*, which is a directed acyclic graph in which there is exactly one special root node with indegree 0 and all other nodes are of indegree 1. An update operation using this template atomically removes a subtree from the data structure and replaces it with a newly created subtree using LLX and SCX operations. More precisely, we define R to be the set of nodes in the removed subtree, N to be the set of nodes in the newly added subtree, $F_R = \{x|\text{parent of } x \in R \text{ and } x \notin R\}$ before the update, and $F_N = \{x|\text{parent of } x \in N \text{ and } x \notin N\}$ after the update. Then, the down tree G_R induced by nodes in $R \cup F_R$ before the update is replaced by the down tree G_N induced by nodes in $N \cup F_N$ after the update. Let *parent* be the parent of the root node of G_R in the down tree. Figure 1 gives an example.

An update operation *op* following the tree update template first performs a top-down traversal until it reaches *parent*. It then performs a sequence of LLX operation on *parent* and a contiguous set of its descendants related to the desired update. We define σ to be the set of nodes on which *op* performs these LLX operations, $F_\sigma = \{x|\text{parent of } x \in \sigma \text{ and } x \notin \sigma\}$ before the update, and the down tree G_σ to be the subgraph induced by $\sigma \cup F_\sigma$. If any of these LLX operations returns *fail* or *finalized*, *op* returns *fail*. Otherwise, *op* constructs the required arguments for an SCX operation, which are V, R, *fld* and *new*, and calls SCX to perform the desired update. Lemma 2 summarizes their results.

▶ **Lemma 2** ([5]). *Consider a down tree on which concurrent update operations are performed following the tree update template. Suppose that when constructing SCX arguments in this template, the following conditions are always met:* (1) *V is a subset of σ;* (2) *R is a subset of V;* (3) *fld points to a child pointer of parent, and parent is in V;* (4) *new is a pointer pointing to the root of G_N, and G_N is a non-empty down tree;* (5) *let old be the value of the child pointer pointed by fld before the update, then if old $=$ NULL before the update operation, $R = \emptyset$ and $F_N = \emptyset$;* (6) *if old $\neq$ NULL and $R = \emptyset$, F_N only contains the node pointed to by old;* (7) *all nodes in N must be newly created;* (8) *if a set of concurrent update operations take place entirely during a period of time when no successful SCX operations are performed, the nodes in the sequence V constructed by each of these operations must be ordered in the same tree traversal order;* (9) *if $R \neq \emptyset$ and G_σ is a down tree, then G_R is a non-empty down tree whose root is pointed to by old, and $F_N = F_R$. Then, successful tree update operations*

are linearized at the linearization points of their SCX steps. If tree update operations are performed infinitely often, they succeed infinitely often, and are thus non-blocking.

4 Non-Blocking Ravl Trees

We now describe the non-blocking ravl tree, which is based on the sequential *ravl tree* [26]. We follow the definitions of V, *fld*, *old*, *new*, R, N, F_R and F_N in Section 3. Due to the page limit, the pseudocode and more details of some algorithms are omitted, and we only sketch our proof of Theorem 1. These omitted details be found in the second author's master's thesis [20].

4.1 The Structures and Algorithms of Non-Blocking Ravl Trees

Each node x in a non-blocking ravl tree is represented by a Data-record $< x.left, x.right, x.k, x.v, x.r >$, where $x.left$ and $x.right$ are pointers to x's left child and right child, respectively, $x.k$ is the key that identifies x, $x.v$ is the value stored in x, and $x.r$ is x's rank, which is used for rebalancing. $x.left$ and $x.right$ are mutable fields, and the other fields are immutable. If either one of x's children is missing, we conceptually add *missing* node(s) as its child/children. If both of x's children are missing, x is a *leaf*; otherwise, x is an *internal* node which is for routing purposes only, and we define $x.v = NULL$. If x is a missing node, $x.r = -1$; otherwise, $x.r$ is a non-negative integer. Let z be x's parent. x is called a i*-node* if $z.r - x.r = i$. x is an i,j*-node* if one of x's children is an i-node and the other is a j-node. If x is a 0-node, we call it a *violation*, and the edge (z, x) is called a *violating edge*. In this case, we also say that x is z's 0*-child*, and z is a 0*-parent*. No violation exists in external ravl trees after their rebalancing processes have terminated.

As mentioned previously, different rank rules can achieve different goals for rank-balanced trees. For example, in sequential settings, the original AVL tree is a rank-balanced binary tree in which each internal node is a 1,1-node or a 1,2-node and each leaf is a 1,1-node of rank 0. Our non-blocking ravl tree has the same rank rule as the sequential one: after rebalancing has been carried out, the rank of each parent is strictly larger than that of a child. A sequential ravl tree maintains such an invariant by promoting and demoting nodes and possibly rotating nodes after an insertion, where a promotion or a demotion respectively increases or decreases the rank of a node by 1. In concurrent settings, after an insertion is performed, we use the tree update template [5] to perform rotations and/or replacing some nodes by newly created nodes whose ranks differ by 1. During such a rebalancing process, however, more types of nodes may appear in concurrent settings. For example, a 0,0-node may be created as the result of a concurrent insertion into a non-blocking ravl tree, but it never appears at any time in the sequential case. Thus we carefully design new twists in the algorithms and their analysis in the rest of this section.

To avoid special cases, we introduce the *entry node*, which is the entry point of a non-blocking ravl tree. An empty tree contains an entry node with a single left child leaf, which are *sentinel nodes*. In a non-empty tree, the leftmost grandchild of the entry node is the actual root of the tree. The sentinel nodes in a non-empty tree are the entry node, its left child and its left child's right child. The keys, values and ranks of the sentinel nodes are ∞, *NULL* and ∞, respectively. If a node is neither a missing node nor a sentinel node, we call it an *original node*. We define the height of a non-blocking ravl tree to be the height of the subtree consisting of all its original nodes.

We now describe the implementations of operations in non-blocking ravl trees. When reading the descriptions of these algorithms, readers can refer to Figure 2 for illustration.

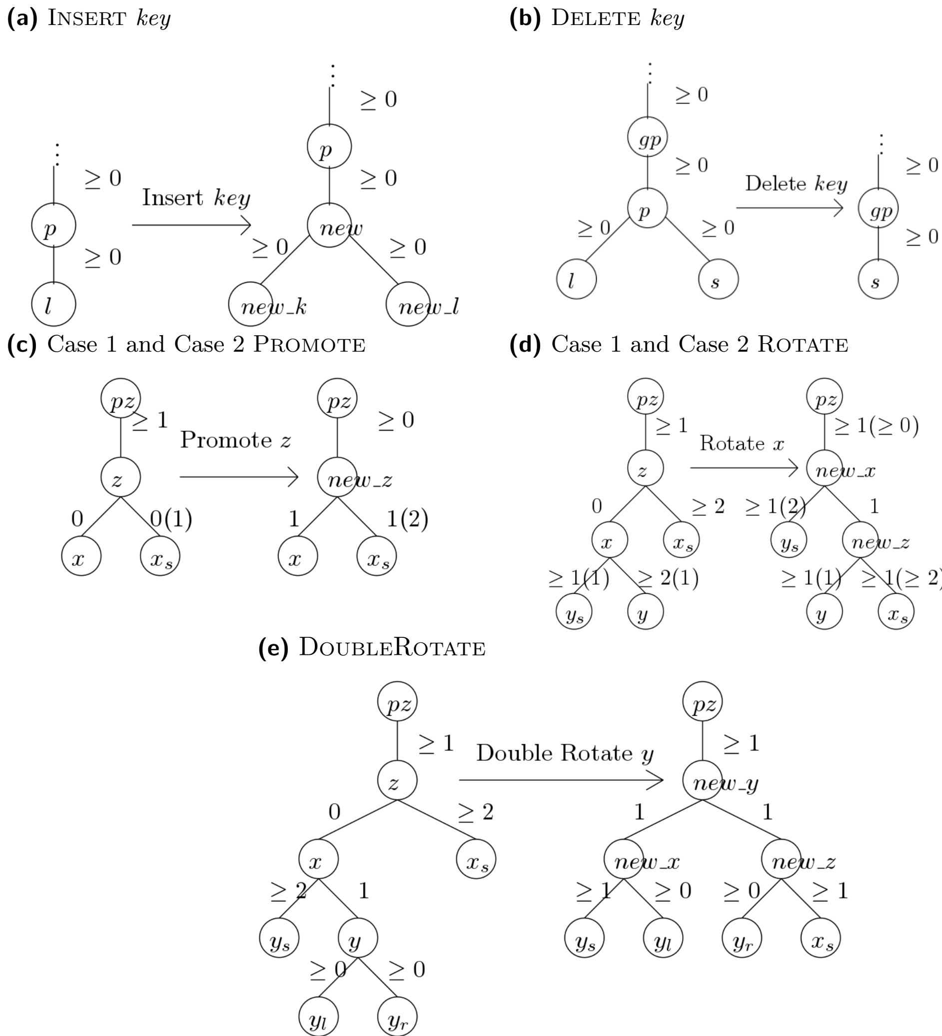

Figure 2 Operations in non-blocking ravl trees.

In this figure, only original nodes are shown. The number beside each edge shows the rank difference between the parent node and the child node.

A SEARCH(*key*) operation performs a regular BST search starting from the entry node and returns the last three nodes visited, $< n_0, n_1, n_2 >$, where n_2 is a leaf, n_1 is n_2's parent, and n_0 is n_2's grandparent. A GET operation calls SEARCH, and returns $n_2.v$ if $n_2.k = key$, or *NULL* otherwise.

An INSERT operation described in Algorithm 1 attempts to insert a new item consisting of *key* and *value*. This operation first calls SEARCH(*key*), which returns a leaf l and its parent p in line 2. If $l.k$ is equal to *key*, then the key is already in the tree, and INSERT returns *false* without inserting any new item in line 3. Otherwise, it invokes TRYINSERT($p, l, key, value$) (Algorithm 2), which carries out the actual insertion shown in Figure 2(a) by following the tree update template summarized in Section 3. If TRYINSERT fails to insert this key, the INSERT operation retries this process from scratch.

When performing the TRYINSERT operation, primitive operations such as LLX and SCX are called to implement the tree update template. Whenever these operations fail, or

Alg. 1 INSERT(key, $value$)

1: **repeat**
2: $\quad < -, p, l > \leftarrow$ SEARCH(key)
3: $\quad$ **if** $l.k = key$ **then** return *false*
4: $\quad$ $result \leftarrow$ TRYINSERT($p, l, key, value$)
5: **until** $result \neq fail$
6: **if** *result* **then** CLEANUP(key)
7: return *true*

Alg. 2 TRYINSERT($p, l, key, value$)

8: **if** LLX(p) $\in \{fail, finalized\}$ **then**
9: $\quad$ return *fail*
10: **if** $l = p.left$ **then** $fld \leftarrow \&(p.left)$
11: **else if** $l = p.right$ **then**
12: $\quad fld \leftarrow \&(p.right)$
13: **else** return *fail*
14: **if** LLX(l) $\in \{fail, finalized\}$ **then**
15: $\quad$ return *fail*
16: $new_k \leftarrow$ pointer to a new Data-Record $< \text{missing}, \text{missing}, key, value, 0 >$
17: **if** $l = entry.left$ **then**
18: $\quad new_l \leftarrow$ pointer to a new Data-Record $< \text{missing}, \text{missing}, l.k, l.v, \infty >$
19: **else**
20: $\quad new_l \leftarrow$ pointer to a new Data-Record $< \text{missing}, \text{missing}, l.k, l.v, 0 >$
21: **if** $key < l.k$ **then**
22: $\quad new \leftarrow$ pointer to a new Data-Record $< new_k,\ new_l,\ l.k,\ NULL,\ l.r >$
23: **else**
24: $\quad new \leftarrow$ pointer to a new Data-Record $< new_l,\ new_k,\ key,\ NULL,\ l.r >$
25: **if** SCX($\{p, l\}, \{l\}, fld, new$)$\neq fail$ **then**
26: $\quad$ return ($new.r = 0$)
27: **else** return *fail*

Alg. 3 CLEANUP(key)

28: **while** *true* **do**
29: $\quad gp \leftarrow NULL$; $p \leftarrow entry$; $l \leftarrow entry.left$; $l_s \leftarrow entry.right$
30: $\quad$ **while** *true* **do**
31: $\quad\quad$ **if** l is a leaf **then** return
32: $\quad\quad$ **if** $key < l.k$ **then**
33: $\quad\quad\quad gp \leftarrow p$; $p \leftarrow l$; $l \leftarrow l.left$; $l_s \leftarrow l.right$
34: $\quad\quad$ **else**
35: $\quad\quad\quad gp \leftarrow p$; $p \leftarrow l$; $l \leftarrow l.right$; $l_s \leftarrow l.left$
36: $\quad\quad$ **if** $p.r = l.r + 1$ and $p.r = l_s.r$ **then**
37: $\quad\quad\quad$ TRYREBALANCE(gp, p, l_s)
38: $\quad\quad\quad$ break out of the inner loop
39: $\quad\quad$ **if** $p.r = l.r$ **then**
40: $\quad\quad\quad$ TRYREBALANCE(gp, p, l)
41: $\quad\quad\quad$ break out of the inner loop

whenever an LLX is performed on a Data-record that has been finalized, TRYINSERT returns immediately with failure. Otherwise, TRYINSERT successfully inserts a key, and returns a boolean value to indicate whether a violation has been created. More specifically, TRYINSERT first performs an LLX operation on p, and then, depending on if l is p's left child or right child, we let pointer *fld* point to the correct child pointer field of p (lines 8–12). If the structure of the related portion of the tree has been changed since the corresponding SEARCH operation returns, so that l is not a child of p anymore when we perform the check above, TRYINSERT returns *fail* in line 13. Then we perform an LLX operation on l in line 14, and create a new subtree rooted at the node pointed to by pointer *new* from line 16 to line 24: The key of the root of this new subtree is $\max(l.k, key)$ and its rank is set to $l.r$. This root has two children which are both leaf nodes, and they are pointed to by pointers new_k and new_l. The leaf node pointed to by new_k stores the key and value of the item to be inserted, and its rank is set to 0. The other leaf stores node l's key and value; its rank is set to ∞ if l was the left child of the entry node (i.e., *entry.left*) which is a sentinel node, and 0 otherwise. Next we construct the SCX arguments in line 25, where we define $V = \{p, l\}$ and $R = \{l\}$. The TRYINSERT operation then calls SCX with the constructed arguments and

attempts to atomically store *new* in the child field of p pointed to by *fld* while finalizing l. If this SCX operation fails, TryInsert returns *fail*. Otherwise, if the rank of the root of the newly inserted subtree is 0, TryInsert returns *true* to indicate that a new violation has been created after a successful insertion, and the corresponding Insert calls Cleanup (Algorithm 3) to rebalance the tree. If no new violation has been created, the TryInsert operation returns *false*. Finally, the corresponding Insert returns *true* to indicate that a new item has been inserted.

A Cleanup operation resolves the new violation created by the corresponding Insert operation, as well as all potential new violations created by the rebalancing steps during the process. Starting from the entry node (line 29), it performs a BST search for *key* and keeps track of the last three consecutive nodes visited, gp, p and l, as well as l's sibling, l_s. If p is a 0,1-node, and l_s is p's 0-child (line 36), the Cleanup operation calls TryRebalance(gp, p, l_s) to resolve the violation on l_s. This step is required to avoid livelocks. Otherwise, if l is a 0-node (line 39), the Cleanup operation calls TryRebalance(gp, p, l) to resolve the violation on l. Once the TryRebalance subroutine returns, the corresponding Cleanup operation retries this process in case that a new violation has been created by the previous TryRebalance call. The Cleanup operation returns when l reaches a leaf in line 31. At this point, the violation created by the corresponding Insert operation has been resolved.

A TryRebalance operation takes three consecutive nodes pz, z and x, which correspond to the nodes with same names in Figures 2(c)–(e). For Figure 2(c) and Figure 2(d), the numbers outside of the parenthesis show the rank difference for case 1 Promote and Rotate, respectively, while the numbers inside are for case 2 Promote and Rotate, respectively. Without loss of generality, assume that x is z's left child. Let y be x's right child, and y_s be x's left child. The TryRebalance operation attempts to resolve the violation on x following the tree update template. Based on the ranks of pz, z, z's sibling z_s, x and x's sibling x_s, the TryRebalance operation perform one of the following rebalancing steps: (1) if z is a 0,0-node or 0,1-node, it performs a case 1 or case 2 Promote on z as illustrated in Figure 2(c) by replacing z with a newly created node new_z whose rank is $z.r + 1$; (2) if z is a 0,i-node, where $i \geq 2$, there are three subcases: (2a) if $x.r \geq y.r + 2$ or y is a missing node, it performs a case 1 Rotate on x as illustrated in Figure 2(d) and replace x and z with newly created nodes new_x and new_z whose ranks are $x.r$ and $z.r - 1$, respectively; (2b) if $x.r = y.r + 1$ and $x.r = y_s.r + 1$, it performs a case 2 Rotate on x as illustrated in Figure 2(d) and replace x and z with newly created nodes new_x and new_z whose ranks are $x.r + 1$ and $z.r$, respectively; (2c) if $x.r = y.r + 1$ and $x.r \geq y_s.r + 2$, it performs a DoubleRotate operation on y as illustrated in Figure 2(e) and replace x, y and z with newly created nodes new_x, new_y and new_z whose ranks are $x.r - 1$, $y.r + 1$ and $z.r - 1$, respectively. If a rebalancing step does not introduce a new violation after resolving the old one, we say that it is *terminating*; otherwise, it is *non-terminating*.

A Delete operation follows the tree update template to remove a key as shown in Figure 2(b), and no rebalancing is needed after the tree is updated. It returns *false* if the key to be deleted does not exist in the tree, and *true* otherwise.

4.2 Properties of Non-Blocking Ravl Trees

In this section, we sketch our proof of Theorem 1. We first prove the correctness of our algorithms by arguing that the rebalancing operations described in Figures 2(c)–(e) cover all possible violation cases in concurrent settings. The only case that is not explicitly covered by our algorithm is when the 0-child of a 0,i-node, where $i \geq 2$, has a 0-child, and the following lemma shows that such a case does not occur.

▶ **Lemma 3.** *If a* 0*-node in a non-blocking ravl tree has at least one* 0*-child, then this node's parent is either a* 0,0*-node or a* 0,1*-node.*

We then prove that a non-blocking ravl tree remains a BST at any time, and thus all BST operations are performed correctly. For this, we define the *search path* [5] for a key k at any time to be the root-to-leaf path formed by the original nodes that a SEARCH operation for k would visit as if this operation were performed instantaneously at this time. We then prove that a SEARCH operation in non-blocking ravl trees follows the correct search path at any time, and thus is performed correctly. We complete the proof by showing that the BST property of a non-blocking ravl tree is always preserved after update operations.

We next prove the linearizability of our solution. It would have been ideal to define the linearization point of a SEARCH to be the time when it reaches a leaf node that is in the tree when visited. However, this claim cannot be guaranteed as a SEARCH for a given key, k, in non-blocking ravl trees does not check the status of visited nodes, and by the time when it is linearized, this leaf might not be in the tree anymore. Thus, we prove that this leaf was in the tree and on the search path for k at some time earlier during this SEARCH operation, so it is the correct node to return. We then define the linearization points of all operations:

▶ **Lemma 4.** *The linearization points of operations in non-blocking ravl trees are defined as follows:* (1) *a SEARCH operation can be linearized when the leaf eventually reached was on the search path for the query key (such a time exists);* (2) *a GET operation is linearized at the linearization point of its SEARCH step;* (3) *if an INSERT operation returns true, it is linearized at the linearization point of the SCX step performed by its TRYINSERT step;* (4) *if an INSERT operation returns false, it is linearized at the linearization point of its SEARCH step;* (5) *if a DELETE operation returns true, it is linearized at the linearization point of the SCX step by its TRYDELETE step;* (6) *if a DELETE operation returns false, it is linearized at the linearization point of its SEARCH step.*

Next, we show the progress properties of non-blocking ravl trees. We first show that if updates are invoked infinitely often, they follow the tree update template infinitely often. Thus, by Lemma 2, they will succeed infinitely often. We then construct a proof by contradiction to show that all operations in a non-blocking ravl tree are non-blocking. Consider a non-blocking ravl tree built via a sequence of arbitrarily intermixed GET, INSERT and DELETE operations. To derive a contradiction, assume that starting from a certain time T_1, active processes are still executing instructions, but none of them completes any operation. These active operations must include updates, because otherwise, the tree structure would remain stable and all GET operations would eventually terminate. We then use the property that update operations succeed infinitely often when called infinitely often and the fact that each update operation can succeed at most once, to argue that there exists a certain time T_2, after which the tree can only be modified by TRYREBALANCE operations. We next observe that m INSERT operations can only create at most $2m$ violations, and by carefully analyzing the relationship between different types of terminating and nonterminating rebalancing steps, we derive contradictions.

Finally, we bound the tree height. Consider a ravl tree T built via a sequence of arbitrarily intermixed INSERT and DELETE operations from an empty tree. At a time t, let m be the number of INSERT operations that have successfully inserted new keys into T. Let T' be the balanced ravl tree rooted at rt' that can be constructed by completing all rebalancing operations to eliminate all the violations in T. We first bound the height of T' by augmenting the potential functions defined in [26] to include 0,0-nodes in our analysis. Here F_i denotes the ith Fibonacci number, i.e., $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$ where $i \geq 2$. We also

use the inequality $F_{i+2} \geq \phi^i$. We define the potential of an original node x in an external ravl tree, whose rank is k, as follows: (1) if x is a 0,0-node, its potential is F_{k+3}; (2) if x is a 0,1-node, its potential is F_{k+2}; (3) if x is a 0,i-node, where $i \geq 2$, its potential is F_{k+1}; (4) if x is a 1,1-node, its potential is F_k; 5) otherwise, its potential is 0. We also define the potential of an external ravl tree to be the sum of the potentials of all its original nodes.

The keys steps of our analysis involves showing that each rebalancing step either does not change the potential of the tree, or decreases it by a certain amount. However, the analysis would not go through if a rebalancing step could also create a new 0,0-node. To ensure that this case never happens, we add additional checks in our rebalancing steps and prove the following lemma for non-terminating operations:

▶ **Lemma 5.** *A non-terminating operation that is either a* Promote *operation or a case* 2 *Rotate operation cannot create a new* 0,0*-node.*

From Figure 2(e), the node y on which a DoubleRotate is performed is allowed to have two 0-children. If this could indeed happen, our analysis would not work again. Thus we prove the next lemma which shows that y can have at most one 0-child in such a case:

▶ **Lemma 6.** *Consider a DoubleRotate operation as shown in Figure 2(e). The node y on which this operation is performed on can have at most one* 0*-child.*

We then perform case analysis to show that an Insert increase the potential of the tree by at most 2, while a Delete does not increase the tree potential. We also show that any non-terminating case 1 Promote/case 2 Promote/case 2 Rotate does not change the tree potential, while a case 1 Rotate or a DoubleRotate does not increase the tree potential. For a terminating case 1 Promote/case 2 Promote/case 2 Rotate, let k be the rank of the node that is promoted/rotated. Then such an operation decreases the potential of the tree by at most F_{k+2}. The potential decrease is exactly F_{k+2} if this terminating case 1 and 2 Promote operations is performed on the root, rt', of T', or if this case 2 Rotate operations is performed on one of rt''s children. Here we analyze the case 1 Promote as an example:

Let z be the 0,0-node on which a case 1 Promote operation is performed, and let k be its rank. The potential of z before the promotion was F_{k+3}. Let new_z be the newly added node that replaces z. Then, new_z is a 1,1-node, its rank is $k+1$, and its potential is F_{k+1}. Thus, replacing z by new_z decreases the potential of the tree by F_{k+2}. Let $p(z)$ be z's parent before the promotion. In the non-terminating case, by Lemma 5, $p(z)$ could not be a 1,0-node before this operation. If $p(z)$ was a 1,1-node, whose rank was $k+1$ before the promotion, it becomes a 0,1-node after the promotion, and its potential changes from F_{k+1} to F_{k+3}. If $p(z)$ was a 1,i-node, where $i \geq 2$, whose rank was $k+1$ before the promotion, it becomes a 0,i-node after the promotion, and its potential changes from 0 to F_{k+2}. In either case, the potential of $p(z)$ is increased by F_{k+2}, and the potential of the tree is not changed. In the terminating case, if $p(z)$ was a 1,2-node, whose rank was $k+2$ before the promotion, it becomes a 1,1-node after the promotion, and its potential changes from 0 to F_{k+2}. If $p(z)$ was a 0,2-node with rank $k+2$ before the promotion, where z was its 2-child, $p(z)$ becomes a 0,1-node after the promotion, and its potential changes from F_{k+3} to F_{k+4}. In either case, the potential of $p(z)$ is increased by F_{k+2}, and the potential of the tree is not changed if z is not the root of the tree, i.e., $p(z)$ is an original node. If $p(z)$ was not a 1,2-node or 0,2-node, this promotion does not change its potential, and the tree potential is decreased by F_{k+2}. If z is the root of the tree, then $p(z)$ is a sentinel node. Since the potential of the tree is the sum of the potentials of its original nodes only, the tree potential is decreased by F_{k+2}.

We now make use of the above claims to bound the tree height. Initially, the potential of an empty tree is 0. Based on the analysis above, the potential of the tree can only be

increased by at most 2 after each INSERT operation. The first INSERT operation, where we insert the root of T', rt', into the empty tree, does not change the potential of the tree. Thus, after m INSERT operations, the potential of the tree is at most $2(m-1)$. Since the initial rank of rt' is 0, the number of operations that increases the rank of rt' by 1, i.e., terminating case 1 and 2 PROMOTE operations performed on rt' plus the number of case 2 ROTATE operations performed on one of rt''s children, is equal to $rt'.r$. As analyzed above, each time one of these operations changes the rank of rt' from k to $k+1$, the potential of the tree is decreased by F_{k+2}. Therefore, these operations decrease the potential of the tree by $\sum_{i=0}^{rt'.r-1} F_{i+2} = \sum_{i=2}^{rt'.r+1} F_i = F_{rt'.r+3} - 2$. Since the potential of the tree is always non-negative, $2(m-1) \geq F_{rt'.r+3} - 2$, and $2m \geq F_{rt'.r+3} > F_{rt'.r+2} \geq \phi^{rt'.r}$. Therefore, $rt'.r < \log_\phi 2m$. We then show that the height of T' is no greater than $rt'.r$, which is bounded by $\log_\phi(2m)$ from the analysis above. Next, we borrow ideas from [5] to prove that the number of violating edges on each search path in T is bounded by the number, c, of active INSERT operations that are in the rebalancing phase. Using these two claims, we bound the height of T by $\log_\phi(2m) + c$.

5 Experimental Evaluation

We compared the non-blocking ravl tree, **lf-ravl**, against the following concurrent BSTs: (1) **lf-chrm**, the non-blocking chromatic tree proposed by Brown et al. [5] which uses the tree update template summarized in Section 3; (2) **lf-chrm6**, a variant of lf-chrm in which the rebalancing process is only invoked by an update operation if the number of violations on the corresponding search path exceeds 6 [5]. Compared to lf-chrm, this variant achieves superior performance since it reduces the total number of rebalancing steps; (3) **lc-davl**, the concurrent AVL tree proposed by Drachsler et al. [9] which supports wait-free GET and lock-based updates; (4) **lf-nbst**, the unbalanced external non-blocking BST proposed by Natarajan and Mittal [21]; (5) **lf-ebst**, the unbalanced external non-blocking BST proposed by Ellen et al. [11]; (6) **lf-ibst**, the unbalanced internal non-blocking BST proposed by Ramachandran and Mittal [24]; (7) **lc-cast**, the unbalanced internal lock-based BST proposed by Ramachandran and Mittal [23]; (8) **lc-citr**, the unbalanced internal lock-based BST proposed by Arbel and Attiya [1]. We also implemented a variant of the non-blocking ravl tree called **lf-ravl3** in which CLEANUP is only invoked by INSERT if the number of violations on the search path exceeds 3. We allow at most three violations on a search path since experiments showed that this is when the tree achieved the best performance in most experimental settings.

The original source code for lf-chrm [5], lf-ebst [11] and lc-davl [9] was written in Java, and we re-implemented them in C. We used the source code for other concurrent BSTs developed by their original authors [21, 24, 23], and followed the framework in [16] to test the performance. CAS operations are performed using the APIs provided by `libatomic_ops` [2]. We also used `jemalloc` [14] for memory allocations to achieve optimal results. For lock-based data structures, we used mutex locks and the APIs provided by `pthread`. All experiments were conducted on a computer with two Intel® Xeon® E5-2650 v2 processors (20M Cache, 2.60 GHz) supporting 32 hardware threads in total. It operates on CentOS 6.7. All implementations were compiled using `gcc-4.4.7` with `-O3` optimization.

We first conduct experiments using random numbers as operation keys. All keys are positive integers within a user-specified range generated under uniform distributions. We also used different operation mixes. An operation mix xr-yi-zd represents x% GET operations, y% INSERT operations and z% DELETE operations. As in [9], we used operation mixes 90r-9i-1d

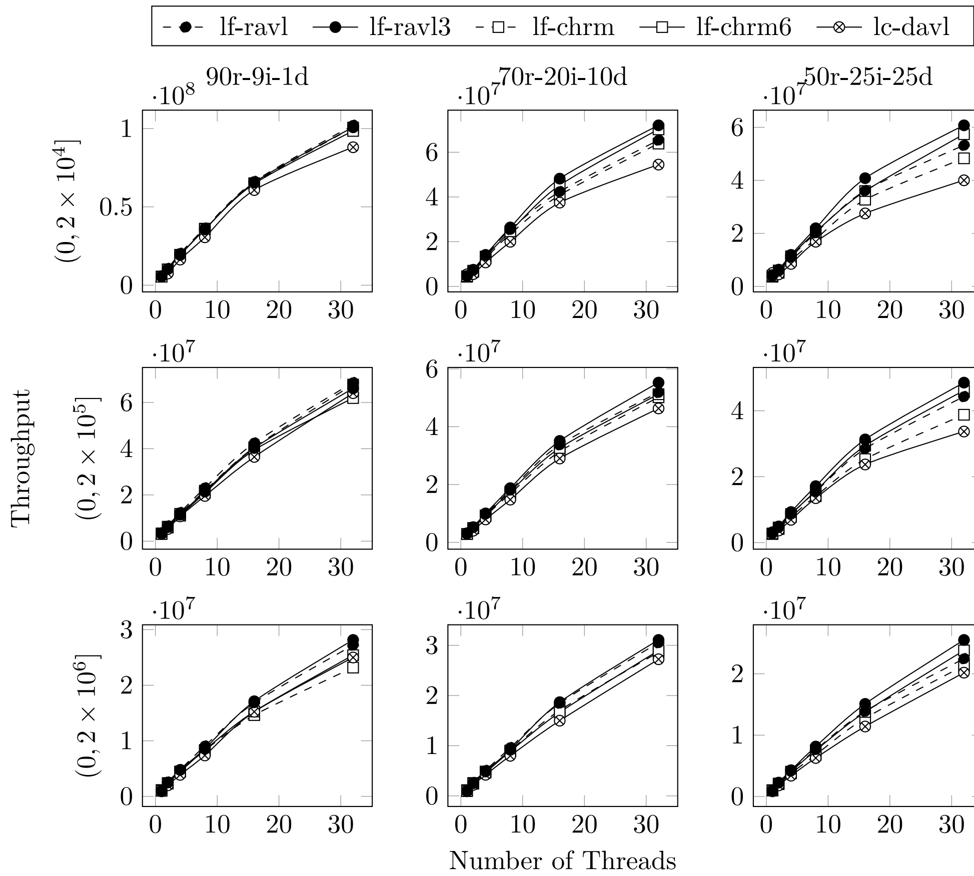

Figure 3 Experimental results comparing lf-ravl and lf-ravl3 against other self-balancing concurrent BSTs using randomly generated sequences under uniform distributions.

(read-dominant), 70r-20i-10d (mixed) and 50r-25i-25d (write-dominant). For each concurrent BST, under each key range and using each operation mix, we ran its program with 1, 2, 4, 8, 16 and 32 threads for 5 seconds as one trial. The performance of the data structures is measured using their average throughput (the number of operations finished per second). We prefilled each data structure using randomly generated keys until 50% of the keys in the key range are inserted into the tree.

Figure 3 shows the experimental results comparing lf-ravl and lf-ravl3 against other self-balancing concurrent BSTs. To test how different levels of contention can influence the performance, we test the algorithms on the following key ranges: $(0, 2 \times 10^4]$, $(0, 2 \times 10^5]$, and $(0, 2 \times 10^6]$. We group studies under the same key range by row, and studies using the same operation mix by column. The x-axis of each study shows the number of threads, and the y-axis shows the average throughput. For example, the studies in the first row shows results for studies under key range $(0, 2 \times 10^4]$. For this key range, lf-ravl outperforms lf-ravl3 slightly in case 90r-9i-1d. Since lf-ravl3 has a larger average search path length than lf-ravl, GET operations in lf-ravl are faster than in lf-ravl3. In other cases, lf-ravl3 achieves superior performance. This is because lf-ravl3 reduces the total number of rebalancing steps and introduces less overhead. lf-ravl outperforms lf-chrm in every case. Though lf-chrm6 outperforms lf-ravl in case 50r-25i-25d, lf-ravl3 still outperforms lf-chrm6. This shows that lf-ravl and lf-ravl3 improve performance by avoiding rebalancing after DELETE operations.

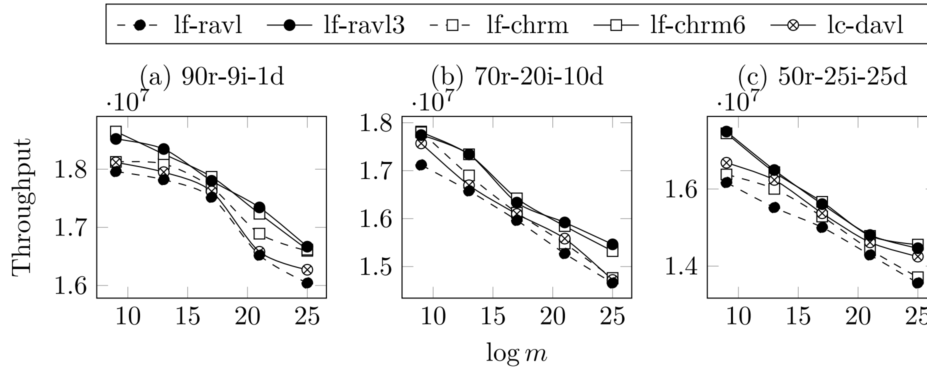

Figure 4 Experimental results comparing lf-ravl and lf-ravl3 against other self-balancing concurrent BSTs using sequences of size 2^{26} with different degrees of presortedness (32 threads).

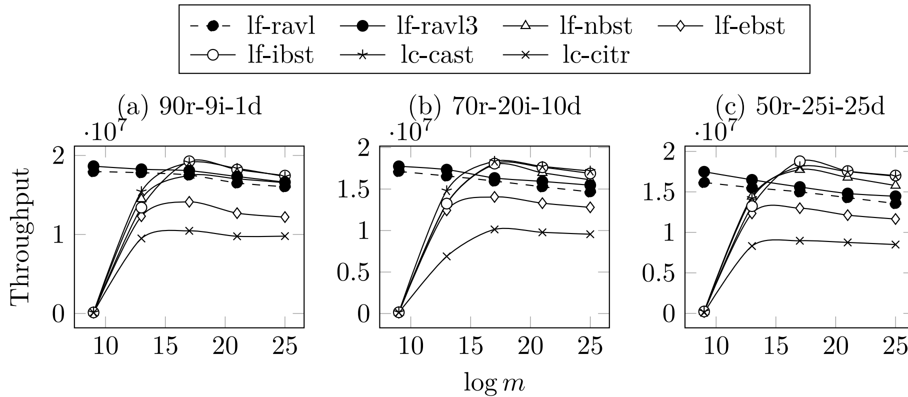

Figure 5 Experimental results comparing lf-ravl and lf-ravl3 against unbalanced concurrent BSTs using sequences of size 2^{26} with different degrees of presortedness (32 threads).

lf-ravl3 and lf-chrm6 both outperform lc-davl in every case. Results are similar for other key ranges, showing that lf-ravl and lf-ravl3 outperform other data structures in most cases. This performance difference is more noticeable with smaller key ranges (higher contention levels), which shows that our solution is more contention-friendly compared to other concurrent self-balancing BSTs.

Data generated from uniform distributions favor unbalanced BSTs, as they are balanced with high probability and do not have the overheads of rebalancing. Nevertheless, lf-ravl and lf-ravl3 still outperform some of the unbalanced BSTs in our studies. For example, they scale much better for operation sequences with a higher update ratio (70r-20i-10d and 50r-25i-25d) compared to lc-citr. The detailed experimental results are reported in [20].

We further test the performance using synthetic sequences in which keys are partially sorted, i.e., with a certain degree of presortedness. Presortedness has been extensively studied in adaptive sorting algorithms [12, 25, 13], and we apply this concept to the context of studying concurrent binary search trees. The presortedness of a sequence is measured by the number of *inversions*, which is the number of pairs of elements of the sequence in which the first item is larger than the second. We generate the inversions in a data sequence using the algorithm in [13, 25], which is expected to generate $mn/2$ inversions on average, where n is the size of the data sequence and m is a user-specified parameter to control the degree

of presortedness. We constructed data sequences with presortedness in the following way: starting from a sorted sequence of distinct positive integers of size $n = 2^{26}$, we generated sequences with different levels of presortedness by creating inversions using the algorithm in [12, 25, 13] with the values of m in range $[2^9, 2^{25}]$. For each data structure, using each data sequence and under each operation mix, we ran the program with 32 threads (which is the maximum number of hardware threads available in our setup) as one trial. Before each trial, we prefilled the data structure using the first half of each data sequence to ensure stable performance. During each trial, we insert items in the order of the data sequences. We also randomly selected keys within range $(0, 2^{26}]$ to perform GET and DELETE operations. Each trial terminated after all numbers in the data sequence had been inserted.

Figure 4 and Figure 5 give the experimental results comparing lf-ravl and lf-ravl3 against concurrent self-balancing BSTs and unbalanced BSTs, respectively, in which the x-axis shows the value of $\log m$ and the y-axis shows the throughput. lf-ravl3 outperforms other self-balancing BSTs in most cases. lf-chrm outperforms lf-ravl slightly. We believe that this is because the expected success rate of INSERT operations in the current experimental settings (100%) is higher than the previous studies (at most 50%), as lf-ravl rebalances the tree more often. lf-ravl3 outperforms lc-davl, while lf-davl achieves slightly better performance compared to lf-ravl.

We also consider how the value of m affect the heights of the BSTs when comparing the performance of our solution against other unbalanced concurrent BSTs. The average heights of our solution are not affected by the value of m significantly; the tree height is always between 30 and 34. When m is no greater than 2^9, the heights of unbalanced BSTs are notably larger than the heights of self-balancing trees, and lf-ravl and lf-ravl3 outperform unbalanced BSTs significantly. The average tree heights of unbalanced BSTs decrease from 41445 to 177 when m changes from 2^9 to 2^{15}, which explains their rapid performance improvement. When m is approximately 2^{15}, where unbalanced BSTs are approximately 5 times as tall as our solution, unbalanced BSTs start to have better performance. When m is larger than 2^{17}, unbalanced BSTs outperform our solution. Previous studies [22] have shown that, in real-world applications, it is very common for data to be accessed in some sorted order, and unbalanced BSTs are likely to be more than five times taller than self-balancing BSTs when implemented in system software products. In addition, if the comparisons between keys require more time (e.g., the keys are strings), the smaller heights of self-balancing BSTs will potentially be even more attractive. From the results and analysis above, we believe that our solution is the best candidate for many real-world applications.

References

1. Maya Arbel and Hagit Attiya. Concurrent updates with RCU: Search tree as an example. In *Proc. PODC*, pages 196–205, New York, NY, USA, 2014. ACM. `doi:10.1145/2611462.2611471`.
2. Hans Boehm. The atomic_ops library (libatomic_ops). URL: `https://github.com/ivmai/libatomic_ops`.
3. Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *SIGPLAN Not.*, 45(5):257–268, January 2010. `doi:10.1145/1837853.1693488`.
4. Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proc. PODC*, pages 13–22, New York, NY, USA, 2013. ACM. `doi:10.1145/2484239.2484273`.

5. Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. *SIGPLAN Not.*, 49(8):329–342, February 2014. Source code available at `http://www.cs.toronto.edu/~tabrown/chromatic/`. `doi:10.1145/2692916.2555267`.
6. Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient lock-free binary search trees. In *Proc. PODC*, pages 322–331, New York, NY, USA, 2014. ACM. `doi:10.1145/2611462.2611500`.
7. Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *Proc. PPOPP*, pages 161–170, 2012.
8. Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In *Proc. Euro-Par*, pages 229–240, Berlin, Heidelberg, 2013. Springer-Verlag. `doi:10.1007/978-3-642-40047-6_25`.
9. Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. *SIGPLAN Not.*, 49(8):343–356, February 2014. Source code available at `https://github.com/logicalordering/trees`. `doi:10.1145/2692916.2555269`.
10. Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proc. PODC*, pages 332–340, New York, NY, USA, 2014. ACM. `doi:10.1145/2611462.2611486`.
11. Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proc. PODC*, pages 131–140, New York, NY, USA, 2010. ACM. Source code available at `http://www.cs.toronto.edu/~tabrown/ksts/StaticDictionary5.java`. `doi:10.1145/1835698.1835736`.
12. Amr Elmasry. *Exploring New Frontiers of Theoretical Informatics: IFIP 18th World Computer Congress TC1 3rd International Conference on Theoretical Computer Science (TCS2004) 22–27 August 2004 Toulouse, France*, chapter Adaptive Sorting with AVL Trees, pages 307–316. Springer US, Boston, MA, 2004. `doi:10.1007/1-4020-8141-3_25`.
13. Amr Elmasry and Abdelrahman Hammad. An empirical study for inversions-sensitive sorting algorithms. In *Proc. WEA*, pages 597–601, 2005. `doi:10.1007/11427186_52`.
14. Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. BSDCan*, 2006. URL: `http://www.canonware.com/jemalloc/`.
15. Michael T. Goodrich and Roberto Tamassia. *Algorithm Design and Applications.* Wiley Publishing, 1st edition, 2014.
16. Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proc. PPoPP*, pages 1–10, New York, NY, USA, 2015. ACM. `doi:10.1145/2688500.2688501`.
17. Bernhard Haeupler, Siddhartha Sen, and Robert Endre Tarjan. Rank-balanced trees. *ACM Trans. Algorithms*, 11(4):30, 2015. `doi:10.1145/2689412`.
18. Philip W. Howard and Jonathan Walpole. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience*, 2013.
19. Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proc. SPAA*, pages 161–171, New York, NY, USA, 2012. ACM. `doi:10.1145/2312005.2312036`.
20. Mengdu Li. Deletion without rebalancing in non-blocking self-balancing binary search trees. Master's thesis, Dalhousie University, 2016.
21. Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proc. PPoPP*, pages 317–328, New York, NY, USA, 2014. ACM. Source code available at `https://github.com/anataraja/lfbst`. `doi:10.1145/2555243.2555256`.
22. Ben Pfaff. Performance analysis of BSTs in system software. In *Proc. SIGMETRICS*, pages 410–411, New York, NY, USA, 2004. ACM. `doi:10.1145/1005686.1005742`.
23. Arunmoezhi Ramachandran and Neeraj Mittal. CASTLE: Fast concurrent internal binary search tree using edge-based locking. In *Proc. PPoPP*, pages 281–282, New York, NY,

USA, 2015. ACM. Source code available at `https://github.com/arunmoezhi/castle`. `doi:10.1145/2688500.2688551`.

24 Arunmoezhi Ramachandran and Neeraj Mittal. A fast lock-free internal binary search tree. In *Proc. ICDCN*, pages 37:1–37:10, New York, NY, USA, 2015. ACM. Source code available at `https://github.com/arunmoezhi/lockFreeIBST`. `doi:10.1145/2684464.2684472`.

25 Riku Saikkonen and Eljas Soisalon-Soininen. Bulk-insertion sort: Towards composite measures of presortedness. In *Proc. SEA*, pages 269–280, 2009. `doi:10.1007/978-3-642-02011-7_25`.

26 Siddhartha Sen and Robert E. Tarjan. Deletion without rebalancing in balanced binary trees. In *Proc. SODA*, pages 1490–1499, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.