Deletion Without Rebalancing in Non-Blocking Self-Balancing Binary Search Trees

by

Mengdu Li

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
April 2016

© Copyright by Mengdu Li, 2016

Table of Contents

List of Tables . . . . **iv**

List of Figures . . . . **v**

Abstract . . . . **vii**

List of Abbreviations and Symbols Used . . . . **viii**

Acknowledgements . . . . **xi**

Chapter 1 Introduction **1**

1.1 Our Work 3

Chapter 2 Related Work **6**

2.1 Lock-Based BSTs 7

2.2 Lock-Free BSTs 7

Chapter 3 Preliminaries **9**

3.1 LLX and SCX Operations 9

3.2 Tree Update Template 11

Chapter 4 Sequential External Ravl Trees **16**

4.1 The Structures of Sequential External Ravl Trees 16

4.2 Operations in Sequential External Ravl Trees 17

4.3 Bounding the Tree Height 21

Chapter 5 Non-Blocking Ravl Trees **28**

5.1 The Structures and Algorithms of Non-Blocking Ravl Trees 28

5.2 Correctness of Non-Blocking Ravl Trees 41

5.3 Progress Properties of Non-Blocking Ravl Trees 50

5.4 Bounding the Tree Height 53

Chapter 6 Experimental Evaluation 67

6.1 Compared Data Structures 67

6.2 Implementation Details 68

6.3 Random Data Set 69

6.4 Data Sequence with Difference Degrees of Presortedness 77

Chapter 7 Discussion 84

Bibliography 86

List of Tables

6.1 Average tree heights using sequences with existing order under different values of m. 82

List of Figures

3.1 An example of a tree update operation following the template in [7]. 12

4.1 The structures of an empty and a non-empty sequential external ravl tree. 16

4.2 Operations in sequential external ravl trees. The number beside each edge indicates the rank difference between the parent and the child. 18

5.1 Operations in non-blocking ravl trees. 29

6.1 Experimental results comparing lf-ravl and lf-ravl3 against self-balancing concurrent BSTs using randomly generated data within key range $(0, 2 \times 10^4]$. 70

6.2 Experimental results comparing lf-ravl and lf-ravl3 against self-balancing concurrent BSTs using randomly generated data within key range $(0, 2 \times 10^5]$. 71

6.3 Experimental results comparing lf-ravl and lf-ravl3 against self-balancing concurrent BSTs using randomly generated data within key range $(0, 2 \times 10^6]$. 72

6.4 Experimental results comparing lf-ravl and lf-ravl3 against unbalanced concurrent BSTs using randomly generated data within key range $(0, 2 \times 10^4]$. 74

6.5 Experimental results comparing lf-ravl and lf-ravl3 against unbalanced concurrent BSTs using randomly generated data within key range $(0, 2 \times 10^5]$. 75

6.6 Experimental results comparing lf-ravl and lf-ravl3 against unbalanced concurrent BSTs using randomly generated data within key range $(0, 2 \times 10^6]$. 76

6.7 Experimental results comparing lf-ravl and lf-ravl3 against self-balancing concurrent BSTs using sequences of size 2^{26} with different degrees of presortedness (32 threads). 80

6.8 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). 81

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 bounded by $\log_{\phi}(2m) + c$, where $\phi = \frac{1+\sqrt{5}}{2}$ 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 performing Insert operations at this time. The most significant feature of the non-blocking ravl tree is that it does not rebalance itself after Delete operations. Compared to other self-balancing *Binary Search Trees* (BSTs), which typically introduce a large number of rebalancing cases after Delete operations, the non-blocking ravl tree has much fewer rebalancing cases, which makes it much simpler to implement in practice. Our experimental studies show that our solution is potentially the best candidate for many real-world applications.

List of Abbreviations and Symbols Used

F_N	Set of nodes composed of children of nodes in set N in the tree update template.
F_R	Set of nodes composed of children of nodes in set R in the tree update template.
F_i	The ith Fibonacci number.
F_σ	Set of nodes composed by child of nodes in σ.
G_N	Graph induced by nodes in $N \cup F_N$ in the tree update template.
G_R	Graph induced by nodes in $R \cup F_R$ in the tree update template.
G_σ	Graph induced by $\sigma \cup F_\sigma$.
Inv	The number of pairs of numbers of the sequence in which the first item is larger than the second in a data sequence.
N	Set of new nodes added into the tree by a update operation following the tree update template.
R	A sequence of Data-record removed by an SCX operation.
V	A sequence of Data-record that cannot be modified by other processes during an SCX operation.
ϕ	The golden ratio $\frac{1+\sqrt{5}}{2}$.
σ	Set of nodes visited by a update operation following the tree update template.
fld	A pointer pointing to the mutable field of a Data-record to be modified by an SCX operation.
new	The value to be stored in the field pointed to by fld.

parent	The node whose child pointer field is changed in the tree update template.
BST	Binary Search Tree.
BTS	Bit-Test-and-Set.
CAS	Compare-And-Set.
lc-cast	The unbalanced internal lock-based BST proposed by Ramachandran et al. [30].
lc-citr	the unbalanced internal lock-based BST proposed by Arbel et al. [2].
lc-davl	The concurrent AVL tree proposed by Drachsler et al. [12].
lf-chrm	The chromatic tree proposed by Brown et al. [7].
lf-chrm6	The chromatic tree proposed by Brown et al. [7] allowing at most three violations on each search path.
lf-ebst	The unbalanced external non-blocking BST proposed by Ellen et al. [15].
lf-ibst	The unbalanced internal non-blocking BST proposed by Ramachandran et al. [30].
lf-nbst	The unbalanced external non-blocking BST proposed by Natarajan et al. [24].
lf-ravl	The non-blocking ravl tree.
lf-ravl3	The non-blocking ravl tree allowing at most three violations on each search path.
LLX	Load-Linked Extended.

RCU Read Copy Update.

SCX Store-Condition Extended.

STM Software Transactional Memory.

Acknowledgements

I would like to express my sincere appreciation to my supervisor, Dr. Meng He, for his generous guidance and support during my research. This work would not have been possible without his valuable suggestions. It has been a pleasure working with him.

In addition, I am grateful to my thesis committee, Dr. Andrew Rau-Chaplin and Dr. Alex Brodsky, for their insightful comments and constructive criticisms.

Finally, I would like to give special thanks to my parents and friends for their support and encouragements.

Chapter 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 that store uniquely identifiable data items by keys to support the following operations: Get which returns the value stored in the node identified by a given search key or $NULL$ if the key is not in the tree, Insert which inserts a new item identified by a given key into the tree if the key is not in the tree, and Delete which removes an existing item identified by a given key from the tree if the key is in the tree. Lock-based BSTs are the most intuitive solutions and have been shown to be efficient [2, 4, 10, 12, 21, 29]. There is, however, a potential issue: if a process holding a lock on an object is halted by the operating system, all other processes requiring access to the same object will be prevented from making any further progress. Non-blocking (lock-free) BSTs have thus been proposed in recent years to overcome this limitation [7, 8, 13, 15, 22, 24, 30].

Despite the extensive work on lock-based and non-blocking BSTs, only a few support self-balancing [4, 7, 10, 12, 21]. 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 [26] 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, for experimental evaluations, almost all existing empirical studies [2, 4, 7, 12, 20, 22, 24, 29, 30] are performed using randomly generated data under uniform distribution. This approach, however, has some drawbacks. As mentioned in [2, 7, 26], such experimental settings favor BSTs without self-balancing greatly as they are balanced with high probability. Studies [26] have also shown that it is common that in real-world applications, keys of data elements in an access sequence are partially sorted (some degree of *presortedness*) instead of randomly. As an example, when entering student grades of a course into a database, the data are likely to be entered in the order of the student IDs or names. Thus, random data do not simulate these scenarios well. Therefore, 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 on the study of self-balancing BSTs in sequential settings. In particular, Sen and Tarjan [32] 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. Indeed, in [32], they highlighted an incident in which a company, in order to avoid the development time required to make each rebalancing case robust, relied on a solution that rebuilt the tree structure when the tree was higher than a threshold value. However, some time after the software product was deployed, it crashed and went offline for an extended period of time because of an issue in the rebuilding process. To provide developers 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. The number of rebalancing cases introduced is thus much fewer than other non-blocking self-balancing BSTs such as the non-blocking chromatic tree proposed by Brown et al. [7]. More precisely, it need only consider 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* ϕ

$= \frac{1+\sqrt{5}}{2}$ *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.*

This data structure is designed for asynchronous systems where shared memory locations can be accessed by multiple processes.

We conducted experimental studies to evaluate the performance of our solution by comparing it against other state-of-the-art concurrent BSTs. In these experiments, we also implemented a variant of the non-blocking ravl tree which rebalances the tree less frequently to reduce the total number of rebalancing steps. We first use randomly generated data under uniform distribution. Results show that our solution outperforms other concurrent self-balancing BSTs in every case. It is, however, outperformed by some concurrent BSTs without self-balancing. This is consistent with well-known facts in sequential settings: in theory, the expected heights of randomly built BSTs without self-balancing are $O(\log n)$ [9] where n is the number of nodes in the tree, and they outperform self-balancing BSTs in experimental studies with randomly generated data, as they avoid the overhead introduced in rebalancing processes [26]. Even though BSTs without self-balancing have advantages under these experimental settings, our solution still outperforms the RCU-based CITRUS tree proposed by Arbel et al. [2] significantly under operation sequences with a high update ratio, and has comparable performance to the non-blocking external tree proposed by Ellen et al. [15] under low contention level. We then conducted experiments using data sequences with different degrees of presortedness to simulate real-world applications. Experimental results show that our solution achieves the best performance in all cases 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 studies [26] have shown that, when implemented in system software products, it is very common

that BSTs without self-balancing are more than five times higher than self-balancing BSTs, we believe that our solution is the best candidate for many real-world applications.

To achieve these results, we first design a *sequential external ravl tree* in Section 4, which is a variant of the ravl tree proposed by Sen and Tarjan [32]. The original ravl tree is an *internal tree*, in which all nodes store values. The sequential external ravl tree is an *external tree*, in which only leaves store values, and internal nodes are used for routing only. We make this change to achieve maximum concurrency for non-blocking implementation to be designed later. As new conflict cases are introduced by these modifications, we carefully design new algorithms, prove correctness and bound the tree height. We then combine the sequential external ravl tree and the tree update template proposed by Brown et al. [7] to design the non-blocking ravl tree, and carefully prove all its properties summarized in Theorem 1. Finally, experimental results are given and analyzed in Section 6.

Chapter 2

Related Work

In this chapter, we introduce some state-of-the-art concurrent BSTs proposed recently. We first introduce the following terminologies uses throughout this thesis:

1. **Linearizability**: a linearizable operation should appear to take effect instantaneously at some point between its invocation and termination. Such a point is called a *linearization point*. In a linearizable data structure, the sequence of operation calls that appears to take effect should be consistent with the sequence of their linearization points executed sequentially. We can prove the linearizability of an operation by identifying the linearization point of its method calls.

2. **Lock-based**: a lock-based data structure uses locks for synchronization.

3. **Non-blocking** or **lock-free**: a non-blocking data structure is always guaranteed to make progress, and it will not be affected by failure or suspension of some threads.

4. **Wait-free**: in a wait-free data structure, each thread is guaranteed to make progress.

A number of lock-based and non-blocking BSTs have been designed to achieve different goals in concurrent systems, such as increasing overall throughput, improving memory efficiency, and adapting to contentions to reduce conflicts between processes. In the rest of the paper, we say that a BST is unbalanced if it does not support self-balancing.

2.1 Lock-Based BSTs

Bronson et al. [4] 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. Drachsler et al. [12] proposed a partially non-blocking 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 et al. [2] is an unbalanced internal BST offering wait-free GET operations and lock-based update operations. It allows concurrent update operations on BSTs based on *Read Copy Update* (RCU) synchronization and fine-grained locking. Ramachandran et al. [29] proposed the CASTLE tree, an unbalanced lock-based internal BST which locks edges instead of nodes to achieve higher concurrency.

2.2 Lock-Free BSTs

The first non-blocking BST that has been theoretically proven to be linearizable and non-blocking was proposed by Ellen et al. [15]. It is an unbalanced external BST which uses *Compare-And-Set* (CAS) to perform update operations. A CAS operation atomically compares the value stored at a memory location to a given value, and if these two values are equal, it stores a given new value at this memory location. In this non-blocking BST, each process *mark*s or *flag*s related nodes by storing extra information fields into these nodes to indicate its intended update operation. If a process intends to update some nodes that have been marked or flagged by some other processes, it will perform the update operation stored in these nodes first, and then retry its intended update operation. This way, all processes can make progress without waiting for other processes. Ellen et al. [13] also proposed a variant of [15],

which improves the performance by avoiding recursive helping. In this non-blocking BST, 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 internal non-blocking BST was proposed by Howley et al. [22], in which both processes performing GET and processes performing update operations help other processes by performing the update operations stored in the information fields of related nodes. In the unbalanced non-blocking external BST proposed by Natarajan et al. [24], each process performing update operations operates on edges by marking their end nodes. It implements *Bit-Test-and-Set* (BTS) to test if the end nodes of an edge have been marked, and CAS to perform update operations. Ramachandran et al. [30] proposed an unbalanced lock-free BST by combining ideas from [22] and [24].

Brown et al. [7] proposed a general technique for developing lock-free BSTs using non-blocking primitive operations [6]. 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. To further demonstrate the usefulness of the framework, they provided a non-blocking implementation of chromatic trees [25]. At any time during any execution, the height of a non-blocking chromatic tree is bounded by $O(\log n + \dot{c})$, where n is the number of nodes in the tree, and $\dot{c}$ is the number of active concurrent processes at this time.

Chapter 3

Preliminaries

In this chapter, we describe the previous results that are used in our solution.

3.1 LLX and SCX Operations

Non-blocking ravl trees implement primitive operations *Load-Linked Extended* (LLX) and *Store-Condition Extended* (SCX) proposed by Brown et al. [6] 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. A Data-record is *finalized* if it is removed from the data structure, and its mutable fields cannot be further changed afterwards.

A successful LLX operation performed on a Data-record r reads r's mutable fields and returns the values stored in these fields at the linearization point of this LLX operation. 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 LLX operation performed on a non-finalized Data-record r returns $fail$ if it is concurrent with any SCX operation that modifies r. An LLX operation performed on a finalized Data-record returns $finalized$. To perform an SCX operation, a process must first perform LLX operations on all Data-records in V, and the last LLX operation performed on each Data-record must not return $fail$ or $finalized$. Such LLX

operations are *linked* to the corresponding SCX operation. An SCX operation will fail if it is concurrent with any other SCX operation performed by another process that modifies the Data-records in V.

To implement LLX and SCX operations, two fields are added to each Data-record: a *marked* bit, which indicates if this Data-record has been finalized, and an $info$ pointer, which points to an *SCX-record* storing the following information of the last SCX operation performed on this Data-record:

1. V, R, fld, new: The required arguments passed to the corresponding SCX operation.

2. *old*: The value stored in the mutable field pointed to by fld before the corresponding SCX operation takes place.

3. *state*: The current status of the corresponding SCX operation. State *InProgress* indicates that this SCX is active; state *Committed* indicates that this SCX has completed; and state *Aborted* indicates that this SCX has failed.

4. $allFrozen$: *true* if all Data-records in R have been removed from the tree; $false$ otherwise.

5. $infoFields$: The list of pointers pointing to the $info$ fields of the set of Data-records in V.

An LLX operation on a Data-record r first reads r's *marked* bit and $info$ field to determine the status of r. If the *marked* bit is *true*, the LLX operation returns $finalized$. If the *marked* bit is $false$ and the state of r's $info$ field is not $InProgress$, the LLX operation returns the values of r's mutable fields. If the SCX operation associated with the SCX-record pointed to by r's $info$ fields is active, the LLX operation helps this SCX operation, and then returns $finalized$ if this SCX operation removes r, or $fail$ otherwise.

To perform an SCX operation, S, we first create an SCX-record, r_s, which contains all the information required to perform the desired update. We also set the state of r_s to be $InProgress$, and the $allFrozen$ bit of r_s to be $false$. Next, we perform a sequence of CAS operations attempting to store the address of r_s into the $info$ fields of all Data-records in V. If one of these CAS operation on a Data-record r in V fails, we perform one of the following:

1. If r's $info$ field points to r_s, some other processes must have helped S. In this case, we move on to the next Data-record in V.

2. If the $allFrozen$ field of r_s is $true$, some other processes must have completed S. In this case, S returns *success*.

3. In all other cases, S cannot be accomplished. We then set r_s's *state* to be *Aborted*, and S will return $fail$ afterwards.

If these CAS operations all succeed, we set the $allFrozen$ bit of r_s to be $true$. S then sets the *marked* bit of each Data-record in R to be $true$, and performs a CAS operation to store *new* into the mutable field pointed to by fld to update the tree. We the set the *state* field of r_s to be $Committed$. Finally, S returns *success*.

Lemma 2 shows the linearizability and non-blocking property of LLX and SCX operations.

Lemma 2 ([6]). *Successful LLX and SCX operations are linearizable. If LLX and SCX operations are invoked infinitely often, they succeed infinitely often, and are thus non-blocking.*

3.2 Tree Update Template

Non-blocking ravl trees use the template proposed by Brown et al. [7] to perform update operations. This template provides a framework to design non-blocking *down*

trees, which are directed acyclic graphs of indegree one. 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. Figure 3.1 gives an example of such an update operation. 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. Thus, the subgraph G_R induced by nodes in $R \cup F_R$ before the update is replaced by the subgraph G_N induced by nodes in $N \cup F_N$ after the update. G_R and G_N are both down trees. Let *parent* be the parent of the root node in G_R. The update operation modifies the corresponding child pointer of *parent* so that the pointer points to the root of G_N after the update.

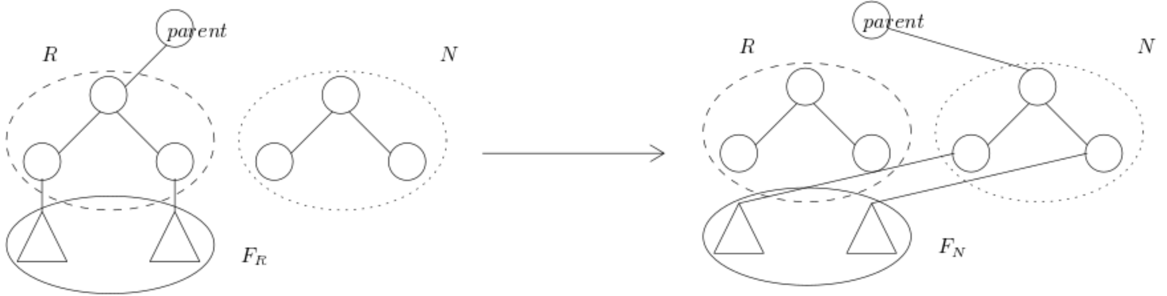


Figure 3.1: An example of a tree update operation following the template in [7].

Algorithm 1 gives the detailed implementation of the update template. An update operation *op* performing TEMPLATE takes an argument, *args*, which contains the information needed to perform the desired update. *op* traverses the down tree until it reaches *parent* in line 1. *op* then performs an LLX operation on *parent* and stores the values of the mutable and immutable fields of *parent* in r_p and r'_p in line 2 and line 5, respectively. If this LLX operation returns *fail* or *finalize*, *op* returns *fail* in line 4. Between line 7 and line 15, *op* performs an LLX operation on each node visited, and stores the values of its mutable fields and immutable fields in r_i and r'_i in line 8 and line 11, respectively. If any of these LLX operations returns *fail* or *finalized*, *op*

Algorithm 1 TEMPLATE($args$)[7]

1: Perform a top-down traversal starting from the root of the tree until $parent$ is reached
2: $r_p \leftarrow$ LLX($parent$)
3: **if** $r_p \in \{fail, finalized\}$ **then**
4:    return $fail$
5: $r'_p \leftarrow$ the immutable fields of $parent$
6: $i \leftarrow 0$
7: **while** $true$ **do**
8:    $r_i \leftarrow$ LLX(n_i)
9:    **if** $r_i \in \{fail, finalized\}$ **then**
10:       return $fail$
11:    $r'_i \leftarrow$ the immutable fields of n_i
12:    **if** CONDITIONMET($r_p, r'_p, r_0, r'_0, r_1, r'_1, \ldots, r_i, r'_i, args$) **then**
13:       break out of the loop
14:    $i \leftarrow i + 1$
15:    $n_i \leftarrow$ NEXTNODE($r_p, r'_p, r_0, r'_0, r_1, r'_1, \ldots, r_i, r'_i, args$)
16: **if** SCX(SCXARGUMENT($r_p, r'_p, r_0, r'_0, r_1, r'_1, \ldots, r_i, r'_i, args$) **then**
17:    return RESULT($r_p, r'_p, r_0, r'_0, r_1, r'_1, \ldots, r_i, r'_i, args$)
18: **else**
19:    return $fail$

returns $fail$ in line 10. If some conditions determined by a user-provided function named CONDITIONMET are met in line 12, *op* breaks out of the loop. Note that these conditions must be met eventually. *op* determines which node to visit next via a user-provided function named NEXTNODE in line 15. Let $\sigma = \{parent, n_0, n_1, \ldots\}$ be the sequence of nodes visited by *op*. We also define $F_\sigma = \{x | \text{parent of } x \in \sigma \text{ and } x \notin \sigma\}$ before the update, and the down tree G_σ to be the subgraph induced by $\sigma \cup F_\sigma$.

Finally, *op* updates the data structure via an SCX operation in line 16. If this

SCX operation succeeds and returns *true*, *op* returns the results computed by a user-provided function named RESULT in line 17. Otherwise, *op* returns $fail$ in line 19. The required arguments of the SCX operation, V, R, fld and *new*, are constructed by a user-provided function named SCXARGUMENT. Lemma 3 gives the requirements on these arguments, and update operations using this template are linearizable and non-blocking if these requirements are met.

Lemma 3 ([7]). *Consider a down tree structure 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. 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.

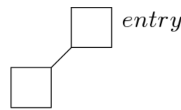
Chapter 4

Sequential External Ravl Trees

In this chapter, we describe the sequential external ravl tree, which is a variant of Sen and Tarjan's ravl tree [32]. We first introduce external ravl trees in Section 4.1 and describe how operations are performed on those trees in Section 4.2. We then bound the height of a sequential external ravl tree in Section 4.3.

4.1 The Structures of Sequential External Ravl Trees

(a) An empty sequential ravl tree.



(b) A non-empty sequential ravl tree.

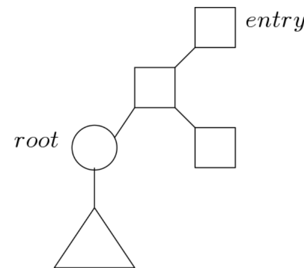


Figure 4.1: The structures of an empty and a non-empty sequential external ravl tree.

We first introduce the notation used throughout this chapter. Each node x in an external ravl tree maintains two child pointers, $x.left$ and $x.right$. 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. We define the height of x, $h(x)$, to be the length of the longest node-to-leaf path starting from x. The height of a tree rooted at node rt is then $h(rt)$. x contains key, value and rank fields, denoted by $x.k$, $x.v$ and $x.r$, respectively. If x is an internal node, $x.v = NULL$. The rank of x is used by rebalancing operations. As to be shown

later, we compute this value depending on the operation to be performed on x, and determine if rebalancing is required and which rebalancing step to take based on the rank difference among related nodes. 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 an *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.

To avoid special cases in the concurrent version of this data structure to be described in Chapter 5, we introduce the *entry* node, which is the entry point of an external ravl tree. Figure 4.1(a) illustrates the structure of an empty tree which contains an entry node with a single leaf left child. We call these two nodes *sentinel* nodes. In a non-empty tree, as shown in Figure 4.1(b), the leftmost grandchild of the entry node is the actual root of the external ravl tree, and its height is defined to be the height 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. We define the keys, values and ranks of the sentinel nodes to be ∞, $NULL$ and ∞, respectively. If a node is neither a missing node nor a sentinel node, we call it an *original node*. Note that in all figures of this thesis, we use squares to represent sentinel nodes, circles to represent original nodes and triangles to represent subtrees, and we do not show missing nodes.

4.2 Operations in Sequential External Ravl Trees

Sequential external ravl trees support Get operations by standard BST searches. To insert an item with key key and value $value$, we first search for key and get a leaf l and its parent p. If $l.k = key$, the Insert operation returns $false$. Otherwise, it replaces l with a newly created subtree composed of a node new and its two leaf children new_k and new_l as shown in Figure 4.2(a). We define $new.k = \max(l.k, key)$, $new.r = l.r$,

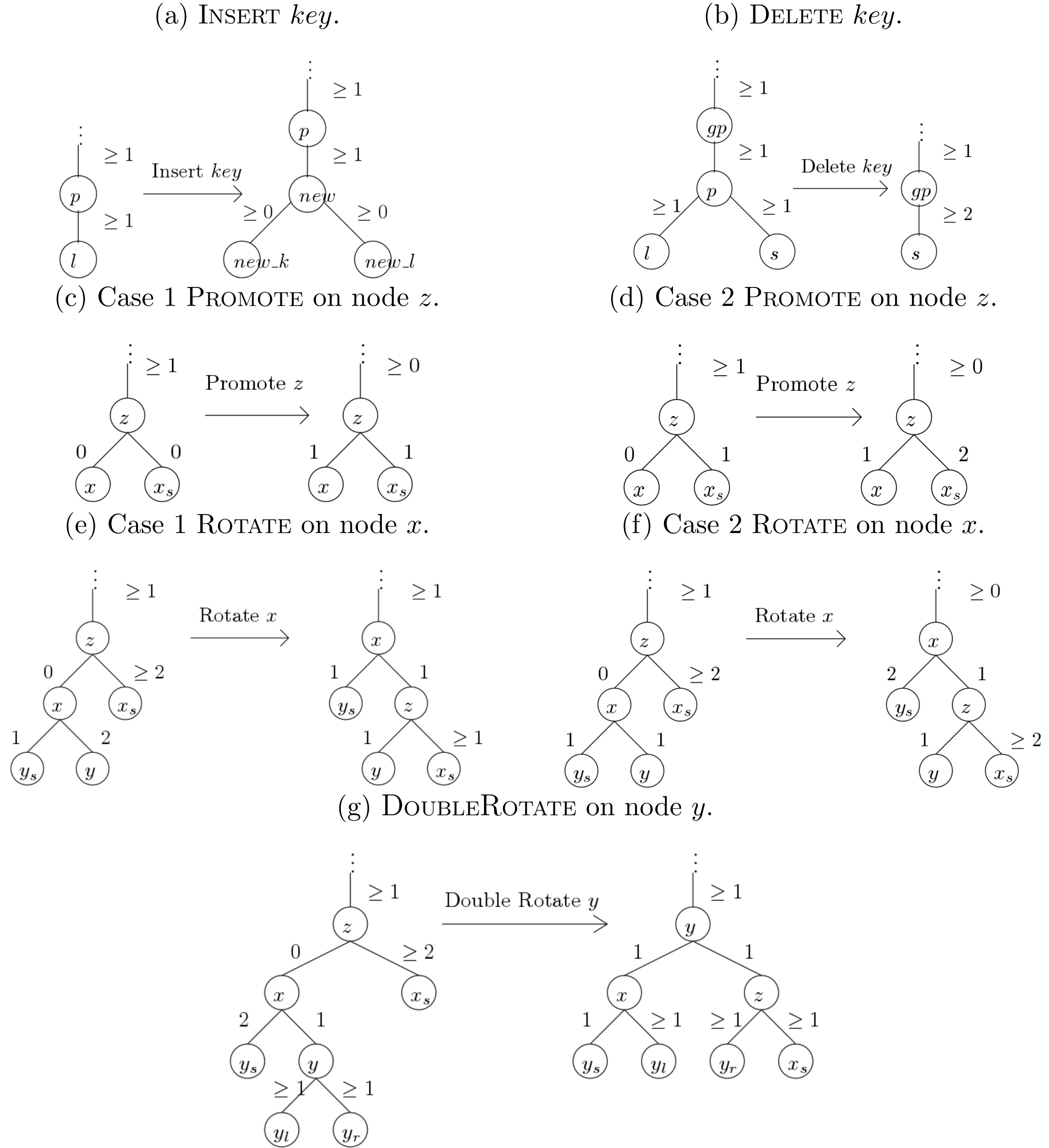


Figure 4.2: Operations in sequential external ravl trees. The number beside each edge indicates the rank difference between the parent and the child.

and $new.v = NULL$. new_k contains key and $value$, and we define $new_k.r$ to be 0. new_l stores $l.k$ and $l.v$, and we define $new_l.r$ to be ∞ if l was a sentinel node before the INSERT operation, or 0 otherwise. If $new_k.k < new.k$, we set new_k to be new's left child, and new_l to be new's right child, and vice versa. In the sequential case, a more efficient solution is to reuse node l and update its rank instead of creating a new

node *new_l*. However, to support concurrent updates in Chapter 5, it is necessary to create *new_l*. Thus, to be consistent with the description in Chapter 5, we do the same in sequential case. Violations might be created after inserting a new item into the tree, so the INSERT operation starts a rebalancing process (to be described later) after inserting the new item, and returns after the rebalancing process terminates and all violations have been resolved.

To delete an item identified by *key*, we search for *key* and get a leaf l, its parent p and its grandparent gp. If $l.k \neq key$, the DELETE operation returns $false$. Otherwise, it removes l and p, and sets l's sibling s to be gp's child by modifying the corresponding child pointer as shown in Figure 4.2(b). Violations cannot be created by DELETE operations. Since DELETE operations in sequential external ravl trees are not concurrent with INSERT operations, there is no violation in the tree during the execution of a DELETE operation.

Violations can be created by both INSERT and rebalancing operations. We resolve all potential violations via case 1 and 2 PROMOTE, case 1 and 2 ROTATE and DOUBLEROTATE operations illustrated in Figures 4.2(c) - (g). Let x be a 0-node in an external ravl tree, z be its parent, and x_s be its sibling. 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. To resolve the violation on x, we perform one of the following rebalance operations:

1. **Case** 1 **PROMOTE**: If z was a 0,0-node before this rebalancing step, we *promote* z by increasing its rank by 1 as illustrated in Figure 4.2(c). If the rank difference between z and its parent was greater than 1 before the promotion, no new violation is created in this step, and the rebalancing process terminates. Otherwise, z has become a 0-node, and we perform another rebalancing step to resolve the violation on z.

2. **Case** 2 **PROMOTE**: If z was a 0,1-node before this rebalancing step, we promote

z as illustrated in Figure 4.2(d). What happens next after this step is similar to a case 1 PROMOTE operation.

3. **Case** 1 **ROTATE**: If z was a 0,i-node before this rebalancing step, where $i \geq 2$, x was a 1,2-node, and y was a 2-node before the rotation, we perform a case 1 ROTATE operation on x as illustrated in Figure 4.2(e). We also *demote* z by reducing its rank by 1. The rebalancing process terminates after this step.

4. **Case** 2 **ROTATE**: If z was a 0,i-node before this rebalancing step, where $i \geq 2$, and x was a 1,1-node, we perform a case 2 ROTATE operation on x and promote x as illustrated in Figure 4.2(f). If the rank difference between z and its parent was greater than 1 before the rotation, no new violation is created in this step, and the rebalancing process terminates. Otherwise, x is still a 0-node after the rotation, and we perform another rebalancing step to resolve the violation on x.

5. **DOUBLEROTATE**: If z was a 0,i-node before this rebalancing step, where $i \geq 2$, x was a 1,2-node, and y was a 1-node, we perform a DOUBLEROTATE operation on y as shown in Figure 4.2(g). We set x to be y's left child and z to be y's right child after the rotation. To restore the BST property, we set y's previous left child and right children, y_l and y_r, to be x's right child and z's left child after the rotation, respectively. We also demote x and z and promote y. The rebalancing process terminates after this step.

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*. The rebalancing operations above explicitly cover all violation cases except the case in which the parent of a 0-node in a external ravl tree is a 0,i-node, where $i \geq 2$, and this 0-node is neither a 1,1-node, nor a 1,2-node. Lemma 4 proves that such a case does not exist in external ravl trees.

Lemma 4. *If the parent of a* 0*-node in an external ravl tree is a* 0*,*i*-node, where* $i \geq 2$*, then this* 0*-node must be a* 1*,*1*-node or a* 1*,*2*-node.*

Proof. Assume to the contrary that this lemma is not true. Initially in an empty tree, there is no violation. Let S be the first operation such that after S, there exists a 0-node x, which is neither a 1,1-node nor a 1,2-node, and its parent $p(x)$ is a 0,i-node, where $i \geq 2$. By Figure 4.2, only a Promote operation or a case 2 Rotate operation performed on x can make $p(x)$ a 0,i-node, where $i \geq 2$. If S is a case 1 Promote operation, x was a 0,0-node before S, and it becomes a 1,1-node after S, which is a contradiction. If S is a case 2 Promote operation, x was a 0,1-node before S, and it becomes a 1,2-node after S, which contradicts the assumption. If S is a case 2 Rotate operation, x was a 1,1-node before S, and it becomes a 1,2-node after S, which is a contradiction as well. This completes the proof. □

Now we have shown that our rebalancing operations can resolve all possible violation cases. Since we have defined the ranks of sentinel nodes to be ∞, the rank difference between the root of an external ravl tree and its parent is always positive. Thus, the rebalancing process in an external ravl tree can eventually terminate at its root via performing a Promote operation on the root, a Rotate operation on one of the root's children or a DoubleRotate operation on one of the root's grandchildren.

4.3 Bounding the Tree Height

To bound the heights of sequential ravl trees, we first show the relationship between the heights and ranks of original nodes in Lemma 5.

Lemma 5. *In an external ravl tree, the height of each original node is no greater than its rank.*

Proof. We prove this by induction on the heights of nodes. The height of any leaf node in an external ravl tree is 0, and its rank is greater than or equal to 0. Thus, the

statement holds in the base case. For the inductive case, assume that this property holds for all nodes with heights no greater than h, and we prove that this statement holds for any node u whose height is $h+1$. Let v and w be u's children. Without loss of generality, we assume that $height(w) \leq height(v) = h$. Therefore, $height(u) = h+1 = height(v)+1 \leq v.r+1 \leq max(v.r+1, w.r+1) \leq u.r$. Thus, the induction goes through. □

We are now ready to bound the tree height in Lemma 6:

Lemma 6. *For an external ravl tree built via a sequence of arbitrarily intermixed* Insert *and* Delete *operations from an empty tree, the height of the tree is bounded by* $\log_\phi(2m)$*, where* $\phi = \frac{1+\sqrt{5}}{2}$ *is the golden ratio and* m *is the total number of* Insert *operations executed.*

Proof. We apply a modified version of the potential functions used in [32] to prove this lemma. In our analysis, 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, we define its potential to be F_{k+3}.
2. If x is a 0,1-node, we define its potential to be F_{k+2}.
3. If x is a 0,i-node, where $i \geq 2$, we define its potential to be F_{k+1}.
4. If x is a 1,1-node, we define its potential to be F_k.
5. For all other cases, we define the potential of x to be 0.

We also define the potential of an external ravl tree to be the sum of the potentials of all its original nodes. Thus, the potential of the tree is always non-negative. The potential of an empty tree is 0. Insert, Delete, and rebalancing operations change the potential of the tree as follows:

1. **Insert**: Inserting a new item into the tree will increase the potential of the tree by at most 2. An Insert operation replaces a leaf l with a newly created node new with two leaf children, where $new.r = l.r$. By the potential functions defined above, the potentials of leaves are 0. Therefore, removing l will not change the potential of the tree. If $l.r$ was 0 before the insertion, new is a 0,0-node, whose rank is 0, and inserting new into the tree increases the potential of the tree by $F_3 = 2$. If $l.r$ was 1 before the insertion, new is a 1,1-node, whose rank is 1, and inserting new into the tree increases the potential of the tree by $F_1 = 1$. If $l.r$ was greater than 1 before the insertion, then this insertion does not change the potential of the tree.

2. **Delete**: Deleting an item identified by a given key from the tree does not increase the potential of the tree. Let l be the leaf identified by the given key, and p be its parent. The Delete operation removes l and p by making the corresponding child pointer of p's parent gp point to l's sibling s. Since the potentials of l and p are non-negative, removing them from the tree does not increase the potential of the tree. Since there is no violation in the tree during the execution of a Delete operation, p, s, l and p's sibling are not 0-nodes. Let the rank of p be k. If gp was a 1,1-node before the deletion, the potential of gp was F_{k+1}. Otherwise, the potential of gp was 0. Since $gp.r - p.r \geq 1$ and $p.r - s.r \geq 1$, the rank difference between gp and s is greater than or equal to 2. Thus, gp is not a 1,1-node after the update. Since a Delete operation cannot create violations, gp cannot have any 0-child after the deletion. Thus, the potential of gp becomes 0 after the deletion, and the total potential of the tree is not increased.

3. **Case** 1 **Promote**: Let z be the 0,0-node on which a case 1 Promote operation

is performed, and let k be its rank. If this case 1 PROMOTE operation is non-terminating, it does not change the potential of the tree; otherwise, it decreases the potential of the tree by at most F_{k+2}. After a case 1 PROMOTE operation, z becomes a 1,1-node, its rank becomes $k+1$, and its potential changes from F_{k+3} to F_{k+1}. Thus, the potential of z is decreased by F_{k+2}. Let $p(z)$ be z's parent. In the non-terminating case, 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}. In this case, the potential of $p(z)$ is increased by F_{k+2}, and the potential of the tree is not changed. Otherwise, this promotion does not change the potential of $p(z)$, and the potential of the tree is decreased by F_{k+2}. If z is the root of the tree, since the potential of the tree is the sum of the potentials of its original nodes only, it is decreased by F_{k+2}.

4. **Case** 2 **PROMOTE**: Let z be the 0,1-node on which a case 2 PROMOTE operation is performed, and let k be its rank. If this case 2 PROMOTE operation is non-terminating, it does not change the potential of the tree; otherwise, it decreases the potential of the tree by at most F_{k+2}. After a case 2 PROMOTE operation, z becomes a 1,2-node, its rank becomes $k+1$, and its potential changes from F_{k+2} to 0. Thus, the potential of z is decreased by F_{k+2}. Other than z, the only node whose potential can possibly be changed is p's parent $p(z)$, and using the same analysis in the case 1 PROMOTE operation, we claim the following: in

the the non-terminating 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 before the promotion, the potential of $p(z)$ is increased by F_{k+2}, and the potential of the tree is not changed; otherwise, the potential of $p(z)$ is not changed, and the potential of the tree is decreased by F_{k+2}. If z is the root of the tree, the potential of the tree is decreased by F_{k+2}.

5. **Case** 1 **Rotate**: A case 1 Rotate operation does not increase the potential of the tree. Let x be the 1,2-node on which this case 1 Rotate operation is performed, and let k be its rank. After this rotation, x becomes a 1,1-node, and its potential changes from 0 to F_k. Thus, the potential of x is increased by F_k. Let z be x's parent. If z was a 0,2-node before the rotation, its potential changes from F_{k+1} to F_{k-1}. Thus, the potential of z is decreased by F_k, and the potential of the tree is not changed. Otherwise, the potential of z changes from F_{k+1} to 0, and the potential of the tree is decreased.

6. **Case** 2 **Rotate**: Let x be the 1,1-node on which a case 2 Rotate operation is performed, and let k be its rank. If this case 2 Rotate operation is non-terminating, it does not change the potential of the tree; otherwise, it decreases the potential of the tree by at most F_{k+2}. Let z be x's parent, and without loss of generality, assume that x is z's left child. After this rotation, the potentials of x and z change from F_k to 0 and from F_{k+1} to 0, respectively. Thus, the potential of z is decreased by F_{k+2}. Other than z, the only node whose potential can possibly be changed is p's parent $p(z)$, and using the same analysis in the case 1 Promote operation, we claim the following: in the non-terminating 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 before the promotion, the potential of $p(z)$ is increased by F_{k+2}, and the potential of the tree is not

changed; otherwise, the potential of $p(z)$ is not changed, and the potential of the tree is decreased by F_{k+2}. If z is the root of the tree, the potential of the tree is decreased by F_{k+2}.

7. **DoubleRotate**: A DoubleRotate operation does not increase the potential of the tree. Let x be the parent of the node on which this DoubleRotate operation is performed. Then x was a 1,2 node before the rotation. Let z be x's parent, and without loss of generality, assume that x is z's left child, and y is x's right child. This DoubleRotate operation is performed on y. If y was a 1,1-node before the rotation, the potentials of x and y change from 0 to F_{k-1} and from F_{k-1} to F_k, respectively. If z was a 0,2-node before the rotation, the potential of z changes from F_{k+1} to F_{k-1}. Otherwise, the potential of z changes from F_{k+1} to 0. In either case, the potential of the tree is not increased. If y was not a 1,1-node before the rotation, the potential of y changes from 0 to F_k after the rotation. The potentials of x and z were 0 and F_{k+1} before the rotation, respectively. If y's left child was a 1-node before the rotation, the potentials of x and z change to F_{k-1} and 0, respectively. Otherwise, if y's right child was a 1-node before the rotation, the potentials of x and z change to 0 and at most F_{k-1}. If none of y's children was a 1-node before the rotation, the potentials of x and z become 0 after the rotation. Thus, the potential of the tree is not increased.

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 the external ravl tree, 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 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 $2m \geq F_{rt.r+3} > F_{rt.r+2} \geq \phi^{rt.r}$. Finally, by Lemma 5, $h(rt) \leq rt.r < \log_\phi 2m$. $\square$

Chapter 5

Non-Blocking Ravl Trees

In this chapter, we present the non-blocking ravl tree, which is a lock-free implementation of the sequential external ravl tree in concurrent settings. As in Chapter 4, we define the root of of a non-blocking ravl tree to be the root of the subtree formed by all original nodes in the tree. Also, we use the same definitions of V, fld, *old*, *new*, R, N, F_R and F_N from Section 3.2 in this chapter.

5.1 The Structures and Algorithms of Non-Blocking Ravl Trees

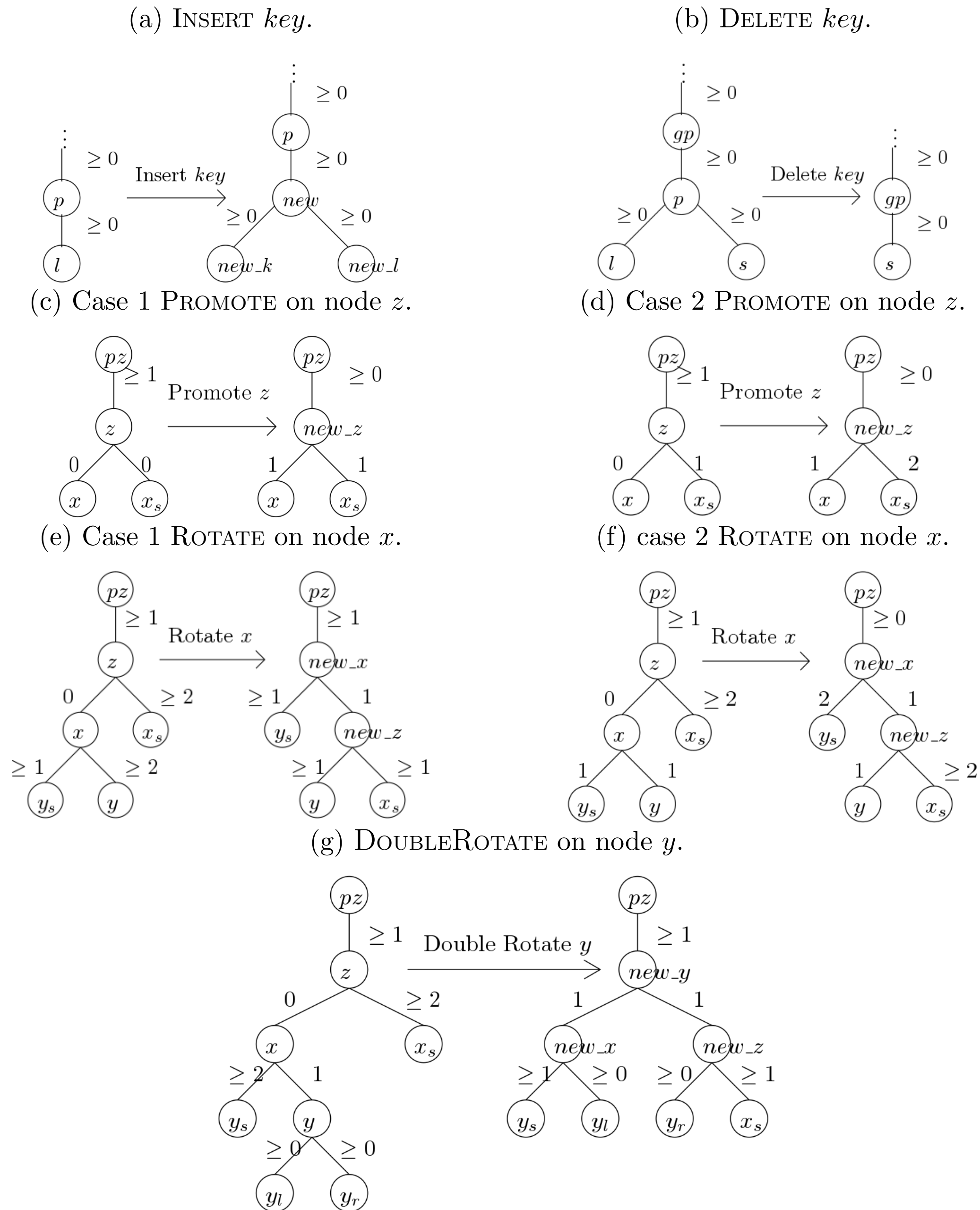
The composition and structure of a non-blocking ravl tree are the same as those of a sequential external ravl tree. Each node x in a non-blocking ravl tree is represented by a Data-record$< x.left, x.right, x.k, x.v, x.r >$, where $x.left$ and $x.right$ are mutable fields, and the other fields are immutable. We conceptually add missing nodes to newly created leaves as their children. Algorithm 2 describes the INIT operation, which initializes an empty non-blocking ravl tree consisting of the entry node with a single left child. Figure 5.1 illustrates the update operations in non-blocking ravl trees using the same notation in Figure 4.2.

Algorithm 2 INIT

1: $s \leftarrow$ pointer to a new Data-Record$< \text{missing}, \text{missing}, \infty, NULL, \infty >$

2: *entry* $\leftarrow$ pointer to a new Data-Record$< s, \text{missing}, \infty, NULL, \infty >$

Algorithm 3 presents the algorithm for a SEARCH operation in a non-blocking ravl tree, which performs a regular BST search starting from the entry node, and returns the last three nodes visited, n_0, n_1 and n_2, where n_2 is a leaf, n_1 is n_2's parent, and



(a) INSERT *key*. (b) DELETE *key*.

(c) Case 1 PROMOTE on node z. (d) Case 2 PROMOTE on node z.

(e) Case 1 ROTATE on node x. (f) case 2 ROTATE on node x.

(g) DOUBLEROTATE on node y.

Figure 5.1: Operations in non-blocking ravl trees.

in the leaf in line 13; otherwise, it returns $NULL$.

An INSERT operation described in Algorithm 5 attempts to insert a new item consisting of *key* and *value* into a non-blocking ravl tree. It returns *true* if the insertion succeeds, or *false* if *key* already exists in the dictionary that the non-blocking ravl tree represents. An INSERT operation first calls SEARCH(*key*), which

Algorithm 3 SEARCH(key)

3: $n_0 \leftarrow NULL$; $n_1 \leftarrow entry$; $n_2 \leftarrow entry.left$

4: **while** n_2 is internal **do**

5: $\quad n_0 \leftarrow n_1$; $n_1 \leftarrow n_2$

6: $\quad$ **if** $key < n_1.k$ **then**

7: $\quad\quad n_2 \leftarrow n_1.left$

8: $\quad$ **else**

9: $\quad\quad n_2 \leftarrow n_1.right$

10: return $< n_0, n_1, n_2 >$

Algorithm 4 GET(key)

11: $< -, -, l > \leftarrow$ SEARCH(key)

12: **if** $l.k = key$ **then**

13: $\quad$ return $l.v$

14: **else**

15: $\quad$ return $NULL$

Algorithm 5 INSERT($key, value$)

16: **repeat**

17: $\quad < -, p, l > \leftarrow$ SEARCH(key)

18: $\quad$ **if** $l.k = key$ **then**

19: $\quad\quad$ return $false$

20: $\quad result \leftarrow$ TRYINSERT($p, l, key, value$)

21: **until** $result \neq fail$

22: **if** $result$ **then**

23: $\quad$ CLEANUP(key)

24: return $true$

returns a leaf l and its parent p in line 17. If $l.k$ is equal to key, the INSERT operation returns $false$ in line 19, which indicates that it does not insert a new item into the tree. Otherwise, it invokes TRYINSERT($p, l, key, value$) (Algorithm 6), which carries out the

Algorithm 6 TryInsert$(p, l, key, value)$

25: **if** LLX$(p) \in \{fail, finalized\}$ **then**
26: return $fail$
27: **if** $l = p.left$ **then**
28: $fld \leftarrow \&p.left$
29: **else**
30: **if** $l = p.right$ **then**
31: $fld \leftarrow \&p.right$
32: **else**
33: return $fail$
34: **if** LLX$(l) \in \{fail, finalized\}$ **then**
35: return $fail$
36: $new_k \leftarrow$ pointer to a new Data-Record$<$ missing, missing, $key, value, 0 >$
37: **if** $l = entry.left$ **then**
38: $new_l \leftarrow$ pointer to a new Data-Record$<$ missing, missing, $l.k, l.v, \infty >$
39: **else**
40: $new_l \leftarrow$ pointer to a new Data-Record$<$ missing, missing, $l.k, l.v, 0 >$
41: **if** $key < l.k$ **then**
42: $new \leftarrow$ pointer to a new Data-Record$<$ new_k, new_l, $l.k$, $NULL$, $l.r >$
43: **else**
44: $new \leftarrow$ pointer to a new Data-Record$<$ new_l, new_k, key, $NULL$, $l.r >$
45: **if** SCX$(\{p, l\}, \{l\}, fld, new)$ **then**
46: return $(new.r = 0)$
47: **else**
48: return $fail$

actual insertion by following the tree update template summarized in Section 3.2. If the TryInsert step returns $fail$, the Insert operation retries this process from scratch.

A TRYINSERT operation performs the update shown in Figure 5.1(a). It first performs an LLX operation on p in line 25. If this LLX operation returns $fail$ or $finalized$, the TRYINSERT operation returns $fail$ in line 26. We then determine if l is p's left child or right child in line 27 or line 30, and we let pointer fld point to the correct child pointer field of p. If the structure of the related portion of the tree has been changed since the corresponding SEARCH operation returns, and l is not a child of p anymore when we perform the check above, the TRYINSERT operation returns $fail$ in line 33. Then we perform an LLX operation on l in line 34, and the TRYINSERT operation returns $fail$ if this LLX operation returns $fail$ or $finalized$. We create a new subtree rooted at the node pointed by pointer new from line 36 to line 44. Similar to an INSERT operation in a sequential external ravl tree, we define the key, value and rank of the root of the new subtree to be $\max(l.k, key)$, $NULL$ and $l.r$, respectively. We also create two new leaf nodes pointed to by pointer new_k and new_l, respectively. The key, value and rank of the leaf node pointed to by new_k are key, $value$ and 0, respectively. If l was not a sentinel node before the insertion, we define the key, value and rank of the leaf node pointed to by new_l to be $l.k$, $l.v$ and 0, respectively. Otherwise, we define the rank of this new leaf to be ∞. If $key < l.k$, we let the leaf node pointed to by new_k be the left child of the root of the new subtree, and the leaf node pointed to by new_l be the root's right child, and the other case is symmetric. We construct the SCX arguments in line 45, 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, the TRYINSERT operation returns $fail$. Otherwise, if the rank of the root of the newly inserted subtree is 0, the TRYINSERT operation returns $true$ to indicate that a new violation has been created after a successful insertion. If no new violation has been created, the TRYINSERT operation returns $false$.

The corresponding INSERT operation stores the returned value of its TRYINSERT subroutine in *result*. If *result* is *true*, a new violation has been created by this insertion, and the INSERT operation calls CLEANUP (Algorithm 7) to rebalance the tree. Finally, the INSERT operation returns *true* to indicate that a new item has been inserted into the tree.

Algorithm 7 CLEANUP(*key*)

49: **while** *true* **do**
50:     $gp \leftarrow NULL$; $p \leftarrow entry$; $l \leftarrow entry.left$; $l_s \leftarrow entry.right$
51:     **while** *true* **do**
52:         **if** l is a leaf **then**
53:             return
54:         **if** $key < l.k$ **then**
55:             $gp \leftarrow p$; $p \leftarrow l$; $l \leftarrow l.left$; $l_s \leftarrow l.right$
56:         **else**
57:             $gp \leftarrow p$; $p \leftarrow l$; $l \leftarrow l.right$; $l_s \leftarrow l.left$
58:         **if** $p.r = l.r + 1$ and $p.r = l_s.r$ **then**
59:             TRYREBALANCE(gp, p, l_s)
60:             break out of the inner loop
61:         **if** $p.r = l.r$ **then**
62:             TRYREBALANCE(gp, p, l)
63:             break out of the inner loop

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 50), 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 58), the CLEANUP operation calls TRYREBALANCE(gp, p, l_s) (Algorithm 8) to resolve the violation on l_s.

This step is required to avoid livelocks which will be explained in Section 5.3. Otherwise, if l is a 0-node (line 61), 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 52. 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 5.1(c) - (g). These figures illustrate case 1 and case 2 Promote, case 1 and case 2 Rotate and DoubleRotate operations in non-blocking ravl trees. They are very similar to the corresponding rebalancing operations in sequential external ravl trees, except in concurrent settings, we modify the rank of a node by replacing it with a newly created node with the updated rank, and we perform update operations by following the tree update template. In addition, since rebalancing operations are concurrent with Insert and Delete operations in concurrent settings, the rank difference between x and its children could be arbitrarily large, and y can have 0-children, while in sequential settings, x can only be a 1,1-node or a 1,2-node when z is a 0,i-node, where $i \geq 2$, and y cannot have any 0-child.

A TryRebalance operation attempts to resolve the violation on x. It first performs a sequence of LLX operations on pz, z and x in line 64, line 73 and line 89. If any of these LLX operations returns $fail$ or $finalized$, this TryRebalance operation returns without attempting to update the tree. We then determine if z is pz's left child or right child, and let pointer fld point to the correct child pointer field of pz in line 67 or line 70. If z is not a child of pz anymore when we perform the check above, the TryRebalance operation returns. We also perform the same check in line 75 and line 78 to verify that x is still a child of z and to store x's sibling in x_s.

Algorithm 8 TryRebalance(pz, z, x)

```
64: if LLX(pz) ∈ {fail, finalized} then
65:     return
66: if z = pz.left then
67:     fld ← &pz.left; z_s ← pz.right
68: else
69:     if z = pz.right then
70:         fld ← &pz.right; z_s ← pz.left
71:     else
72:         return
73: if LLX(z) ∈ {fail, finalized} then
74:     return
75: if x = z.left then
76:     x_s ← z.right
77: else
78:     if x = z.right then
79:         x_s ← z.left
80:     else
81:         return
82: if z.r ≠ x.r then
83:     return
84: if x_s.r = z.r or x_s.r = z.r − 1 then
85:     if pz.r = z.r + 1 and pz.r = z_s.r then
86:         return
87:     Promote(pz, z, x, x_s, fld)
```

Next, the TryRebalance operation examines the ranks of pz, z, z's sibling z_s, x and x's sibling x_s to determine which rebalancing step to take. If x is not a 0-node anymore (line 82), some other process must have resolved the violation on x, and

```
88: if z.r > x_s.r + 1 then
89:   if LLX(x) ∈ {fail, finalized} then
90:     return
91:   if x = z.left then
92:     y ← x.right; y_s ← x.left
93:   else
94:     y ← x.left; y_s ← x.right
95:   if x.r ≥ y.r + 2 or y is a missing node then
96:     Rotate1(pz, z, x, x_s, y, y_s, fld)
97:   if x.r = y.r + 1 and x.r = y_s.r + 1 then
98:     if pz.r = z.r + 1 and pz.r = z_s.r then
99:       return
100:    Rotate2(pz, z, x, x_s, y, y_s, fld)
101:  if x.r = y.r + 1 and x.r ≥ y_s.r + 2 then
102:    if LLX(y) ∈ {fail, finalized} then
103:      return
104:    DoubleRotate(pz, z, x, x_s, y, y_s, y.left, y.right, fld)
```

this TryRebalance operation returns without further changes. Otherwise, if z is a 0,0-node or 0,1-node (line 84), we check if promoting z will make pz a 0,0-node in line 85. If the condition in this line is *true*, the TryRebalance operation returns without attempting to modify the tree structure. This check is required for bounding the tree height in Section 5.4. If z can be promoted, the TryRebalance operation calls Promote (Algorithm 9) in line 87, which perform a case 1 or case 2 Promote operation on z as illustrated in Figure 5.1(c) or Figure 5.1(d), respectively. If z is a 0,i-node, where $i \geq 2$, there are three subcases. Without loss of generality, assume that x was z's left child. Let y be x's right child, and y_s be x's left child before the update. Before we decide which rebalancing step to take, we perform an LLX

operation on x, and the corresponding TryRebalance operation returns if this LLX operation returns $fail$ or $finalized$. The three subcases are as follows:

1. If $x.r \geq y.r{+}2$ or y is a missing node (line 95), the TryRebalance operation calls Rotate1 (Algorithm 10) in line 96, which performs a case 1 Rotate operation on x as illustrated in Figure 5.1(e).

2. If $x.r = y.r + 1$ and $x.r = y_s.r + 1$ (line 97), we perform the same check performed before calling Promote in line 98 to avoid making pz a 0,0-node. If the condition in this line is $false$, the TryRebalance operation calls Rotate2 (Algorithm 11) in line 100, which performs a case 2 Rotate operation on x as illustrated in Figure 5.1(f).

3. If $x.r = y.r + 1$ and $x.r \geq y_s.r + 2$ (line 101), the TryRebalance operation performs an LLX operation on y in line 102. If this LLX operation returns $fail$ or $finalized$, the corresponding TryRebalance operation returns. We then call DoubleRotate (Algorithm 12) in line 104, which performs a DoubleRotate operation on y as illustrated in Figure 5.1(g).

Algorithm 9 Promote(pz, z, x, x_s, fld)

105: **if** $x = z.left$ **then**
106: $\quad new_z \leftarrow$ pointer to a new Data-Record$< x, x_s, z.k, z.v, z.r + 1 >$
107: **else**
108: $\quad new_z \leftarrow$ pointer to a new Data-Record$< x_s, x, z.k, z.v, z.r + 1 >$
109: SCX($\{pz, z\}, \{z\}, fld, new_z$)

Algorithms 9 - 12 describe the implementations of case 1 and case 2 Promote, case 1 and case 2 Rotate and DoubleRotate operations in non-blocking ravl trees corresponding to update operations illustrated in Figures 5.1(c) - (g), respectively. All nodes in these algorithms correspond to the nodes with the same names in Figure 5.1.

Algorithm 10 ROTATE1($pz, z, x, x_s, y, y_s, fld$)

110: **if** $x = z.left$ **then**

111: $new_z \leftarrow$ pointer to a new Node-record$< y, x_s, z.k,\ z.v, z.r - 1 >$

112: $new_x \leftarrow$ pointer to a new Node-record$< y_s, new_z,\ x.k, x.v, x.r >$

113: **else**

114: $new_z \leftarrow$ pointer to a new Node-record$< x_s, y, z.k,\ z.v, z.r - 1 >$

115: $new_x \leftarrow$ pointer to a new Node-record$< new_z, y_s,\ x.k, x.v, x.r >$

116: SCX($\{pz, z, x\}, \{z, x\}, fld, new_x$)

Algorithm 11 ROTATE2($pz, z, x, x_s, y, y_s, fld$)

117: **if** $x = z.left$ **then**

118: $new_z \leftarrow$ pointer to a new Node-record$< y, x_s, z.k,\ z.v, z.r >$

119: $new_x \leftarrow$ pointer to a new Node-record$< y_s, new_z,\ x.k, x.v, x.r + 1 >$

120: **else**

121: $new_z \leftarrow$ pointer to a new Node-record$< x_s, y, z.k,\ z.v, z.r >$

122: $new_x \leftarrow$ pointer to a new Node-record$< new_z, y_s,\ x.k, x.v, x.r + 1 >$

123: SCX($\{pz, z, x\}, \{z, x\}, fld, new_x$)

Algorithm 12 DOUBLEROTATE($pz, z, x, x_s, y, y_s, y_l, y_r, fld$)

124: **if** $x = z.left$ **then**

125: $new_x \leftarrow$ pointer to a new Node-record$< y_s, y_l,\ x.k, x.v, x.r - 1 >$

126: $new_z \leftarrow$ pointer to a new Node-record$< y_r, x_s,\ z.k, z.v, z.r - 1 >$

127: $new_y \leftarrow$ pointer to a new Node-record$< new_x,\ new_z, y.k, y.v, y.r + 1 >$

128: **else**

129: $new_x \leftarrow$ pointer to a new Node-record$< y_r, y_s,\ x.k, x.v, x.r - 1 >$

130: $new_z \leftarrow$ pointer to a new Node-record$< x_s, y_l,\ z.k, z.v, z.r - 1 >$

131: $new_y \leftarrow$ pointer to a new Node-record$< new_z,\ new_x, y.k, y.v, y.r + 1 >$

132: SCX($\{pz, z, x, y\}, \{z, x, y\}, fld, new_y$)

A PROMOTE operation promotes z by replacing it with a newly created node pointed by pointer new_z (line 106 or line 108). The rank of this new node is $z.r + 1$. If x

was z's left child before the promotion, we make x new_z's left child, and x's sibling x_s becomes new_z's right child. The other case is symmetric. We construct SCX arguments in line 109, where we define $V = \{pz, z\}$, $R = \{z\}$ and $new = new_z$. We then perform an SCX operation attempting to atomically store new_z in the child field pointed to by pointer fld while finalizing z. Rotate1, Rotate2 and DoubleRotate operations in non-blocking ravl trees use the same strategy as Promote operations.

Algorithm 13 Delete(key)

133: **repeat**
134: $< gp, p, l > \leftarrow$ Search(key)
135: **if** $l.k \neq key$ **then**
136: return $false$
137: $result \leftarrow$ TryDelete(gp, p, l, key)
138: **until** $result \neq fail$
139: return $true$

Finally, Algorithm 13 describes a Delete operation in a non-blocking ravl tree, which attempts to remove key from the dictionary that the non-blocking ravl tree represents. It returns $true$ if the deletion succeeds, or $false$ if key does not exist in the dictionary. A Delete operation first calls Search(key), which returns a leaf l, its parent p and its grandparent gp in line 134. If $l.k$ is not equal to key, the Delete operation returns $false$ in line 136, which indicates that it does not remove any item from the tree. Otherwise, it then invokes TryDelete(gp, p, l, key) (Algorithm 14), which carries out the actual deletion by following the tree update template summarized in Section 3.2 as shown in Figure 5.1(b). If the TryDelete step returns $fail$, the Delete operations retries this process from scratch. As in a TryInsert operation, a TryDelete operation performs a sequence of LLX operations on gp, p and l, and it returns $fail$ if any of these LLX operations returns $fail$ or $finalized$. We also make sure that the related tree structure has not been modified by other processes

Algorithm 14 TryDelete(gp, p, l, key)

```
140: if LLX(gp) ∈ {fail, finalized} then
141:     return fail
142: if p = gp.left then
143:     fld ← &gp.left
144: else
145:     if p = gp.right then
146:         fld ← &gp.right
147:     else
148:         return fail
149: if LLX(p) ∈ {fail, finalized} then
150:     return fail
151: if l = p.left then
152:     s ← pointer to p.right
153: else
154:     if l = p.right then
155:         s ← pointer to p.left
156:     else
157:         return fail
158: if LLX(l) ∈ {fail, finalized} then
159:     return fail
160: if SCX({gp, p, l}, {p, l}, fld, s) then
161:     return success
162: else
163:     return fail
```

by verifying if p is a child of gp, and l is a child of p. We let pointer fld point to the left child pointer field of gp if p was gp's left child before the deletion (line 143); otherwise, fld points to the right child field of gp (line 146). We construct the SCX

arguments in line 160, where we define $V = \{gp, p, l\}$, $R = \{p, l\}$ and *new* to be s which is a pointer to l's sibling. The corresponding SCX operation attempts to atomically store s into the child field of gp pointed to by fld while finalizing p and l. If this SCX operation succeeds, the TryDelete operation returns *success*; otherwise, it returns $fail$. The corresponding Delete operation returns *true* after a successful TryDelete call.

5.2 Correctness of Non-Blocking Ravl Trees

We prove the correctness of non-blocking ravl trees by showing the following properties:

1. All violations can be resolved using rebalancing operations in non-blocking ravl trees.

2. A non-blocking ravl tree remains a BST at any time.

3. All operations in non-blocking ravl trees are linearizable.

We first prove that rebalancing operations in non-blocking ravl trees can resolve all possible violation cases. TryRebalance described in Algorithm 8 explicitly covers all possible violation cases except the case in which the 0-child of a 0,i-node, where $i \geq 2$, has at least one 0-child. Thus, it suffices to prove that such a case does not exist in concurrent settings, as shown in the following lemma.

Lemma 7. *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.*

Proof. Assume to the contrary that this lemma is not true. Initially in an empty tree, there is no violation. Let S be the first operation such that after S, there exists a 0-node that has at least one 0-child, and this 0-node's parent is a 0,i-node, where $i \geq 2$. We examine the following cases:

1. S is an INSERT operation which replaces a leaf l with a new subtree rooted at node new. Let p be l's parent before S. By the definition of S, S must make a node become a 0-node whose parent is a 0,i-node, where $i \geq 2$, and this node must also have at least one 0-child. By Algorithm 6 and Figure 5.1(a), this node can only be new, and it must be a 0-child after S is performed. S must also make p a 0,i-node, where $i \geq 2$. If l was not a 0-node before S, new cannot be p's 0-child after S, which is a contradiction. Therefore, l was a 0-node before S. There are two subcases:

   (a) If $l.r$ was not 0 before S, new does not have any 0-child after S, which contradicts the assumption.

   (b) If $l.r$ was 0 before S, new is a 0,0-node after S. However, since l was a 0-node before S, $p.r$ is 0, and p cannot be a 0,i-node, where $i \geq 2$, which is a contradiction.

2. S is a DELETE operation which removes leaf l and its parent p. Let gp be p's parent, and s be l's sibling before S. In this case, by Algorithm 14, Figure 5.1(b) and the definition of S, S must make gp a 0,i-node, where $i \geq 2$. In addition, after S is performed, s becomes gp's 0-child, and s has at least one 0-child. There are two subcases:

   (a) If p was a 0-node before S, s must not have been a 0-node before S. Otherwise, gp would have been a 0,i-node, where $i \geq 2$, and p would have had a 0-child before S, which contradicts the assumption that S is the first operation that creates such a case. Thus, S cannot make s gp's 0-child, which contradict the assumption.

   (b) If p was not a 0-child before S, since $gp.r > p.r$ and $p.r \geq s.r$, S cannot make s gp's 0-child, which is a contradiction.

3. If S is a rebalancing operation, by Figures 5.1(c) - (g) and Algorithms 9 - 12, S cannot be a case 1 ROTATE operation or a DOUBLEROTATE operation. We consider the following subcases:

   (a) S is a PROMOTE operation which promotes a node z by replacing it with a newly created node new_z whose rank is $z.r + 1$. Let pz be z's parent before S. In this case, by Figure 5.1(c), Figure 5.1(d) and the definition of S, S must make pz a 0,i-node, where $i \geq 2$. In addition, after S is performed, new_z becomes pz's 0-child. However, new_z does not have any 0-child after S, which is a contradiction.

   (b) S is a case 2 ROTATE operation performed on a 0-node x. Let z be x's parent, new_x and new_z be the newly created nodes that replace x and z, respectively, and pz be x's grandparent before S. In this case, by Figure 5.1(f) and the definition of S, S must make pz a 0,i-node, where $i \geq 2$. In addition, after S is performed, new_x becomes pz's 0-child after S. However, new_x does not have any 0-child after S, which contradicts the assumption.

Thus, no such S exists. This completes the proof. □

Using the conclusion in Lemma 7, we have now shown that rebalancing operations in non-blocking ravl trees can resolve all possible violation cases. We next prove that a non-blocking ravl tree remains a BST at any time during any execution, which guarantees that BST operations are always performed correctly. Lemma 8 shows that all original nodes in a non-blocking ravl tree always form a subtree rooted at the leftmost grandchild of the entry node, which is the actual root of a non-blocking ravl tree.

Lemma 8. *An empty non-blocking ravl tree always consists of an entry node with a single left child, which are the sentinel nodes of the tree. In a non-empty tree,*

all original nodes in the tree form a subtree that is always rooted at the left-most grandchild of the entry node. In this case, the sentinel nodes of the tree are the entry node, its left child and its left child's right child. The key, value and rank of sentinel nodes are always ∞*,* $NULL$ *and* ∞*, respectively.*

Proof. We prove this by induction on the number of SCX operations that have been performed successfully in a non-blocking ravl tree. The base case holds for an empty tree. In the inductive case, assume that this lemma holds before a successful SCX operation S. If S inserts a new key into an empty tree, by Algorithm 6, S replaces the left child of the entry node with a newly created subtree rooted at node x that has two leaf children. $x.left$ is the only original node in the tree. It contains the newly inserted key, and it is the leftmost grandchild of the entry node. In addition, the key, value and rank of x and $x.right$ are ∞, $NULL$ and ∞, respectively. Thus, inserting the first item into an empty tree does not not affect the correctness of this lemma. If S removes the last original node x from the tree, by Algorithm 14, S replaces x and its parent with x's sibling, whose key, value and rank are ∞, $NULL$ and ∞, respectively. Thus, deleting the last original node from the tree does not affect the correctness of this lemma. By Algorithm 6 and Algorithm 14, if S is performed by an INSERT operation in a non-empty tree or a DELETE operation that does not remove the last original node from the tree, S does not change the sentinel nodes and thus, it does not affect the correctness of this lemma either. If S is performed by a rebalancing step, since the ranks of the sentinel nodes are ∞, the root will never become a violation. Thus, S can only be performed on original nodes, and thus it does not affect the correctness of this lemma. This completes the proof for the inductive case. □

Next, we show that SEARCH operations in non-blocking ravl trees are always performed correctly. We define the *search path* [7] for a key k to be the root-to-leaf path formed by the original nodes that a SEARCH operation for k visits as if this operation

finishes instantaneously at the time when it is invoked. In other words, if we take a snapshot of the tree structure at time t, the search path for k at t is the root-to-leaf path that a BST search for k follows in the snapshot. Using this definition, Lemma 9 proves the correctness of SEARCH operations by showing that if a node was on the search path for some key at some time, it remains on the search path for this key at any time later as long as it is still in the tree.

Lemma 9. *In a non-blocking ravl tree, if at time t_1, an original node v is on the search path for a key k, and at a later time t_2, v is still in the tree, then v is still on the search path for k at t_2.*

Proof. From the update operations defined in concurrent settings, we claim that the nodes removed from a non-blocking ravl tree are finalized and not reachable from the tree. Since all nodes added to a tree by SCX operations are newly created, removed nodes cannot be added back to the tree. Since v is in a non-blocking ravl tree at time t_1 and t_2, it must be in the tree at anytime between t_1 and t_2.

Let S be a successful SCX operation performed between t_1 and t_2, which changes a child of some node from *old_child* to *new_child*. If v was not a descendant of *old_child* before S, v is not affected by S, and thus is not removed from the search path for k. If v was a descendant of *old_child* before S, since v is not removed from the tree, v is either a member of F_R, or a descendant of some nodes in F_R. In the latter case, let f be v's lowest ancestor in F_R, and f was on the search path for k at the time immediately before S is performed. Following all possible tree transformations described in Figure 5.1, if a node in F_R was on the search path for some keys before S, it remains in the tree in set F_N, and it is on the search path for the same set of keys after S. Thus, in the former case, where v was a member of F_R before S, v is still on the search path for k after S. In the latter case, all nodes on the path from f up to v are not affected by S. Therefore, after S is performed, f is still on the search

path for k after S, so is v. To conclude, if v was on the search path for k at time t_1, SCX operations cannot remove it from the search path for k at any time after t_1 as long as v is still in the tree. Since no other operation can change the tree structure and invalidate this statement, we have completed the proof. □

We have shown that if some original nodes are in a non-blocking ravl tree at some time, they preserve the BST property at any time later as long as they are still in the tree. It remains to show that nodes newly added into the tree by SCX operations preserve the BST property as well. Lemma 10 completes the proof and shows that the BST property is preserved in a non-blocking ravl tree at any time.

Lemma 10. *The subtree composed of all original nodes in a non-blocking ravl tree is a BST at any time.*

Proof. We prove this by induction on the number of SCX operations that have been performed successfully. In the base case, the tree is empty, thus the statement is true. In the inductive case, assume that the statement holds before a successful SCX operation S. If S is performed by an Insert operation that inserts the first key into an empty tree or a Delete operation that removes the last key from the tree, then, by Lemma 8, this statement holds after S. Otherwise, we consider the following cases:

1. If S is not an SCX step performed by an Insert operations, by Algorithms 9 - 12 and 14, F_R was not empty before S, and $F_N = F_R$ after S. In addition, nodes in set N preserve the BST property, i.e., for each node x in N, $x.left.k < x.k \leq x.right.k$. By Lemma 9, if nodes in F_R and their descendants were on the search paths for some set of keys, s_k, they remain in the tree and are on the search paths for keys in s_k after S. Since nodes in N are ancestors of nodes in F_N, the nodes in F_N are on the search paths for keys in s_k after S as well, and thus the BST property is preserved.

2. If S is an SCX step invoked by an Insert operation that does not insert the first key into an empty tree, let k be the inserted key, l be the leaf node removed by S, and *new* be the root of the newly added subtree. By Algorithm 6, l is the leaf node on the search path for k when the corresponding Search operation executes line 7 or line 9. Since S succeeds, S is not concurrent with any other SCX operation performed by another process that modifies the child pointer fields of p. Therefore, l is still a child of p and is in the tree immediately before S, and by Lemma 9, it is still on the search path for k. After S replaces l with *new*, *new* is on the search path for k as well. By Algorithm 6, the BST property preserves in the newly created subtree. Thus, the BST property of the non-blocking ravl tree is preserved after S.

This completes the proof for the inductive case. □

We then show that non-blocking ravl trees are linearizable in the next two lemmas. Ideally, we want to define the linearization point of a Search operation to be the time when it reaches a leaf node, which is in the tree when it is visited. It is, however, non-trivial to show that such a definition works. This is because a Search operation for a given key in non-blocking ravl trees does not check the status of nodes visited, and by the time when it reaches a leaf in line 7 or line 9, or when it returns the leaf node in line 10, this leaf might not be in the tree anymore. Thus, it is important to prove that this leaf was in the tree and on the search path for the given key at some time earlier during this Search operation, and thus it is the correct node to return in line 10. If the leaf returned by a Search operation is a sentinel node, by Algorithm 3, the tree is empty, and it is the correct node to return. Lemma 11 shows that Search operations in non-empty non-blocking ravl trees returns the correct leaf as well.

Lemma 11. *If an original node x is visited by a Search operation for a given key, x was on the search path for this key at some earlier time during this Search operation,*

no matter whether x is still in the tree when it is visited or not. Thus, the leaf node visited and returned by the Search *operation is correct.*

Proof. We prove this by induction on the number of nodes visited during the Search operation. For the base case, since the Search operation starts from the root of a non-blocking ravl tree, this claim holds. For the inductive case, assume that this claim holds when the Search operation visits node v on the search path for k, and v was in the tree at time t before it is visited. Without loss of generality, assume that $k < v.k$, and let v' be v's left child when the Search operation visits v' at time t'. To prove that this statement holds for the inductive case, we show that there exists a time t'' between t and t' when v' is on the search path for k.

If at t', v is in the tree, we define t'' to be the time immediately before the Search operation reads v' from $v.left$. This happens after the Search operation visits v at time t and before it visits v' at time t'. Since v is in the tree at t and t', by Lemma 9, v is on search path for k at time t''. Finally, since v' is chosen to be the next node to visit because $k < v.k$, v' is on the search path for k at t''.

If at t', v is not in the tree, then we consider the following two subcases: 1) if at time t, v' is v's left child, then v' was in the tree before t; 2) otherwise, there exists a successful SCX operation that set v's left child to be v' after t. In either case, there exists a time at or after t when v' is in the tree, and it is v's left child. Since v is not in the tree at time t', there exists a successful SCX operation S between t and t' that removes v from the tree. We define t'' to be the time immediately before S. Since v is still in the tree at time t'', by Lemma 9, it is still on the search path for k. Since we define t'' to be the time immediately before S, the child pointers of v cannot be changed by any other SCX operation between t'' and S. After S, the child pointers of v cannot be further changed as v has been finalized. Since v' is v's left child at time t' after S, it must have been v's left child at time t''. To conclude, at time t'', v is on the search path to k, v' is the left child of v, and v' is chosen to be the next

node to visit because $k < v.k$. Thus v' is on the search path to k at this time. This completes the proof for the inductive case. □

Finally, we prove the linearizability of non-blocking ravl trees by defining the linearization points of their operations in Lemma 12.

Lemma 12. *A non-blocking ravl tree is linearizable, and the linearization points of its operations are defined as follows:*

1. *A* Search *operation is linearized when* n_2 *reaches a leaf in line 7 or line 9.*
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.*

Proof. By Lemma 11, it can be concluded that the leaf that a Search operation stores in n_2 at its linearization point was in the tree and on the search path for the search key at some time during this Search operation. Thus, this Search operation returns the correct leaf node in line 10, regardless if this node is still in the tree or not.

A Get operation in a non-blocking ravl tree is linearized when its Search step reaches a leaf in line 7 or line 9. By Lemma 10, since the original nodes in the non-blocking ravl tree always form a BST, if the leaf contains the search key, this key is

in the dictionary that the tree represents at this time; otherwise, the key is not in the dictionary. In either case, this GET operation always returns the correct result in line 13 or line 15. The same analysis applies to INSERT and DELETE operations that return *false* without performing SCX operations.

If an INSERT operation or a DELETE operation returns *true* after perform an successful SCX operation, we claim that it follows the tree update template summarized in Section 3.2. From the pseudocode in Algorithms 6, 9 - 12 and 14, we observe that all the update operations in non-blocking ravl trees performing SCX operations meet the requirements specified in Lemma 3. In addition, observe that rebalancing operations do not change keys stored in the dictionary that a non-blocking ravl tree represents. Thus, by Lemma 3, an INSERT operation or a DELETE operation that returns *true* is linearized at the linearization point of the SCX step performed by the corresponding TRYINSERT or TRYDELETE step, respectively. □

5.3 Progress Properties of Non-Blocking Ravl Trees

We now prove the progress properties for non-blocking ravl trees. We first prove that if update operations are invoked infinitely often, they follow the tree update template summarized in Section 3.2 infinitely often. It is obvious that this statement holds for TRYINSERT operations, TRYDELETE operations and TRYREBALANCE operations that do not return in line 86 or line 99. It remains to show that if TRYREBALANCE is invoked infinitely often, conditions in line 85 or line 98 (Algorithm 8) for these invocations will eventually return *false*. Thus, TRYREBALANCE follows the tree update template infinitely often.

Lemma 13. *If TRYREBALANCE is invoked infinitely often, it follows the tree update template infinitely often.*

Proof. Assume to the contrary that the lemma is not true. To derive a contradiction, assume that TryRebalance is invoked infinitely often, but only a finite number of these invocations follow the tree update template. Since TryInsert operations, TryDelete operations and TryRebalance operations that do not return in line 86 or line 99 (Algorithm 8) always follow the tree update template, by Lemma 3, if they are invoked infinitely often, they succeed infinitely often. Since an Insert operation or a Delete operation can perform one successful TryInsert or TryDelete, respectively, and according to the assumption, only a finite number of tree updates are performed by TryRebalance operations, there exists a time T when only TryRebalance operations that return in line 86 or line 99 without following the tree update template remain active, while all other operations have terminated. Since these TryRebalance operations do not perform any SCX operation, they cannot change the tree structure. Thus the tree structure remains stable after T.

Consider the following set of nodes S: node x is in this set if and only if, after T, there exists a process for which the invoked TryRebalance keeps returning in line 86 or line 99 because it cannot perform a Promote operation on x, or cannot perform a case 2 Rotate operation on x's 0-child. Among the nodes in S, let z be the node with the minimum depth, and let P be the process whose calls to TryRebalance keep failing to perform a Promote operation on z or to perform a case 2 Rotate operation on z's 0-child. Let pz be z's parent, z_s be z's sibling. Since the conditions in line 86 or line 99 are true from all nodes in S, $pz.r = z.r + 1$ and $pz.r = z_s.r$. After an invoked TryRebalance operation returns in line 86 or line 99, P will start another attempt in line 50 (Algorithm 7). Since the tree is stable, P will follow the same search path and visit the same set of nodes on the search path as in the last attempt. Thus, when P visits pz and z again, it will discover that pz is a 0,1-node, and z_s is pz's 0-child in line 58. P will then invoke TryRebalance to resolve the violation on z_s by promoting pz. Since z is the node with the minimum depth among

the set of nodes in S, pz is not in S. Therefore, this time the TryRebalance invoked by P will not return in line 86 or line 99, and it will follow the tree update template. This contradicts the assumption that only TryRebalance operations that return in line 86 or line 99 without following the tree update template are active after T. □

Finally, we show that non-blocking ravl trees are non-blocking in the next lemma:

Lemma 14. *All operations in non-blocking ravl trees are non-blocking.*

Proof. Assume to the contrary that the lemma is not true. 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. We now discuss the following subcases:

If none of these operations is an Insert operation or a Delete operation, no SCX operation is performed after T_1, and the tree structure cannot be changed. Thus, Get operations will eventually terminate, which contradicts the assumption.

If some of these operations are Insert or Delete operations, then TryInsert, TryDelete and TryRebalance are repeatedly invoked. Based on the assumption, Lemma 3 and Lemma 13, these operations are invoked infinitely often, and an infinite number of these invocations are successful. An Insert operation can only perform one successful TryInsert operation before it terminates or starts the rebalancing process. Thus, at most one successful TryInsert operation is performed by each active process after T_1. After a successful TryDelete operation, the corresponding Delete operation will terminate after a constant number of steps. Thus, by the definition of T_1, all Delete operations have terminated before T_1. Therefore, there must be an infinite number of successful TryRebalance operations after T_1.

Let T_2 be the time when all successful TryInsert operations have terminated. After T_2, the tree can only be modified by TryRebalance operations. Let m be the

number of INSERT calls in total. As described in Algorithm 5, an INSERT operation can add at most 2 violations into the tree. Thus, at most $2m$ violations are added by INSERT operations. Each successful terminating PROMOTE operation, case 1 ROTATE operation, terminating case 2 ROTATE operation and DOUBLEROTATE operation reduce the number of violations by 1. Thus, the total number of successful invocations of these operations are at most $2m$. Therefore, there must be an infinite number of successful non-terminating operations that are either PROMOTE operations or case 2 ROTATE operations after T_2.

Let T_3 be the time when all other operations have terminated, and there are infinitely many invocations of non-terminating operations that are either PROMOTE operations or case 2 ROTATE operations. Consider a process P which attempts to resolve a violation x via a PROMOTE operation or a case 2 ROTATE operations. In either case, as illustrated in Figure 5.1(c), Figure 5.1(d) and Figure 5.1(f), such transformation will remove the violation x from the tree, and replace x's previous parent with a new violation x'. P will then repeat this process on x'. Eventually, P will make the root of the tree a 0-parent. Since the rank difference between root and its parent is always positive, to resolve the violation on the root's 0-child, P can only perform a terminating PROMOTE operation, a case 1 ROTATE operation, a terminating case 2 ROTATE operation or a DOUBLEROTATE operation. However, this contradicts the assumption that all these update operations have terminated before T_3. This completes the proof. □

5.4 Bounding the Tree Height

As in Section 4.3, we define the height of a non-blocking ravl tree to be the height of the subtree composed of all original nodes in the tree. We take the following steps to bound the tree height of a non-blocking ravl tree T:

1. We bound the height of tree T' build via eliminating all violations in T.

2. We bound the number of violating edges on each root-to-leaf path in T.

3. We show the relationship between the height of T and the height of T'.

4. Using the conclusion above, we bound the height of T.

We first bound the height of a balanced non-blocking ravl tree after all violations have been resolved using the same potential functions defined in Lemma 6. Note that these potential functions will not apply if a rebalancing step creates a new 0,0-node. By Figures 5.1(c) - (g) and Algorithms 9 - 12, only a non-terminating operation that is either a PROMOTE operation or a case 2 ROTATE operation can create a new violation. The next lemma shows that these two operations cannot create 0,0-nodes.

Lemma 15. *A non-terminating operation that is either a* PROMOTE *operation or a case* 2 ROTATE *operation cannot create a new* 0,0*-node.*

Proof. Assume to the contrary that this lemma is not true. Initially in an empty tree, there is no violation. Let S be the first SCX step performed by a non-terminating operation that is either a PROMOTE operation or a case 2 ROTATE operation that creates a new 0,0-node. Let x be the violation that S resolves, whose parent was z before the transformation. Let pz be z's parent, and z_s be z's sibling before S. In either case, by Figure 5.1(c), Figure 5.1(d) and Figure 5.1(f), S replaces z with a new node whose rank is $z.r+1$, and makes pz a 0-parent. Based on the assumption, pz is the new 0,0-node that S creates. Therefore, before S, pz was a 0,1-node where $pz.r = z.r+1$ and $pz.r = z_s.r$. However, in this case, the conditions in line 85 or line 98 would have been *true* before S, and the corresponding TRYREBALANCE operation would have returned in line 86 or line 99. As a result, S cannot be performed. This contradicts the assumption. □

From Figure 5.1(g), the node y on which a DoubleRotate is performed can have 0-children. The next lemma shows that y can have at most one 0-child in such a case.

Lemma 16. *Consider a DoubleRotate operation as shown in Figure 5.1(g). The node y on which this operation is performed on can have at most one 0-child.*

Proof. Assume to the contrary that y is a 0,0-node, and we are going to perform a DoubleRotate on y. By Lemma 15 and Figure 5.1, only Insert operations can create new 0,0-nodes. Thus, the last update operation that modifies y's rank is an Insert operation, which makes y a 0,0-node. However, in this case, the rank of x is 1 since y is a 1-node, and the rank difference between x and y_s cannot be greater than or equal to 2. Thus, the corresponding TryRebalance will not call DoubleRotate to resolve the violation on x, which contradicts the assumption that the DoubleRotate is performed on y. □

We are now ready to bound the height of the non-blocking ravl tree after all violations have been resolved as follows:

Lemma 17. *For a non-blocking ravl tree built via a sequence of arbitrarily intermixed Insert and Delete operations from an empty tree, the height of the tree after all violations have been resolved is bounded by $\log_\phi(2m)$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio and m is the number of Insert operations that have successfully inserted new keys into the tree.*

Proof. We use the same potential functions defined in Lemma 6. The analysis for Insert operations in non-blocking ravl trees is the same as in sequential ravl trees, where an Insert operations increases the potential of the tree by at most 2. In concurrent settings, we replace the existing nodes with newly created nodes with updated ranks, and we use SCX operations to update the tree structure; while in sequential settings, we directly modify the ranks and child pointers of related nodes.

These differences do not affect the potential analysis. The reasons are as follows: 1) replacing an existing node with a new node with updated rank has the same impact on the potential of the tree as directly changing the rank of the existing node; 2) the tree structure and rank differences between nodes are not depend on how the tree is updated, either via SCX operations or modifying child pointers directly. For the same reason, the analysis for Promote operations and case 2 Rotate operations in non-blocking ravl trees are the same as in sequential ravl trees, which are summarized as follows:

1. A non-terminating operation that is either a Promote operation or a case 2 Rotate operation does not change the potential of the tree.

2. A terminating Promote operation or a terminating case 2 Rotate operation performed on a node whose rank was k before this operation decreases the potential of the tree by at most F_{k+2}. If either of these two operations changes the rank of the root of the tree from k to $k+1$, the potential of the tree is decreased by F_{k+2}.

We now show the impact of Delete, case 1 Rotate and DoubleRotate operations on the potential of the tree in concurrent settings.

1. **Delete**: Deleting an item identified by a given key from the tree does not increase the potential of the tree. Let l be the leaf identified by the given key, and p be its parent. The Delete operation removes l and p by making the corresponding child pointer of p's parent gp point to l's sibling s. Since the potentials of l and p are non-negative, removing them from the tree does not increase the potential of the tree. By Figure 5.1(b), we observe the following cases: 1) if both p and s are 1-nodes, the potential of gp is decreased; 2) if p and s were both 0-nodes, or p was not a 0-node before the deletion, the potential of

gp does not change; 3) for all other cases, the potential of gp is decreased. In either case, the total potential of the tree is not increased.

2. **Case** 1 **Rotate**: A case 1 Rotate operation does not increase the potential of the tree. Let x be the node on which this case 1 Rotate operation is performed, and let k be its rank. Let z be x's parent before the rotation, and without loss of generality, assume that x is z's left child. The potentials of x and z were 0 and F_{k+1} before the rotation, respectively. Let new_x and new_z be the newly added nodes that replace x and z, respectively. The potential of new_x is at most F_k (it is F_k if x was a 1,2-node before rotation), and the potential of new_z is at most F_{k-1} (it is F_{k-1} if z was a 0,2-node and x was a 1,2-node before rotation) after the rotation. Thus, the potential of the tree cannot be increased.

3. **DoubleRotate**: A DoubleRotate operation does not increase the potential of the tree. Let x be the parent of the node y on which this DoubleRotate operation is performed. Let k be x's rank. Let z be x's parent, and without loss of generality, assume that x is z's left child, and y is x's right child. Let new_x, new_y and new_z be the newly added nodes that replace x, y and z, respectively. The potentials of x and z were 0 and F_{k+1} before the rotation, respectively. If y was a 1,1-node before the rotation, the potential of y was F_{k-1}. In this case, the potentials of new_x, new_y and new_z are at most F_{k-1} (F_{k-1} if x was a 1,2-node before rotation), F_k and at most F_{k-1} (F_{k-1} if z was a 0,2-node before the rotation), respectively. Thus, the potential of the tree cannot be increased in this case. If y was not a 1,1-node, and it did not have any 0-child before the rotation, the potential of y was 0. If one of y's children was a 1-node before the rotation, there are two subcases: 1) if y's left child was the 1-node, the potentials of new_x, new_y and new_z are at most F_{k-1} (F_{k-1} if x was a 1,2-node before rotation), F_k and 0, respectively; 2) if y's right child

was the 1-node, the potentials of new_x, new_y and new_z are 0, F_k and at most F_{k-1} (F_{k-1} if z was a 0,2 node before the rotation), respectively. In either case, the potential of the tree cannot be increased. If neither of y's children was a 1-node before the rotation, the potentials of new_x, new_y and new_z are 0, F_k and 0, respectively. The potential of the tree cannot be increased in this case, either.

By Lemma 16, y cannot be a 0,0-node before the rotation. If y had exactly one 0-child before the rotation, we first discuss the case in which y's left child was the 0-child. If y was a 0,1-node before the rotation, the potential of y was F_{k+1}. In this case, the potentials of new_x, new_y and new_z are at most F_{k+1} (if x was a 1,2-node before rotation), F_k and at most F_{k-1} (if z was a 0,2-node before the rotation), respectively. Thus, the potential of the tree cannot be increased in this case. If y was a 0,i-node before the rotation, where $i \geq 2$, the potential of y was F_k. In this case, the potentials of new_x, new_y and new_z are at most F_{k+1} (if x was a 1,2-node before rotation), F_k and 0, respectively. Thus, the potential of the tree cannot be increased in this case. The analysis for the case in which y's right child was the 0-child is similar, and this case cannot increase the potential of the tree.

As analyzed above, the impact of each update operation on the potential of a non-blocking ravl tree is exactly the same as in sequential settings. The rest of the analysis is the same as in Lemma 6, and the height of the tree after all violations have been resolved is bounded by $\log_\phi(2m)$. □

Next, we borrow the idea from [7] to bound the number of 0-parents in a non-blocking ravl tree. As described in Algorithm 5, an Insert operation can create at most one 0-parent after inserting a new key into a non-blocking ravl tree, and it will not terminate until its call to Cleanup returns. A successful rebalancing step invoked

by the Cleanup operation does not increase the number of 0-parents. Intuitively, each 0-parent has an injective mapping to a process performing an Insert operation that has already inserted the new key, but its Cleanup call has not yet returned. We say that such a process is in its *cleanup phase* [7]. Thus, the number of 0-parents in the tree is bounded by the number of processes that are in their cleanup phases. Lemma 18 and Lemma 19 formally define this injective mapping and prove the correctness of this statement.

Lemma 18. *If a node in a non-blocking ravl tree is not a* 0*-node at some time, it can never become a* 0*-node at any time in the future. Otherwise, if a node is a* 0*-node at some time, it has always been a* 0*-node since it was first added to the tree.*

Proof. To prove the first statement, assume to the contrary that this statement is not true. To derive a contradiction, consider a node v that is not a 0-node at some time, but it becomes a 0-node at some later time. Since the rank fields are immutable, there exists a successful SCX operation S such that after S, v becomes a 0-node. By Algorithms 6, 9 - 12 and 14, we observe that no such SCX operation exists, which contradicts the assumption. The correctness of the second statement follows directly from the first. □

Lemma 19. *The number of* 0*-parents in a non-blocking ravl tree is bounded by the number of processes that are in their cleanup phases.*

Proof. Given a process P, we define $location(P)$ to be the local variable l if P is performing the inner loop of Cleanup (between line 51 and line 63 in Algorithm 7). If P performs a successful SCX step, we define $location(P)$ to be the root of the tree after this SCX step. If P is executing instructions outside of the inner loop, we define $location(P)$ to be the root of the tree. We prove this lemma by arguing that at any time during in a non-blocking ravl tree, there exists an injection, ρ, which maps each

0-parent x to a process $\rho(x)$ that is in its cleanup phase, and ρ always satisfies the following invariants for each 0-parent x in the tree:

1. $\rho(x)$ is in its cleanup phase;

2. x is on the search path from the root of the tree to the leaf, $k_{\rho(x)}$, which contains the key inserted by $\rho(x)$;

3. One of the following three statements holds:

   (a) x is on the search path from $location(\rho(x))$ to $k_{\rho(x)}$; or

   (b) $location(\rho(x))$ is a 0-node; or

   (c) $location(\rho(x))$ is a 0-parent which has been finalized, and it was on the search path for $k_{\rho(x)}$ before it is removed from the tree.

We prove this claim by induction on the number of steps that have been carried out. The base case holds for an empty tree. In the inductive case, assume that there exists a mapping function ρ that satisfies all the invariants above immediately before a process P carries out a step S. We show that there exists a mapping function ρ' such that the invariants above are still satisfied immediate after S. We discuss all possible S that can affect the claim as follows:

1. If S is the execution of line 52 where $location(P)$ is a leaf, P will return after S, and we define $\rho' = \rho$ immediately after S. In this case, P did not satisfy invariant 3(a) or invariant 3(c) before S as $location(P)$ was a leaf.

   We now show that P did not satisfy invariant 3(b) before S, i.e., $location(P)$ was not a 0-node before S. If S is carried out in the first iteration of CLEANUP's inner loop, $location(P)$ is the left child of the entry node. In this case, the tree is empty, and there is no 0-node in the tree. Therefore, this statement is true. Otherwise, assume to the contrary that $location(P)$ was a 0-node before

S. By Lemma 18, $location(P)$ has been a 0-node since it was first added to the tree. Since in the previous iteration of CLEANUP's inner loop when $location(P)$ is stored in l, it has already been in this data structure, $location(P)$ was a 0-node at this time as well. However, in this case, P would have called TRYREBALANCE instead. As a result, $location(P)$ would have been removed from the tree, and it is no longer reachable when S is carried out. Therefore, $location(P)$ cannot be not a 0-node before S.

As analyzed above, no violation can be mapped to P under ρ. Thus, after S, $\rho' = \rho$ is still an injection satisfying invariant 1, 2 and 3.

2. If S is the execution of line 60 or line 63, we define $\rho' = \rho$ immediately after S. S does not affect any 0-parent mapped to a process that is not P under ρ. If there was a 0-parent x where $\rho(x) = P$ before S, then, since $location(P)$ becomes the root of the tree immediately after S, $\rho'(x)$ satisfies invariant 1, 2 and 3(a). Thus, $\rho' = \rho$ is still an injection satisfying invariant 1, 2 and 3.

3. If S is the execution of line 55, where S changes $location(P)$ from node v to $v.left$, we define $\rho' = \rho$ immediately after S. S does not affect any 0-parent mapped to a process that is not P under ρ. If there was a 0-parent x where $\rho(x) = P$ before S, $\rho'(x)$ satisfies invariants 1 and 2 after S.

   We first show that P did not satisfy invariant 3(b) before S, i.e., v was not a 0-node. If S is carried out in the first iteration of CLEANUP's inner loop, v is the left child of the entry node. Therefore, this statement is true. Otherwise, assume to the contrary that v was a 0-node before S. By Lemma 18, v has been a 0-node since it was first added to the tree. Therefore, v was a 0-node in the previous iteration. However, in this case, the corresponding CLEANUP operation would have called TRYREBALANCE to resolve the violation on v. As a result, v would have been removed from the tree, and it is no longer reachable

in the iteration of CLEANUP's inner loop when S is carried out. Thus, v was not a 0-node before S.

If P satisfied invariant 3(a) immediately before S, there are two subcases: 1) if $x = v$ before S, $\rho'(x)$ satisfies invariants 3(b) after S; 2) if $x \neq v$ before S, $\rho'(x)$ satisfies invariants 3(a) after S. If P satisfied invariant 3(c) immediately before S, $\rho'(x)$ satisfies invariants 3(b) after S. Thus, $\rho' = \rho$ is still an injection satisfying invariant 1, 2 and 3.

4. If S is the execution of line 57, where S changes $location(P)$ from node v to $v.right$, we define $\rho' = \rho$ immediately after S. The analysis for this case is similar to that for the previous case, and $\rho' = \rho$ is still an injection satisfying invariant 1, 2 and 3.

5. If S is a successful SCX operation, we define ρ' for each 0-parent. For an existing 0-parent x that is still in the tree after S, we define $\rho'(x) = \rho(x)$. S does not affect these 0-parents. Next, consider an existing 0-parent x that is removed from the tree by S. If S is invoked by a terminating PROMOTE operation, a case 1 ROTATE operation, a terminating case 2 ROTATE operation, it removes x from the tree without adding a new 0-parent. Thus, we need not define $\rho'(x)$. If S is invoked by a TRYDELETE operation, it does not add any new 0-parent into the tree, and it removes at most one existing 0-parent from the tree. Thus, in this case, if an existing 0-parent x is removed from the tree by S, we need not define $\rho'(x)$ either. If a new 0-parent is created by S, we define its mapped process as follows:

   (a) If S is an SCX operation performed by an INSERT operation which adds a new 0-parent, new_p, into the tree, we define $\rho'(new_p)$ to be P. Invariant 1 holds in this case since P has just finished inserting a new key, and its

call to CLEANUP has not yet returned, i.e., P is in its cleanup phase. Since new_p is the root of the newly added subtree, which is the parent of the leaf containing the key inserted by P, it is on the search path for $k_{\rho'(new_p)}$. Thus, invariant 2 is satisfied. Since P has not yet entered the inner loop of CLEANUP, $location(P)$ is the root of the tree. Thus, invariant 3(a) is satisfied. Thus, ρ' is still an injection satisfying invariant 1, 2 and 3.

(b) If S is an SCX operation performed by a non-terminating PROMOTE operation on a 0-parent p, S makes p's previous parent, new_p, a new 0-parent. In this case, we define $\rho'(new_p) = \rho(p)$. If $P = \rho(p)$, since S has created a new violation, P will continue its rebalancing process. Otherwise, $\rho(p)$ is not affected. In either case, $\rho'(new_p)$ satisfies invariant 1. Since $\rho(p)$ satisfies invariant 2 immediately before S, p is on the search path for $k_{\rho(p)}$. Thus, new_p is on the search path for $k_{\rho'(new_p)}$ after S, which satisfies invariant 2. Since $location(\rho'(new_p))$ is the root of the tree after S, invariant 3(a) is always satisfied after S. Thus, ρ' is still an injection satisfying invariant 1, 2 and 3.

(c) If S is an SCX operation performed by a non-terminating case 2 ROTATE operation on the 0-child of an existing 0-parent p, S makes p's previous parent, new_p, a new 0-parent. In this case, we define $\rho'(new_p) = \rho(p)$. The rest of the analysis for this case is similar to the previous case, and ρ' is still an injection satisfying invariant 1, 2 and 3.

(d) If S is an SCX operation performed by a DOUBLEROTATE operation, we have two subcases. Let x be the 0-node that S resolves. Let z be x's parent. Then, z is a 0,i-node, where $i \geq 2$. 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. As shown in Figure 5.1(g), we replace x, y and z by newly created

node new_x, new_y and new_z, respectively. By Lemma 16, y can only have at most 1 0-child. We consider the following:

i. If y does not have any 0-child, then S removes the 0-parent z from the tree without adding a new one. Thus, we need not define $\rho'(z)$, and ρ' is still an injection satisfying invariant 1, 2 and 3.

ii. If y has one 0-child, we first consider the case that y's left child, y_l, is the 0-child. In this case, S removes 0-parents z and y and makes new_x a new 0-parent. In this case, we define $\rho'(new_x) = \rho(y)$, and we need not define $\rho'(z)$. Similar to the analysis in case 5(c), $\rho'(new_x)$ satisfies invariant 1 and 3(a) after S. In addition, since y_l is on the search path to $k_{\rho(y)}$, and y_l is a child of new_x, new_x is on the search path to $k_{\rho'(new_x)}$, which satisfies invariant 2. The analysis for the case that y's right child is the 0-child is similar. Thus, ρ' is still an injection satisfying invariant 1, 2 and 3.

We have now shown that after S, the mapping function ρ' still satisfies the invariants. This completes the proof for the inductive case. □

Using the conclusions in Lemma 17 and Lemma 19, we are now ready to bound the height of a non-blocking ravl tree in the next two lemmas:

Lemma 20. *Consider a non-blocking ravl tree T rooted at rt which contains n nodes and c 0-parents. Let T' be the balanced ravl tree constructed by eliminating all violations in T, and let rt' be its root. Let $h(rt)$ and $h(rt')$ be the heights of T and T', respectively. Then $h(rt) \leq h(rt') + c$.*

Proof. For a node v in a non-blocking ravl tree, we defined its weight height $wh(v)$ as follows:

1. If v is a leaf, $wh(v) = 0$.

2. If $v.left$ is not a 0-node and $v.right$ is a 0-node, $wh(v) = max(wh(v.left) + 1, wh(v.right))$.

3. If $v.left$ is a 0-node and $v.right$ is not a 0-node, $wh(v) = max(wh(v.left), wh(v.right) + 1)$.

4. If both of v's children are 0-nodes, $wh(v) = max(wh(v.left), wh(v.right))$.

5. Otherwise, $wh(v) = max(wh(v.left) + 1, wh(v.right) + 1)$.

We define the weight heights of T and T' to be $wh(rt)$ and $wh(rt')$, respectively. Using these definitions, we prove the following:

1. $h(rt) \leq wh(rt) + c$: $h(rt)$ equals to the length of the longest root-to-leaf path in T, which is the sum of number of non-violating edges and the number of violating edges on this path. As defined in weight height functions, each root-to-leaf path has at most $wh(rt)$ non-violating edges. Also, the number of violating edges on each root-to-leaf path equals to the number of 0-parents on that path, which is no large than c. Thus, this inequality holds.

2. $wh(rt) \leq wh(rt')$: Consider a rebalancing step S performed in T. Let $wh(rt)'$ and $wh(rt)''$ be the weight heights of T immediately before and after S, respectively. If S is a terminating Promote operation, a case 1 Rotate operation, a terminating case 2 Rotate operation or a DoubleRotate operation, it increases $wh(rt)'$ by at most 1 by eliminating one violating edge and adding a new non-violating edge on the same root-to-leaf path. If S is a non-terminating Promote operation or a non-terminating case 2 Rotate operation, it replaces an existing violating edge with a newly created one on the same root-to-leaf path and thus, $wh(rt)'$ is not changed. In either case, $wh(rt)' \leq wh(rt)''$. Assume that k rebalancing steps are performed to transform T to T'. The initial weight height of T is $wh(rt)$. After the ith rebalancing step, we define the weight height of T to be

$wh(rt)_i$. Therefore, $wh(rt) \leq wh(rt)_1 \leq wh(rt)_2 \leq \ldots \leq wh(rt)_n = wh(rt')$. The inequality holds.

3. $wh(rt') = h(rt')$: Since no violation exists in T', all edges on each root-to-leaf path in T' are non-violating. Thus, $wh(rt') = h(rt')$.

As analyzed above, $h(rt) \leq wh(rt) + c \leq wh(rt') + c = h(rt') + c$. The inequality holds. □

Lemma 21. *For a non-blocking ravl tree built via a sequence of arbitrarily intermixed* Insert *and* Delete *operations from an empty tree, at any time t during the execution, the height of the tree is bounded by $\log_\phi(2m_t) + c_t$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio, m_t is the number of* Insert *operations that have successfully inserted new keys into the tree by t and c_t is the number of* Insert *operations that are in their cleanup phases at t.*

Proof. Let T be such a non-blocking ravl tree, and we bound the height of T at time t. We define rt, T' and rt' the same way as in Lemma 20. By Lemma 17, the height of T', $h(rt')$, is less than $\log_\phi(2m_t)$. Moreover, by Lemma 19, the number of 0-parents in T is at most c_t. Finally, by Lemma 20, $h(rt) < \log_\phi(2m_t) + c_t$. This completes the proof. □

Chapter 6

Experimental Evaluation

In this chapter, we conducted experimental studies to evaluate the performance of the non-blocking ravl tree by comparing it against other state-of-the-art lock-free and lock-based BSTs.

6.1 Compared Data Structures

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. [7] which is a lock-free relaxed red-black tree. It uses the tree update template summarized in Section 3.2.

2. **lf-chrm6**, a variant of lf-chrm in which the rebalancing process is only invoked by an Insert or a Delete operation if the number of violations on the corresponding search path exceeds six [7]. 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. [12] which supports wait-free Get operations and lock-based update operations.

4. **lf-nbst**, the unbalanced external non-blocking BST proposed by Natarajan et al. [24] which operates on edges instead of nodes.

5. **lf-ebst**, the unbalanced external non-blocking BST proposed by Ellen et al. [15] which uses separate objects for process coordination.

6. **lf-ibst**, the unbalanced internal non-blocking BST proposed by Ramachandran et al. [30].

7. **lc-cast**, the unbalanced internal lock-based BST proposed by Ramachandran et al. [30].

8. **lc-citr**, the unbalanced internal lock-based BST proposed by Arbel et al. [2] which is based on RCU synchronization.

Borrowing the idea from lf-chrm6, we also implemented a variant of the non-blocking ravl tree in which CLEANUP is only invoked by INSERT if the number of violations on the corresponding search path exceeds three. We call this variant **lf-ravl3**. We allow at most three violations on search paths since experiments showed that the tree achieved the best performance under this setting. We did not consider other concurrent BSTs [4, 21, 22] as they were outperformed by some other BSTs compared here in previous studies.

6.2 Implementation Details

The original source code for lf-chrm [5], lf-ebst [14] and lc-davl [11] 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 [19, 23, 27, 28], except that, to ensure fairness in experimental studies, we made some minor changes to all source code so that they perform atomic CAS operations using the APIs provided by `libatomic_ops` [3]. We also used `jemalloc` [18] for memory allocations to achieve optimal results. For lock-based data structures, we used mutex locks and APIs provided by `pthread`.

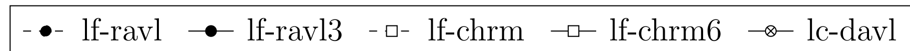
We used the framework *Synchrobench* developed by Gramoli et al. [20] to test the performance. We made some necessary modifications to support more user settings and to use the `gsl` [1] library to generate synthetic data sets, since it provides stable and thread-safe random number generation functions.

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.

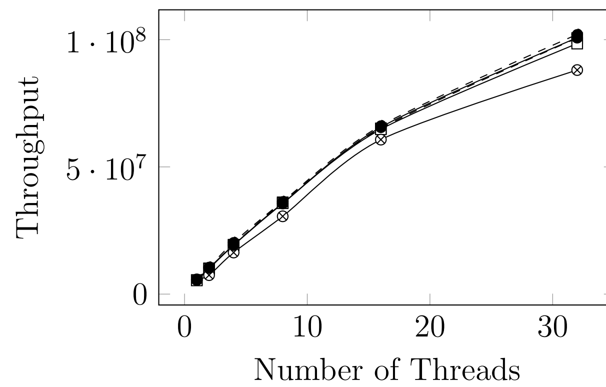
6.3 Random Data Set

In this section, we conduct experiments using random numbers as operation keys. All keys are positive integers within a user-specified range generated under uniform distribution. We used key ranges $(0, 2 \times 10^4]$, $(0, 2 \times 10^5]$ and $(0, 2 \times 10^6]$ to test different contention levels. For example, under a smaller key range, conflicts are more likely to occur among processes. We also used difference operation mixes. An operation mix xr-yi-zd represents x% Get operations, y% Insert operations and z% Delete operations. Similar to the experimental studies in [12], we used operation mixes 90r-9i-1d, 70r-20i-10d and 50r-25i-25d for read-dominant, mixed and write-dominant operation sequences, respectively. 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, and we computed its average throughput (the number of operations that the BST finishes per second). We ran 5 trials for each concurrent BST, and we used the average of these trials as the final result. To ensure stable performance, before each trial, we prefilled each data structure using randomly generated keys until 50% of the keys in the key range are inserted into the tree. We did not measure the performance of concurrent BSTs during the prefilling phase.

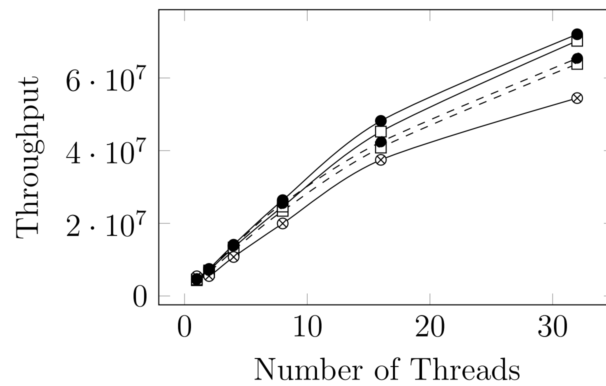
Figures 6.1 - 6.3 show the experimental results comparing lf-ravl and lf-ravl3



(a) Throughput Under Operation Mix: 90r-9i-1d



(b) Throughput Under Operation Mix: 70r-20i-10d



(c) Throughput Under Operation Mix: 50r-25i-25d

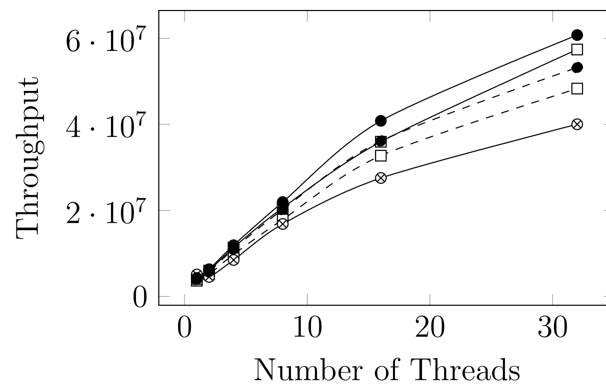
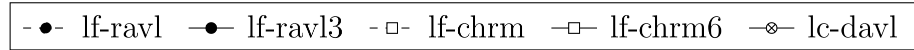
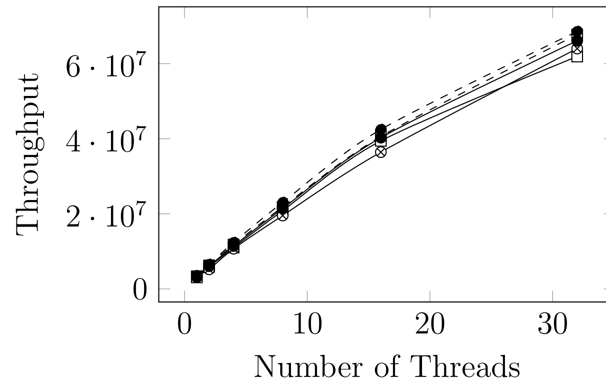


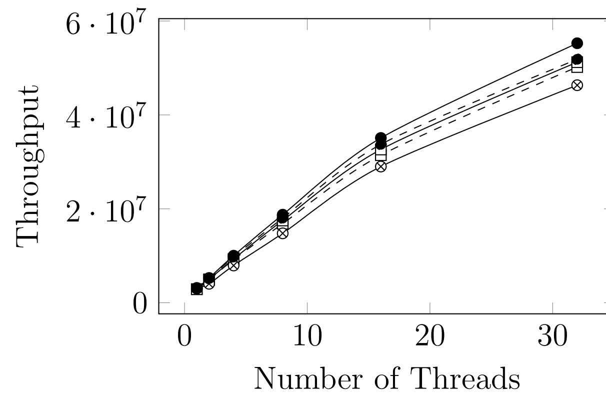
Figure 6.1: Experimental results comparing lf-ravl and lf-ravl3 against self-balancing concurrent BSTs using randomly generated data within key range $(0, 2 \times 10^4]$.



(a) Throughput Under Operation Mix: 90r-9i-1d



(b) Throughput Under Operation Mix: 70r-20i-10d



(c) Throughput Under Operation Mix: 50r-25i-25d

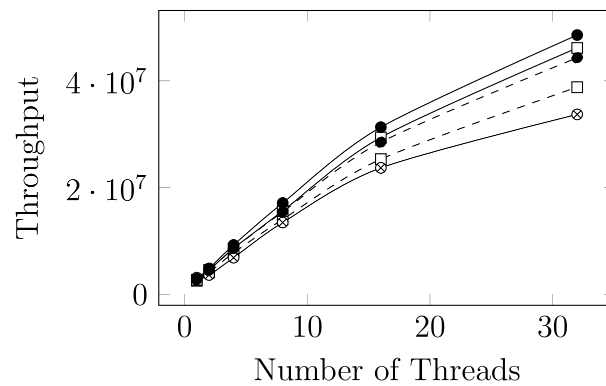
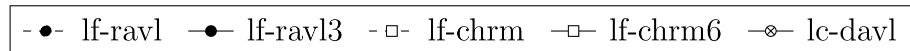
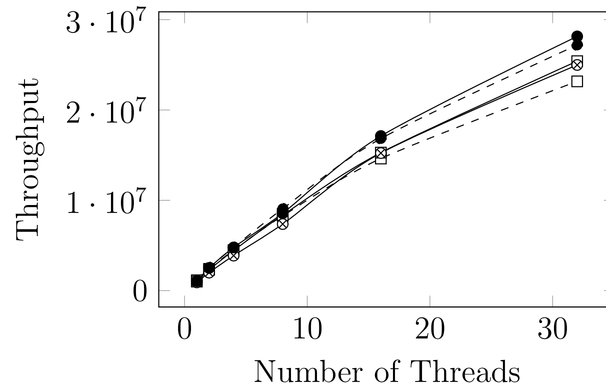


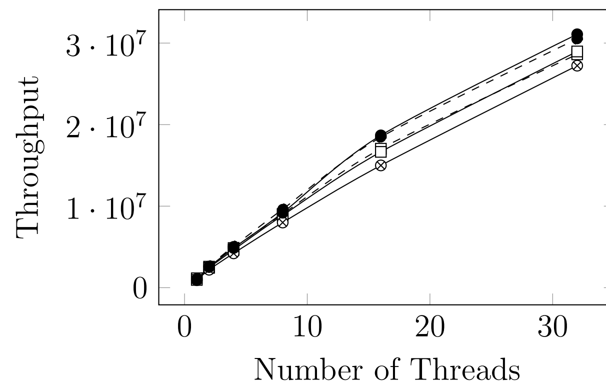
Figure 6.2: Experimental results comparing lf-ravl and lf-ravl3 against self-balancing concurrent BSTs using randomly generated data within key range $(0, 2 \times 10^5]$.



(a) Throughput Under Operation Mix: 90r-9i-1d



(b) Throughput Under Operation Mix: 70r-20i-10d



(c) Throughput Under Operation Mix: 50r-25i-25d

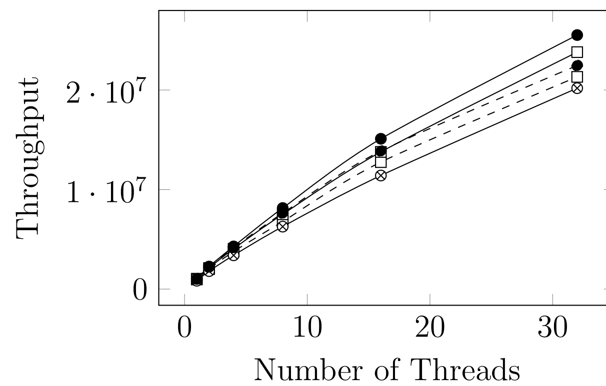


Figure 6.3: Experimental results comparing lf-ravl and lf-ravl3 against self-balancing concurrent BSTs using randomly generated data within key range $(0, 2 \times 10^6]$.

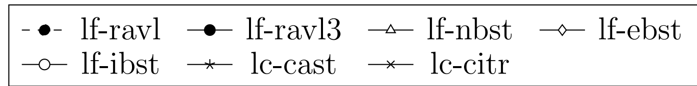
against other self-balancing concurrent BSTs. The x-axis of each study shows the number of threads, and the y-axis shows the average throughput.

lf-ravl3 almost always outperforms lf-ravl, except in case 90r-9i-1d under key ranges $(0, 2 \times 10^4]$ and $(0, 2 \times 10^5]$ where lf-ravl performs slightly better. Since lf-ravl3 has a slightly larger average search path length than lf-ravl, Get operations in lf-ravl are faster than lf-ravl3. This gives lf-ravl advantages under read-dominant operation sequences and in smaller trees. In cases 70r-20i-10d and 50r-25i-25d, lf-ravl3 achieves superior performance. This is because lf-ravl3 reduces the total number of rebalancing steps, and it introduces less overhead in update operations.

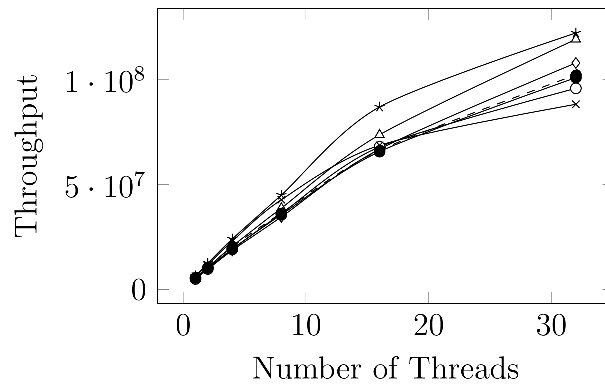
lf-ravl outperforms lf-chrm in every case. Under read-dominant operation sequences (90r-9i-1d), their performance is comparable; while under mixed and write-dominant operation sequences (70r-20i-10d and 50r-25i-25d), their difference in performance becomes more noticeable. Though lf-chrm6 outperforms lf-ravl in case 50r-25i-25d, lf-ravl3 still outperforms lf-chrm6 in every case. This shows that lf-ravl and lf-ravl3 indeed improve performance by avoiding rebalancing after Delete operations.

lf-ravl3 and lf-chrm6 both outperform lc-davl in every case. This difference is more noticeable under key range $(0, 2 \times 10^4]$ and in cases 70r-20i-10d and 50r-25i-25d. This is likely because in lc-davl, processes waste more time trying to acquire locks under higher contention levels. Under key range $(0, 2 \times 10^6]$ and in case 90r-9i-1d, lc-davl achieves comparable performance to lf-ravl and lf-chrm. This is because lc-davl supports wait-free Get operations, and under lower contention levels, update operations in lf-davl waste less time waiting for locks.

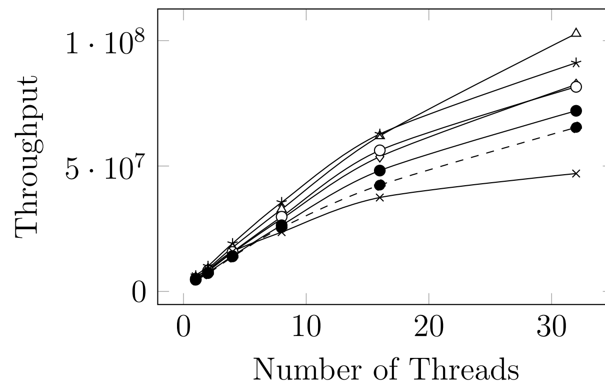
Figures 6.4 - 6.6 show the experimental results comparing lf-ravl and lf-ravl3 against unbalanced concurrent BSTs. In these studies, unbalanced concurrent BSTs have better performance than lf-ravl and lf-ravl3. This is because in the current experimental settings, where all keys are randomly generated under uniform distribution, unbalanced BSTs are balanced with high probability. Therefore, they are likely



(a) Throughput Under Operation Mix: 90r-9i-1d



(b) Throughput Under Operation Mix: 70r-20i-10d



(c) Throughput Under Operation Mix: 50r-25i-25d

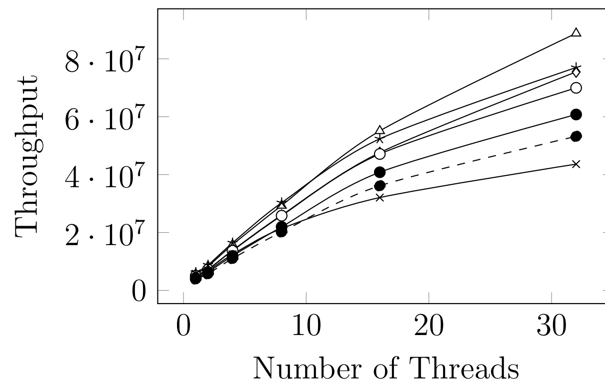
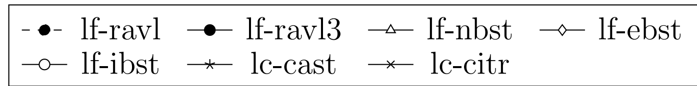
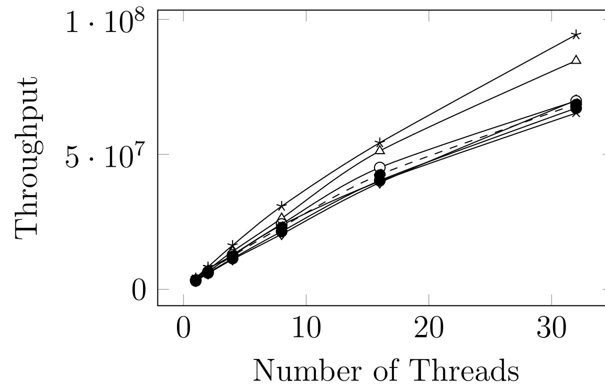


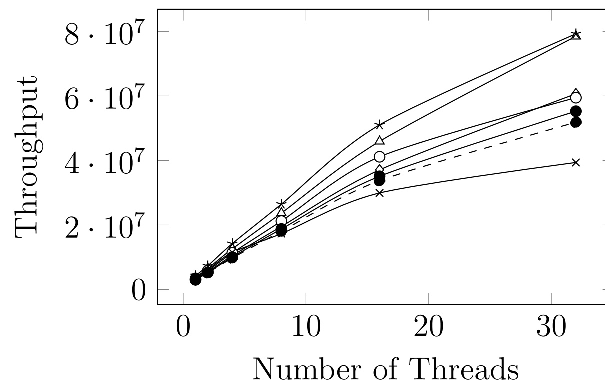
Figure 6.4: Experimental results comparing lf-ravl and lf-ravl3 against unbalanced concurrent BSTs using randomly generated data within key range $(0, 2 \times 10^4]$.



(a) Throughput Under Operation Mix: 90r-9i-1d



(b) Throughput Under Operation Mix: 70r-20i-10d



(c) Throughput Under Operation Mix: 50r-25i-25d

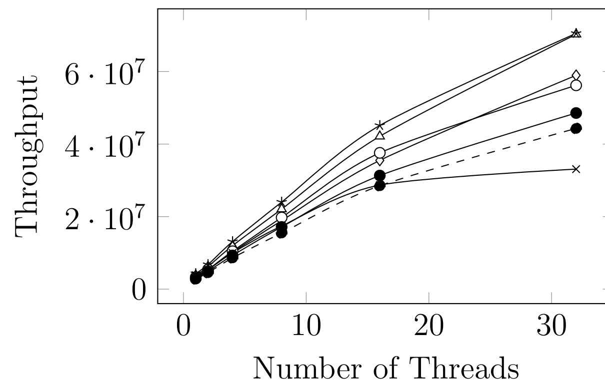
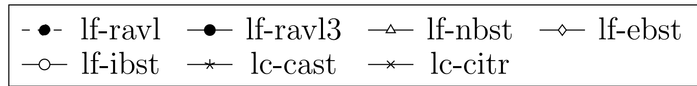
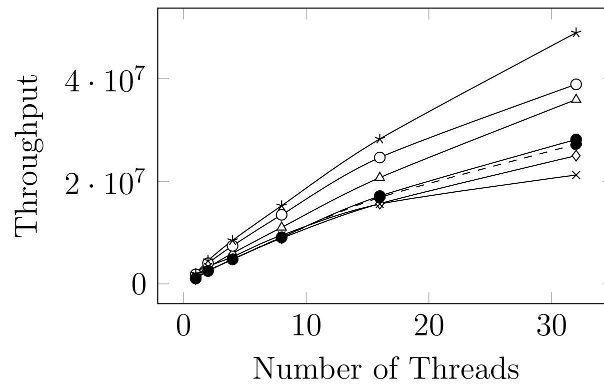


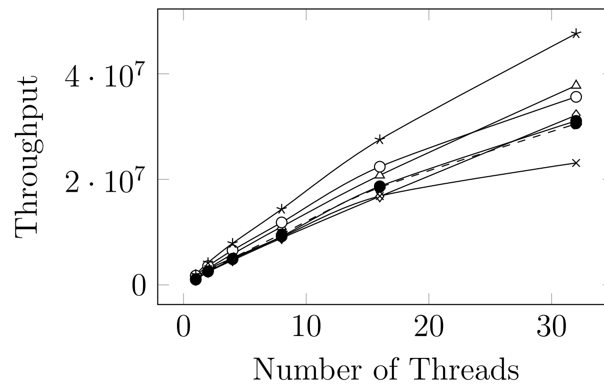
Figure 6.5: Experimental results comparing lf-ravl and lf-ravl3 against unbalanced concurrent BSTs using randomly generated data within key range $(0, 2 \times 10^5]$.



(a) Throughput Under Operation Mix: 90r-9i-1d



(b) Throughput Under Operation Mix: 70r-20i-10d



(c) Throughput Under Operation Mix: 50r-25i-25d

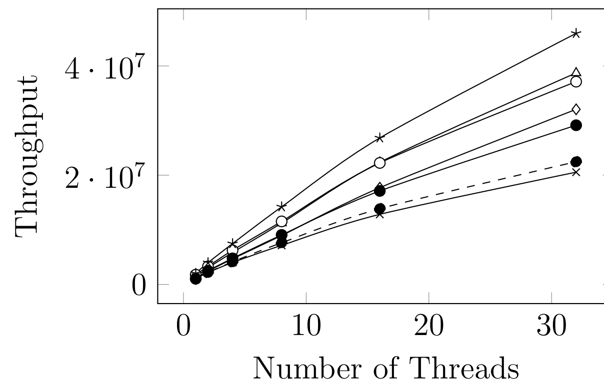


Figure 6.6: Experimental results comparing lf-ravl and lf-ravl3 against unbalanced concurrent BSTs using randomly generated data within key range $(0, 2 \times 10^6]$.

to have better performance as self-balancing BSTs introduce additional overheads during their rebalancing processes.

Even though self-balancing BSTs are not cost-efficient in the current experimental settings, lf-ravl and lf-ravl3 still outperform some unbalanced BSTs. They scale much better for operation sequences with a higher update ratio (70r-20i-10d and 50r-25i-25d) compared to lc-citr (RCU-based). This is due to the synchronization mechanism of RCU where update operations can only be carried out after all existing read operations finish their critical sections and release locks. Also, under key range $(0, 2 \times 10^6]$, lf-ravl and lf-ravl3 outperform lf-ebst in case 90r-9i-1d; while in cases 70r-20i-10d and 50r-25i-25d, their performance is comparable. The main performance bottleneck of lf-ravl, lf-ravl3 and lf-ebst compared to other fast concurrent BSTs is that they all allocate new objects when initiating new updates or retrying failed updates. Under higher contention levels, lf-ravl and lf-ravl3 introduce more overhead during the rebalancing process as conflicts are more likely to occur among processes and thus, they allocate a lot more objects than lf-ebst when retrying failed rebalancing steps. Under lower contention levels, lf-ravl and lf-ravl3 allocates a lot fewer new objects during their rebalancing processes. Also, as the tree grows, the height difference between lf-ravl, lf-ravl3 and lf-ebst becomes larger. As a result, Get operations in lf-ravl and lf-ravl3 are faster than those in lf-ebst, and thus they have better performance under read-dominant operation sequences.

6.4 Data Sequence with Difference Degrees of Presortedness

As data generated from uniform distributions favors unbalanced BSTs, we further test the performance using operation sequences with special properties. In particular, we use synthetic sequences in which keys are partially sorted, i.e., with a certain degree of presortedness. This is motivated by applications in which the data sequence being processed have certain degrees of presortedness. Presortedness has indeed been

extensively studied in adaptive sorting algorithms [16, 17, 31], 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 numbers of the sequence in which the first item is larger than the second. In our study, we generate sequences using the algorithm in [17, 31], which produces larger numbers of inversions efficiently. To create inversions in a sorted data sequence of size n, this algorithm first divides the sequence into $\lceil n/m \rceil$ approximately equal-sized blocks, where m is a user-specified block size between 0 and n to control the degree of presortedness. Then, the numbers in each block are permuted into random order. Next, the algorithm divides the sequence into m approximately equal-sized blocks, picks one random number from each block and permutes these numbers into random order. As a result, this algorithm is expected to generate $mn/2$ inversions on average. Here, the smaller the value of m is, the higher degree of presortedness that the sequence has.

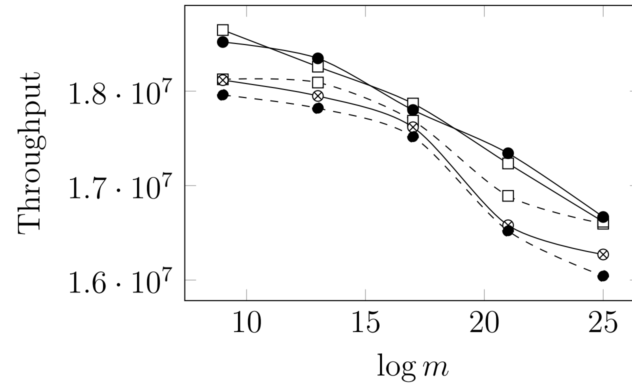
We constructed data sequences with presortedness in the following way: starting from a sorted sequence of unique positive integers of size $n = 2^{26}$, we generated sequences with different levels of presortedness by creating inversions using the algorithm above with the following values of m: 2^9, 2^{13}, 2^{17}, 2^{21} and 2^{25}. We did not show results for m less than 2^9 because of the huge amount of time for unbalanced BSTs to finish the operation sequences. We used the operation mixes 90r-9i-1d, 70r-20i-10d and 50r-25i-25d. For each data structure, using each data sequence and under each operation mix, we ran the program with 32 threads 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, to preserve the presortedness of the data sequences as much as possible, we inserted items in the following way: the thread whose ID is i ($i \in [0, 31)$) starts by inserting the item located at index i of the data sequence, and it keeps inserting items located at index $i + 32$, $i + 64$, $\ldots$, until the entire sequence

is finished. For instance, the thread whose ID is 0 inserts items located at index 0, 32, 64, ..., etc. 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 into the tree. We then counted the total number of operations performed and divided it by the duration of each trial in seconds to compute the average throughput per second. We ran 10 trials for each concurrent BST, and we used the average of these trials as the final results.

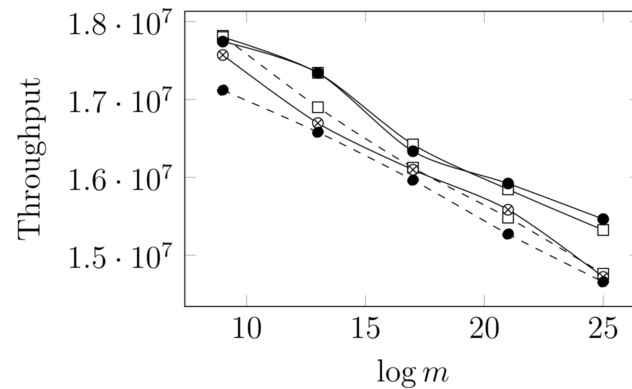
Figure 6.7 shows the experimental results of comparing lf-ravl and lf-ravl3 against other self-balancing concurrent BSTs. The x-axis of each subfigure shows the value of $\log m$, and the y-axis shows the average throughput. lf-ravl3 and lf-chrm6 outperform lf-ravl and lf-chrm in all cases, respectively. However, unlike the results from the previous studies, where lf-ravl and lf-ravl3 always outperform lf-chrm and lf-chrm6, respectively, the current results show that the performance of lf-ravl3 and lf-chrm6 is very similar, and lf-chrm always outperforms lf-ravl slightly. We believe that this is caused by the high success rate of INSERT operations achieved with the current experimental settings. In previous experiments, the expected success rate of INSERT operations is at most 50% (in case 50r-25i-25d); while in the current experimental settings, the success rate is always 100%. Since lf-ravl and lf-ravl3 only rebalance the tree after successful INSERT operations, more rebalancing steps are performed in the current experiments, which affects their performance. lf-ravl3 outperforms lc-davl in all cases. lf-davl has slightly better performance compared to lf-ravl. Since the contention level is lower than the previous experiments (current key range is 2^{26}), update operations in lc-davl waste less time waiting to acquire locks, which improves the performance of this data structure. To conclude, in more cases, lf-ravl3 achieves the best performance among all self-balancing concurrent BSTs. Though lf-ravl is slightly outperformed by others, the performance is still very comparable.

Figure 6.8 shows the experimental results comparing lf-ravl and lf-ravl3 against

(a) Throughput Under Operation Mix: 90r-9i-1d



(b) Throughput Under Operation Mix: 70r-20i-10d



(c) Throughput Under Operation Mix: 50r-25i-25d

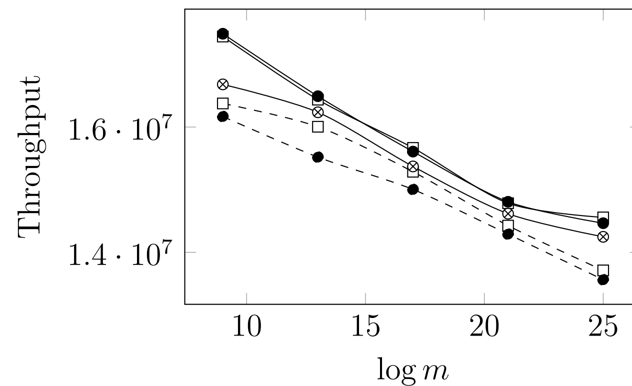


Figure 6.7: Experimental results comparing lf-ravl and lf-ravl3 against self-balancing concurrent BSTs using sequences of size 2^{26} with different degrees of presortedness (32 threads).

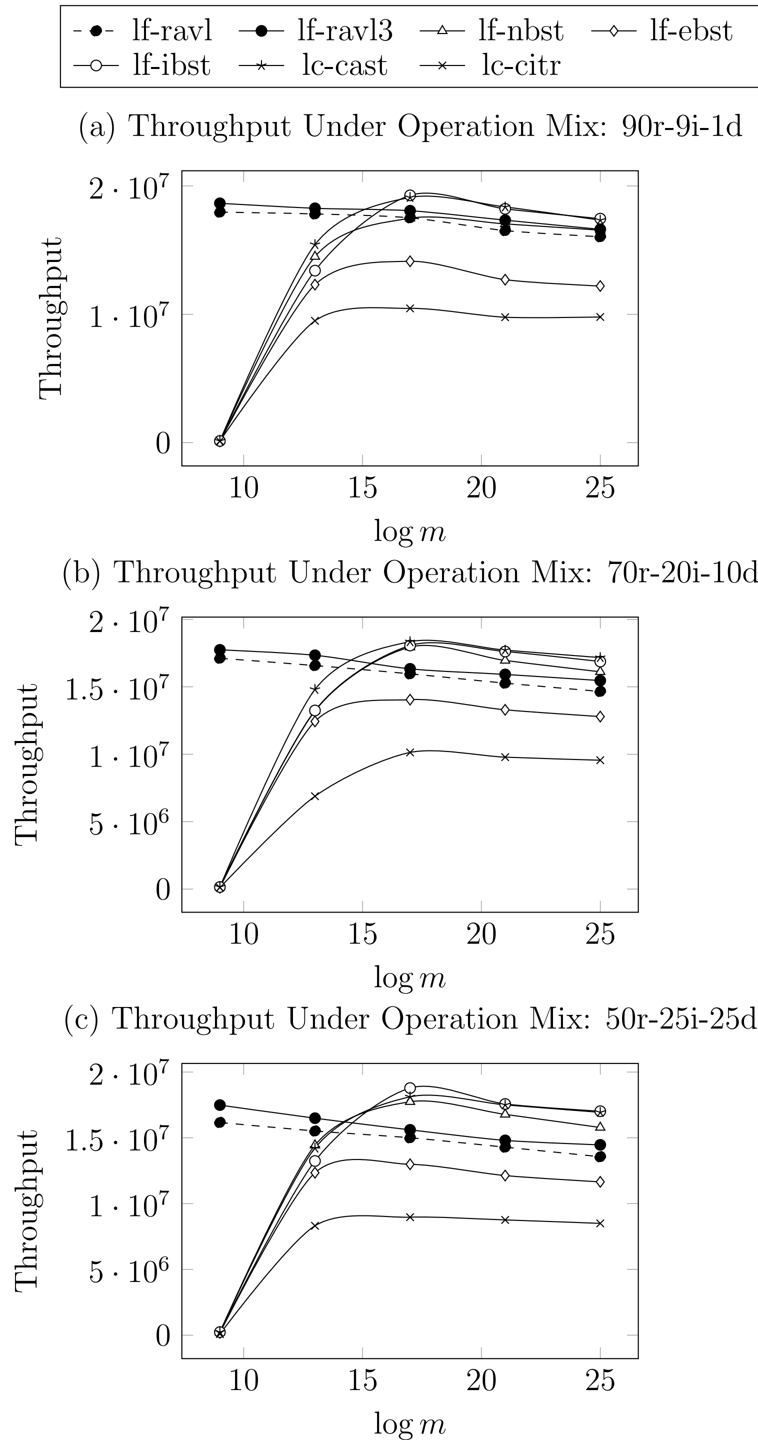


Figure 6.8: Experimental results comparing lf-ravl and lf-ravl3 against unbalanced concurrent BSTs using sequences of size 2^{26} with different degrees of presortedness (32 threads).

Table 6.1: Average tree heights using sequences with existing order under different values of m.

$\log m$	lf-ravl	lf-ravl3	lf-chrm	lf-chrm6	lc-davl	Unbalanced BSTs
9	30	32	42	47	35	41445
11	30	32	40	46	35	10333
13	30	33	40	45	35	1083
15	31	33	39	43	34	177
17	31	33	37	42	36	86
19	31	33	36	40	36	78
21	32	33	34	39	36	70
23	32	33	33	37	35	70
25	32	34	32	37	35	67

other unbalanced concurrent BSTs. To explain these results, we also show the average heights of self-balancing BSTs and unbalanced BSTs under different values of m in Table 6.1. We computed these tree heights by taking the average tree heights under all operation sequences for each data structure for each value of m.

When m is no greater than 2^9 and the data sequences have a high degree of presortedness, the heights of unbalanced BSTs are notably larger than the heights of self-balancing tree, and traversal in unbalanced BSTs takes much longer. Thus, lf-ravl and lf-ravl3 outperform unbalanced BSTs significantly in this case. The average tree heights of unbalanced BSTs decrease drastically when m changes from 2^9 to 2^{17} (from 41445 to 86), which explains their rapid performance improvement. When m is equal to 2^{15}, where unbalanced BSTs are approximately 4 - 5 times higher than self-balancing BSTs, unbalanced BSTs start to have better performance. When m is larger than 2^{17}, the average heights of unbalanced BSTs barely change as the value of m increases (approximately twice of the average heights of self-balancing BSTs). Thus, the data sequences can be considered random, and unbalanced BSTs outperform our solution as in previous studies.

Studies [26] 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 higher 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 concurrent self-balancing BSTs are more suitable for real-world applications compared to unbalanced concurrent BSTs. Considering the fact that lf-ravl3 has the best performance among all self-balancing BSTs, and it introduces only 5 rebalancing cases which makes it extremely easy to implement, we believe that it is the best candidate for many real-world applications.

Chapter 7

Discussion

We present the non-blocking ravl tree, a lock-free relaxed AVL tree that does not rebalance itself after DELETE operations, while still providing a provable non-trivial bound on its height. Our solution introduces a lot fewer rebalancing cases compared to other self-balancing BSTs, which makes it extremely easy to implement in practice. We also conduct experimental studies to compare our solution against other state-of-the-art concurrent BSTs. Experimental results show that our solution is potentially the best candidate for many real-world applications.

When designing and implementing the non-blocking ravl tree, we noticed that it allocates a new object to store coordinating information whenever it initiates a new update attempts. As a result, it allocates more objects compared to other non-blocking BSTs that do not require explicit objects to store information for coordinating among processes, especially under high contention level where the update failure rate is high. Since memory allocations are expensive in concurrent settings, this could be a potential performance bottleneck of the non-blocking ravl tree. The lock-free external BST proposed by Natarajan et al. [24] and the lock-free internal BST proposed by Ramachandran et al. [30] avoid using explicit coordinating objects by not helping INSERT operations and borrowing a few bits of child addresses to store coordinating information. These two solutions are both unbalanced BSTs in which helping is only performed for DELETE operations. Since their coordinating scheme is simple, a few extra bit fields are sufficient for storing all required information. This approach, however, does not directly apply to the non-blocking ravl tree. This is

because our solution introduces several rebalancing operations, and these operations often involve more than one node. Thus, a more complicated coordinating scheme is required. We still think it would be interesting to develop a more memory efficient framework to perform complicated tree update operations using some optimization techniques introduced in [24, 30]. For example, we could potentially reduce the size of the node or information objects by borrowing bits from addresses instead of using extra fields inside of these objects.

In our experimental studies using data sequences with existing order, we also noticed that the performance of self-balancing BSTs decreases slightly as the data sequences become more random (when the value of m gets larger) in Figure 6.7. When the data sequences are nearly completely random (when the value of m is larger than 17), we observed the same behavior in unbalanced BSTs in Figure 6.8. This is consistent with the results from the study by Elmasry et al. [16], who proposed an adaptive sorting algorithm which sorts data sequences by inserting their items into AVL trees. They proved that the number of comparisons required to sort a data sequence X is $n \log \frac{Inv(X)}{n} + O(n)$, where $Inv\ (X)$ is the number of inversions in X and n is the number of items in X, and their experimental results showed that the running time of their algorithm increases as the data sequences become more random (the number of inversions increases). Since inserting items into a self-balancing BST shares many similarities with Elmasry's algorithm, we believe that the decrease in performance of concurrent self-balancing BSTs is related to the increase in the number of inversions in the data sequences. However, since inserting items into BSTs is not exactly an adaptive sorting algorithm, and the data structures tested in our studies are concurrent instead of sequential, we are not able to provide a rigorous proof to bound the running time in terms of the number of inversions in data sequences. In the future, it would be interesting to borrow the idea from [16] or other adaptive sorting algorithms to formally explain this phenomenon.

Bibliography

[1] Gsl - gnu scientific library. `http://www.gnu.org/software/gsl/`.

[2] Maya Arbel and Hagit Attiya. Concurrent updates with rcu: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014. ACM.

[3] Hans Boehm. `http://www.hpl.hp.com/research/linux/atomic_ops/`.

[4] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *SIGPLAN Not.*, 45(5):257–268, January 2010.

[5] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. `http://www.cs.toronto.edu/~tabrown/chromatic/`.

[6] Trevor Brown, Faith Ellen, and Eric 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, New York, NY, USA, 2013. ACM.

[7] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. *SIGPLAN Not.*, 49(8):329–342, February 2014.

[8] Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient lock-free binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 322–331, New York, NY, USA, 2014. ACM.

[9] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2001.

[10] Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 229–240, Berlin, Heidelberg, 2013. Springer-Verlag.

[11] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. `https://github.com/logicalordering/trees`.

[12] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. *SIGPLAN Not.*, 49(8):343–356, February 2014.

[13] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 332–340, New York, NY, USA, 2014. ACM.

[14] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. `http://www.cs.toronto.edu/~tabrown/ksts/StaticDictionary5.java`.

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

[16] 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.

[17] Amr Elmasry and Abdelrahman Hammad. *Experimental and Efficient Algorithms: 4th International Workshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005. Proceedings*, chapter An Empirical Study for Inversions-Sensitive Sorting Algorithms, pages 597–601. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[18] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006.

[19] Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. `https://github.com/gramoli/synchrobench`.

[20] Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 1–10, New York, NY, USA, 2015. ACM.

[21] Philip W. Howard and Jonathan Walpole. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience*, 2013.

[22] Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 161–171, New York, NY, USA, 2012. ACM.

[23] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. https://github.com/anataraja/lfbst.

[24] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, New York, NY, USA, 2014. ACM.

[25] Otto Nurmi and Eljas Soisalon-Soininen. Chromatic binary search trees. a structure for concurrent rebalancing. *Acta Inf.*, 33(6):547–557, 1996.

[26] Ben Pfaff. Performance analysis of bsts in system software. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 410–411, New York, NY, USA, 2004. ACM.

[27] Arunmoezhi Ramachandran and Neeraj Mittal. Castle: Fast concurrent internal binary search tree using edge-based locking. https://github.com/arunmoezhi/castle.

[28] Arunmoezhi Ramachandran and Neeraj Mittal. A fast lock-free internal binary search tree. https://github.com/arunmoezhi/lockFreeIBST.

[29] Arunmoezhi Ramachandran and Neeraj Mittal. Castle: Fast concurrent internal binary search tree using edge-based locking. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 281–282, New York, NY, USA, 2015. ACM.

[30] Arunmoezhi Ramachandran and Neeraj Mittal. A fast lock-free internal binary search tree. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, ICDCN '15, pages 37:1–37:10, New York, NY, USA, 2015. ACM.

[31] Riku Saikkonen and Eljas Soisalon-Soininen. *Experimental Algorithms: 8th International Symposium, SEA 2009, Dortmund, Germany, June 4-6, 2009. Proceedings*, chapter Bulk-Insertion Sort: Towards Composite Measures of Presortedness, pages 269–280. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[32] Siddhartha Sen and Robert E. Tarjan. Deletion without rebalancing in balanced binary trees. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 1490–1499, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.