Concurrent Data Structures Using Multiword Compare and Swap

by

William Sigouin

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

© William Sigouin 2020

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Much of the work within this thesis is also outlined in a paper submitted to PODC 2020 for publication that is currently pending review, titled "Efficient Concurrent Data Structures Made Easier," on which I am the first author. My supervisor Trevor Brown and our collaborator Dan Alistarh are the second and third authors on this work, respectively.

I personally created all the algorithms and implementations for the framework and data structures within this work, meeting with my above collaborators to discuss ideas and work through specific issues.

Abstract

To maximize the performance of concurrent data structures, researchers have turned to highly complex fine-grained techniques. Resulting algorithms are often extremely difficult to understand and prove correct, allowing for highly cited works to contain correctness bugs that go undetected for long periods of time. This complexity is perceived as a necessary sacrifice: simpler, more general techniques cannot attain competitive performance with these fine-grained implementations. To challenge this perception, this work presents three data structures created using multi-word compare-and-swap (KCAS), version numbering, and double-collect searches that showcase the power of using a more coarse-grained approach. First, a novel lock-free binary search tree (BST) is presented that is both *fully-internal and balanced*, which is able to achieve competitive performance with the state-of-the-art fine-grained concurrent BSTs while being significantly simpler. Next, the first concurrent implementation of an Euler-tour data-structure is outlined, solving fully-dynamic graph connectivity. Finally, a KCAS variant of an (a,b)-tree implementation is presented, which shows significant performance improvements in certain workloads when compared to the original.

Acknowledgements

I would like to thank my supervisor Dr. Trevor Brown for his guidance in both an academic and professional capacity. His eye for quality and vast knowledge have helped create something that I am truly proud of. His door was always open when I had questions or concerns, always taking the time to help me when I required it.

I would also like to thank our collaborator, Dr. Dan Alistarh, whose insights were extremely helpful in navigating various hurdles that came up when conducting this research.

I also wish to express my gratitude to my thesis committee readers, Dr. Peter A. Buhr and Dr. Samer Al-Kiswany for their helpful comments.

Finally, I thank my family. My siblings have always been a constant source of love and support, and for that I am eternally grateful. My parents have always encouraged me for as long as I can remember, and the two of them have never failed to be there when I needed them. I am who I am today in no small part because of all they have done for me.

Table of Contents

List of Figures **ix**

1 Introduction **1**

2 Model **5**

3 Background **8**

3.1 KCAS 8

3.2 Related Work 9

4 Motivation **11**

4.1 The Difficulty of Proving Fine-Grained Data Structures Correct . . . 11

4.2 Drachler Tree Bug 12

4.2.1 Counter Example 12

4.2.2 Solution: Search Direction Swap 18

5 Simplifying the use of KCAS **21**

5.1 Interface 21

5.2 Implementation 23

6 AVL Tree via KCAS **25**

6.1 Overview 25

6.2 Algorithm Design . . . 27
6.2.1 Searching . . . 27
6.2.2 Insertion . . . 30
6.2.3 Deletion . . . 32
6.2.4 Rebalancing . . . 35
6.3 Correctness Proof . . . 42
6.4 Progress Proof . . . 48
6.5 Balance Proof . . . 48
6.6 Evaluation . . . 50
6.6.1 Throughput . . . 51
6.6.2 Key Depth and Cache Misses . . . 52

7 Dynamic Connectivity via KCAS 57
7.1 Overview . . . 57
7.2 Algorithm Design . . . 58
7.2.1 Connection Queries . . . 58
7.2.2 Link . . . 60
7.2.3 Cut . . . 61
7.3 Correctness Proof . . . 63
7.4 Progress Proof . . . 65

8 (a,b)-tree via KCAS 67
8.1 Overview . . . 67
8.2 Algorithm Design . . . 68
8.3 Evaluation . . . 70
8.4 Possible Extensions . . . 70

9 Conclusion **75**

9.1 Summary 75

9.2 Future Work 75

9.2.1 Node-based KCAS 75

9.2.2 Combining Data Structures 76

References **77**

List of Figures

1.1 Incorrect update on a concurrent BST 3

2.1 A **linearizable** execution of a concurrent counter 6
2.2 A **non-linearizable** execution of a concurrent counter 7

4.1 Valid state of Drachler tree 13
4.2 Drachler tree after insertion of 175 into the tree, but before rebalancing 14
4.3 Drachler tree after partial insertion of 160 15
4.4 Drachler tree during thread p's rebalancing, during the search of thread s for 160 16
4.5 Drachler tree after thread s' invalid search for 160 17
4.6 Invalid, non-linearizable, execution of the Drachler tree 18
4.7 Drachler tree during thread s' search for 160, now recoverable with new search 19
4.8 Drachler tree during thread s' search for 85, before the rotation that could cause an improper search 20
4.9 Drachler tree after thread s' search for 85, now recoverable with the new search 20

6.1 Node Layout 26
6.2 AVL insert 31
6.3 AVL Two-Child Erase 33
6.4 AVL One-Child Erase 34

6.5 AVL Leaf Erase 34
6.6 AVL Possible Rotations, *height* field is abbreviated to h for brevity . 36
6.7 AVL Right Rotation 36
6.8 AVL Left-Right Rotation 38
6.9 **Throughput** comparison with additional BSTs. 51
6.10 AVL operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^6$ 52
6.11 AVL operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^7$ 53
6.12 AVL operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^8$ 54
6.13 AVL operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^9$ 55
6.14 AVL LLC misses per operation comparison, lower is better 56

7.1 Operation $link(3, 6)$ on (simplified) Euler tour lists 59
7.2 Operation $cut(3, 6)$ on (simplified) Euler tour lists 62

8.1 Simple insert case on the LLX/SCX implementation of the (a,b)-tree 68
8.2 Simplest insert case on the KCAS implementation of the (a,b)-tree . . 69
8.3 (a,b)-tree operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^6$ 71
8.4 (a,b)-tree operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^7$ 72
8.5 (a,b)-tree operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^8$ 73
8.6 (a,b)-tree operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^9$ 74

Chapter 1

Introduction

The field of concurrent data structure research is massive, and is centered around building scalable concurrent variants of fundamental data structures. Fast implementations are now known for a wide array of data structures and *fine-grained synchronization*, meaning data structures implemented using either *fine-grained locking* or *fine-grained lock-free algorithms*, are arguably the best general solution in terms of performance (see recent study by [4]). From these works, a vast range of highly non-trivial techniques have been introduced and implemented.

Concurrent data structures designed in this way, however, are notoriously difficult to analyze and prove correct. Work in this area often adds complexity in order to maximize performance of a data structure, which can lead to extremely intricate correctness arguments. When an approach requires both a complex implementation and a complex proof, it leaves a high probability for error. As outlined in Chapter 4, even published designs that proposed correctness arguments can still hide non-trivial correctness issues.

These fine-grained techniques often rely on relatively weak primitives. The classic example is compare-and-set (CAS), which allows the programmer to *atomically* compare the value at a certain location in memory against an expected value, and to modify this location if they are equal. However, it is often the case that when implementing non-trivial data-structures, the operational semantics require the atomic modification of *multiple* memory locations. A myriad of approaches have been proposed to mitigate this issue, ranging from "ad-hoc" techniques, such as marking, e.g. [17, 25], to more complex software primitives on top of CAS, e.g. [18, 9], or improved hardware support for concurrency, e.g. [20]. These techniques induce non-trivial trade-offs in the design space between implementation and the complexity

of the correctness proof, on the one hand, and practical performance and progress guarantees, on the other.

This work revisits the question of providing general, easy-to-use and practically-efficient support for concurrent data structures. It is possible that the complexity of the designs resulting from fine-grained methods could be stifling progress in the area, by limiting the range of data structures that can be correctly and efficiently implemented. This work seeks a middle ground between the high-performance via high-complexity of fine-grained synchronization algorithms, and the simplicity, but relatively lower performance, of coarser-grained designs. To this end, the core focus of this work is to attempt to determine if it is possible to have designs that are efficient, but also relatively easy to understand and prove correct.

As an instance of such middle ground, three data structures are presented in this work that rely on multi-word compare-and-swap, (a.k.a k-word compare-and-swap, KCAS), version numbering, and double-collect validation of searches. I believe this combined approach can be used to efficiently implement a wide range of concurrent data structures, while maintaining ease of implementation and proof simplicity, and illustrate this claim via three examples. The first implementation presented is a *fully-internal*, lock-free balanced binary search-tree (BST), which is able to closely match the performance of the state-of-the-art fine-grained counterparts, while being significantly simpler to describe and prove correct. Second, a concurrent implementation of an *Euler-tour data-structure* for solving *fully-dynamic graph-connectivity* is outlined. To the best of my knowledge, this is the first concurrent implementation of this classic data structure, which would be extremely challenging—if not impossible—to implement and prove correct using fine-grained synchronization. Finally, a modification of an existing implementation for an (a,b)-tree that uses this approach is presented. This implementation is significantly simpler than the original, while being able to achieve superior performance under certain workloads.

To illustrate the difficulty in creating concurrent data structures, consider a traditional sequential implementation of an internal binary search tree (BST), where we wish to insert some new key k, if it is not already present. An insert operation first searches for k. The last node visited by the search (if k is not found) is the insertion point of the new node containing k. Therefore, the location where the search terminates is the insertion location. The insertion adds a new node as a child of the insertion-location node.

However, in a concurrent context, this approach is not sufficient. In Figure 1.1, thread p is attempting to insert 20 and thread q is attempting to remove 15. First, thread p traverses to the node containing the key 25. Before p can take any more

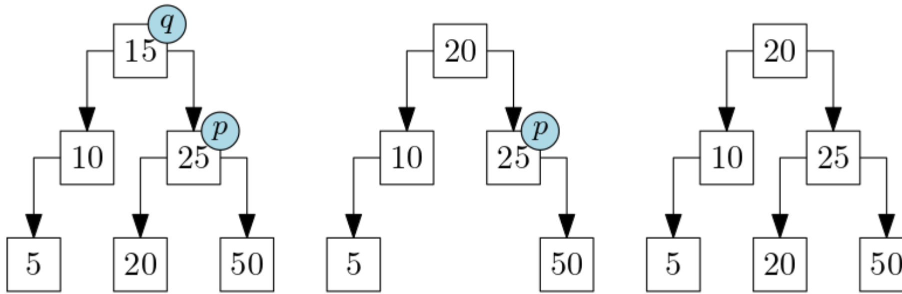

Figure 1.1: Incorrect update on a concurrent BST

steps, q atomically removes the key 15 by replacing the key with 20 (15's successor) and removes the old node containing 20. Thread p will then attempt to traverse to the left child of 25, find none there, assume 20 is not within the tree, and insert it at this location. This update is not valid, *despite its atomicity at the node level*; the key 20 now appears twice within the tree. In fact, even if the search portion of operations are atomic, it is still insufficient. For example, consider Figure 1.1 again. Even if thread p's search is atomic, there is a time between the search for 20 and the insert of 20. If, during this time, another thread inserts 20 and thread q then deletes 15 (replacing it with 20), p still does not observe this change and inserts 20 again.

Because to this complexity, many of the published implementations for balanced concurrent BSTs [6, 13] achieve lock-free progress by using *routing nodes*. These auxiliary nodes are external (or partially-external) to the BSTs and avoid difficult update cases using extra keys that are not necessarily in the set represented by the actual data structure. However, these additional nodes bloat the data structure and increase its height. Using the simpler approach in this work (described in Chapter 6), such difficult cases can be handled without auxiliary nodes. By including a version number in every node in the data structure, and updating a node's version number atomically with each change to that node, atomic searches are simple to implement. It is also shown that KCAS atomic updates, atomic searches, and version numbers in combination allow for operations to be implemented in a way that is intuitive, efficient, and easy to prove correct.

Despite this reduction in complexity, the data structures created using this framework are far from trivial. Consider the example above, there exists a time after the search part of the insertion before the update part during which the data structure can change, possibly invalidating the result of the search used to inform the update. This data structure specific problem can be difficult to solve. In Chapter 6, I illustrate how KCAS and version numbers can be leveraged to simplify the design

process.

Finally, performance experiments show that this BST is competitive with the state-of-the-art. The cache friendliness and low-cost of searches in these implementations allow us to attain similar, and often superior, performance to highly-optimized fine-grained techniques. These results show that fine-grained techniques are not necessary to achieve good performance, and that the techniques used in these highly-complex implementations can often have a non-trivial performance cost.

C++ implementations of all data structures outlined within this work, along with the test harness used for our experiments, are publicly available.[1]

[1] https://gitlab.com/uw-multicore-lab/setbench_kcas

Chapter 2

Model

Throughout this work, the classic asynchronous shared-memory model is assumed. There is an unknown number of threads interacting with memory at any given time. No assumptions are made about the relative speed or order of these threads. Threads may halt at any location and at anytime, either by completion or failure. For simplicity, no memory reclamation is performed, and therefore memory is unbounded. However, in practice, all algorithms presented are compatible with a thread-safe memory reclamation scheme, such as the epoch-based memory reclamation scheme from [7].

All data structures presented are *node-based*. Nodes are located at a specific addresses in memory, and are of fixed size. A data structure may contain nodes of different size. The address range of a node is divided into several fixed-size fields, each being located at a specific offset from the base address of the node, enabling fields to be looked up given the address of the node and the offset of the field. These fields can contain pointers to other nodes or other arbitrary values pertaining to the specific data structure (such as keys).

A *system configuration* is the overall state of all threads and the shared memory accessed by these threads on a system. Consider a system that is in some configuration C. From C, a thread p may take some *step* s_i that changes its state and/or shared memory. A step that changes a thread's state or shared memory moves the configuration of the system from C to a new configuration C'. An *execution* is a sequence of configurations of a system over time, and the steps that transitioned the system from one configuration to the next (e.g. C_0, s_0, C_1, s_1, C_2, $\ldots$, s_{i-1}, C_i).

Linearizability is a correctness condition presented by Herlihy and Wing [21] for concurrent data structures. A specific execution of some concurrent data structure is

called *linearizable* if it is “equivalent” to some legal sequential execution of that same data structure. If *linearization points* during every operation in a concurrent execution can be chosen such that the return values of these operations are equivalent to what they would be if the operations were executed atomically at those linearization points, then it has an equivalent sequential execution. Consider a simple concurrent counter with only one operation, *inc()*, which increments the value of the counter by 1 and returns the value of the counter **before** the increment occurs. In Figure 2.1 a linearizable execution of this counter is shown. Three threads p, q and r (on the y-axis) execute several *inc()* operations represented by the boxes within the diagram. The return value of these operations are shown within these boxes as well. A set of possible linearization points for this execution are represented by arrows for each operation. Note that, if we execute these operations atomically at these points in a concurrent execution, the return value of these operations remain the same, hence this execution is linearizable.

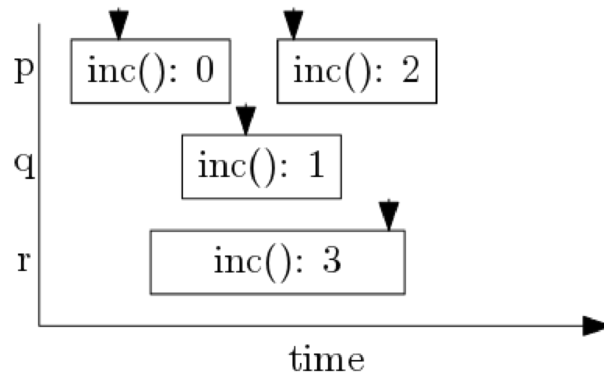

Figure 2.1: A **linearizable** execution of a concurrent counter

Consider another similar execution in Figure 2.2. This execution is not linearizable because there exists no set of correct linearization points. The key issue here is the gap between the only *inc()* of thread r and the second *inc()* of thread p. In order for the *inc()* of r to return 3, the linearization points of three *inc()* operations must precede its linearization point, which is not possible. However, a linearization point for this *inc()* must be chosen during the operation, and only two *inc()* operations occurred before the end of this one. Hence, this execution is not linearizable.

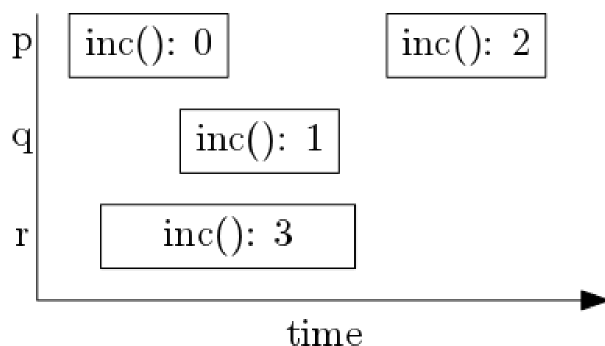

Figure 2.2: A **non-linearizable** execution of a concurrent counter

In general, a data structure is linearizable if every possible concurrent execution on the data structure is linearizable. We prove that the data structures presented within this work are all linearizable.

A data structure is said to be *lock-free* if for every possible valid configuration of the data structure, some operation eventually completes, even if threads crash. In essence, this means that some threads always make progress towards completing operations; however, some threads can starve. It should be noted that implementing a data structure without traditional locks is not sufficient (but is necessary) to achieve lock-free progress. All data structures presented within this work are proved to be lock-free.

Chapter 3

Background

3.1 KCAS

All data structure modifications in this work are performed via the KCAS operation. KCAS is semantically very similar to compare-and-set (CAS), with the key difference that it is able to change **multiple** addresses atomically. In its simplest form, KCAS supports a single operation: *KCAS(*$field_1$*,* $oldValue_1$*,* $newValue_1$*, ...* $field_k$*,* $oldValue_k$*,* $newValue_k$*)*. KCAS does the following atomically: if $field_i$ contains $oldValue_i$ for all i, the value stored at $field_i$ is changed to $newValue_i$ for all i and returns true. If not, false is returned. The fields $field_1, \ldots, field_k$ are called **changing fields**.

The implementation of KCAS used in this work is based on the work of Harris [18], and includes the optimizations described by Arbel-Raviv and Brown [3]. This implementation is created using multiple CAS operations in order to change all k fields atomically. For synchronization and to achieve lock-free progress, this implementation replaces program values with pointers to *descriptors*. These descriptors contain information about ongoing KCAS operations, and threads that read a field containing a KCAS descriptor help complete the KCAS represented by this descriptor enabling lock-free progress.

I improved the interface of the KCAS implementation outlined here to make it simpler to use and less error-prone. These improvements are outlined in Chapter 5.

3.2 Related Work

The question of identifying synchronization primitives with the "right" balance between expressivity and efficiency is almost as old as the field itself. Treiber [35] gave one of the first illustrations of employing CAS to obtain a non-blocking data structure, while seminal work by Herlihy [19] established this primitive is *universal*. Herlihy and Moss [20] proposed Hardware Transactional Memory (HTM) as a form of flexible hardware support for non-blocking data structures. HTM allows programmers to execute entire sections of code atomically. These sections of code, or *transactions*, either commit, where they are executed atomically, or abort, where they have no effect on the configuration of the system. HTM tracks all shared memory accesses by threads, aborting transactions if a conflict is detected. A restricted form of their proposal is now supported by mass-produced processors, e.g. [11], although it lacks **exact** progress guarantees and can still suffer from performance artefacts [27]. In this work, HTM is used on the fast-path of the KCAS implementation, but is not relied upon for progress. Shavit and Touitou introduce Software Transactional Memory [30], as a software-only alternative to HTM. Kumar et al. introduce *Hybrid Transactional Memory* (HyTM) [24], which is a combination of HTM and STM. Guerraoui and Trigonakis present *Optik* [16], which is an approach that uses optimistic concurrency and versioned locks to implement several concurrent data structures such as linked-lists, queues, BSTs, and hash-tables. This approach is potentially generalizable to implement other concurrent data structures.

Anderson and Moir gave the first constant-time implementations of *Load Link* (LL) and *Store Conditional* (SC) [2] from CAS. This work is extended by Brown et al. [9], introducing *LLX* and *SCX*, which function on *data records*. Data records contain *multiple* related fields which are loaded by LLX, and SCX only succeeds in changing a single memory location if none of the fields loaded by *LLX* have changed since they were loaded by *LLX*. LLX/SCX is less expressive than KCAS, as it only allows for the change of a single field. Brown et al. [9] exhibited several search tree data structure designs based on LLX/SCX, one of which is modified within this work to use the KCAS-based approach presented in this work.

Tarjan and Vishkin [32] introduces the concept of Euler Tours for dynamic connectivity. Tseng et al. [36] produced a fully-dynamic connectivity structure that is *batch-parallel*: threads collaborate to carry out a *batch* of updates all at the same time. This work takes the Euler tour structure outlined in [33] and constructs a skip-list structurally similar to the one used in the implementation in Chapter 7. This approach, however, is only useful for applications that allow for this batching

up updates. Within the implementation in this work, however, threads do not collaborate to complete operations (other than via KCAS helping) and have their own independent operations.

There have been many concurrent BST implementations including Bronson's lock-based balanced partially external BST [6], an external non-blocking BST by Ellen et al. [15], Howley et al.'s internal lock-free BST [22], Natarajan et al.'s [28] external balanced BST, and Brown et al.'s [10] chromatic tree from LLX/SCX. None of these achieve both balance and lock-freedom without the use of routing nodes.

Kelly et al. [23] use double-collect searches, originally presented by Afek et al. [1], and KCAS to implement a Robin-Hood hash-table. This double-collect ensures the probing portion of operations is not invalidated by concurrent operations, similar to the searches in the data structures within this work.

Chapter 4

Motivation

4.1 The Difficulty of Proving Fine-Grained Data Structures Correct

The creation and implementation of concurrent data structures is difficult, and is often an error prone process. Correctness bugs exist even in peer reviewed works, going undetected for long periods of time. In this chapter, an overview of previously discovered issues in other work is provided, and one new bug found during my investigation.

Michael and Scott [26] discovered two race conditions within the lock-free concurrent queue by Valois [37], which lead to incorrect memory reclamation. These issues could corrupt the data structure in two ways: by freeing the same nodes multiple times, or freeing nodes that are still logically within the structure. The memory reclamation scheme attempts to avoid ABA problems by relying on reference counters in data structure nodes, where threads increase the reference counter of a node when they read a pointer to it, and decrement the counter when they no longer require it. The thread that moves this reference counter to 0 after a node has been removed reclaims the node. There exists a time, however, between when a pointer to a node is read by a thread and that thread increments the reference counter of the node. During this time another thread could move the reference counter of the node to 0 and free it. The thread about to increment the reference counter is unaware of this, and increments the reference counter of the node to 1, then potentially back to 0, resulting in a double free. A similar issue can also result in nodes being freed despite still being in the data structure.

Shafiei [29] found an execution that dereferences a null pointer within the lock-free doubly-linked-list by Sundell and Tsigas [31] by running the Java PathFinder (JPF) model checker on the implementation. The work demonstrates how the sheer complexity of the algorithm makes it very difficult to reason about its correctness, and that the proofs provided within the work are insufficient. The authors created an updated version of this work where this issue is corrected.

Translating an algorithm into a usable implementation is also a difficult endeavour, and many artifacts provided with published papers contain non-trivial correctness bugs. An investigation into concurrent BSTs by Arbel-Raviv et al. [4] shows that several such implementations of concurrent data structures provided with published work can fail basic checksum validation.

As part of this work, we discovered a new bug in the balanced BST by Drachsler et al. [13]. This tree uses additional pointers in nodes to track the predecessor and successor of a node, which are used to recover from searches that end up in an incorrect location due to a concurrent rotation. An execution exists, however, wherein searches fail to find keys that are present during the entire search operation. I notified the authors of this work, who confirmed the bug via private communication and plan to release errata on the topic at a later date. A full explanation of the incorrect execution is in the following section.

4.2 Drachler Tree Bug

The tree within this work is structured like a normal BST, however nodes have *predecessor* and *successor* pointers to nodes that contain those respective keys. These *predecessor* and *successor* pointers essentially create a doubly linked list connecting all nodes within the data structure. Searches first traverse the BST performing a normal BST search, however if a key is not found during a search, this result will be confirmed by traversing these predecessor and successor pointers. This section outlines an incorrect execution of this data structure that I discovered.

4.2.1 Counter Example

Consider the tree structure in Figure 4.1:

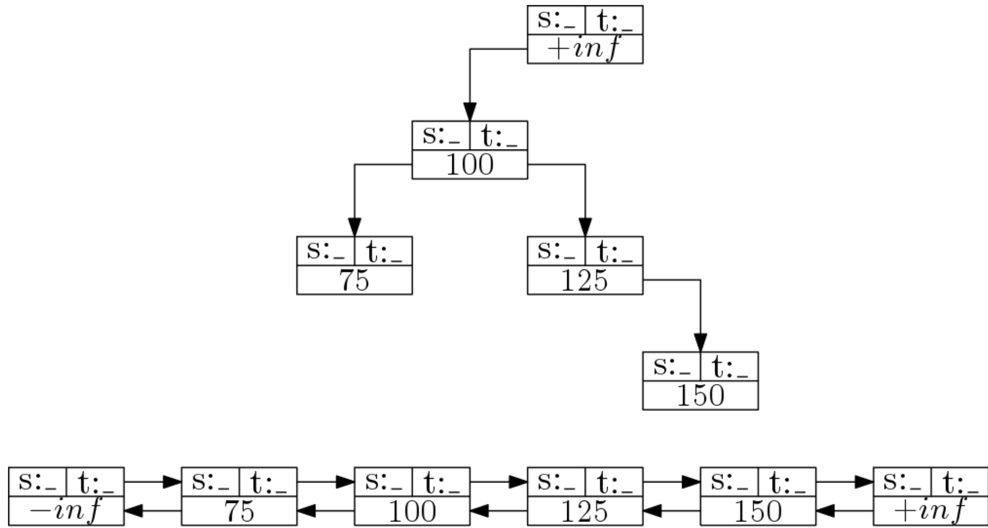

Figure 4.1: Valid state of Drachler tree

This shows both the tree and logical ordering structure as shown in the original Drachler work. The bottom values are keys, and s and t fields represent the succLock, which protects the successor pointer of a node, **and** the predecessor pointer of the node that is pointed to by its successor pointer, and treeLocks, which protects the *left* and *right* child pointers of a node, respectively. The _ represents no thread holds the lock. Assume a thread p inserts the value 175 to both the logical order and the tree, but has not rebalanced the tree yet, i.e., just before line 9 of Algorithm 5 in the original work, leaving the tree in the a state shown in Figure 4.2. Thread p now goes to sleep.

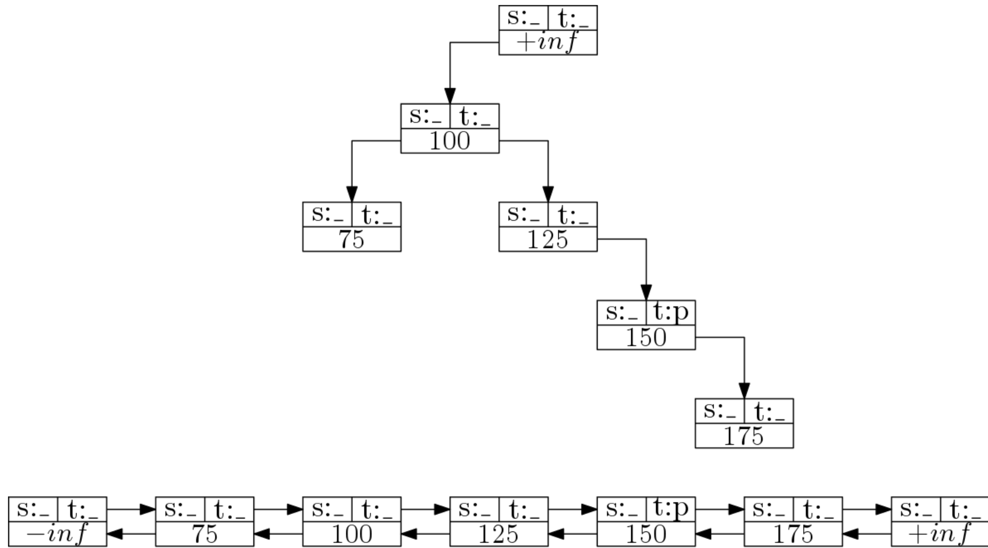

Figure 4.2: Drachler tree after insertion of 175 into the tree, but before rebalancing

Note that the treeLock for the node 150 is still held by thread p, but the succLock is released as that operation is completed. A rotation should occur on the node 125. Before this rotation can occur, thread q performs an insertion of 160 up until line 15 of Algorithm 3 of the original work. This step can occur because the succLock of 150 has been released by p (only the treeLock is held) and the treeLock of 175 is the one that is acquired for this operation. The tree is now in the state shown in Figure 4.3. Thread q now goes to sleep.

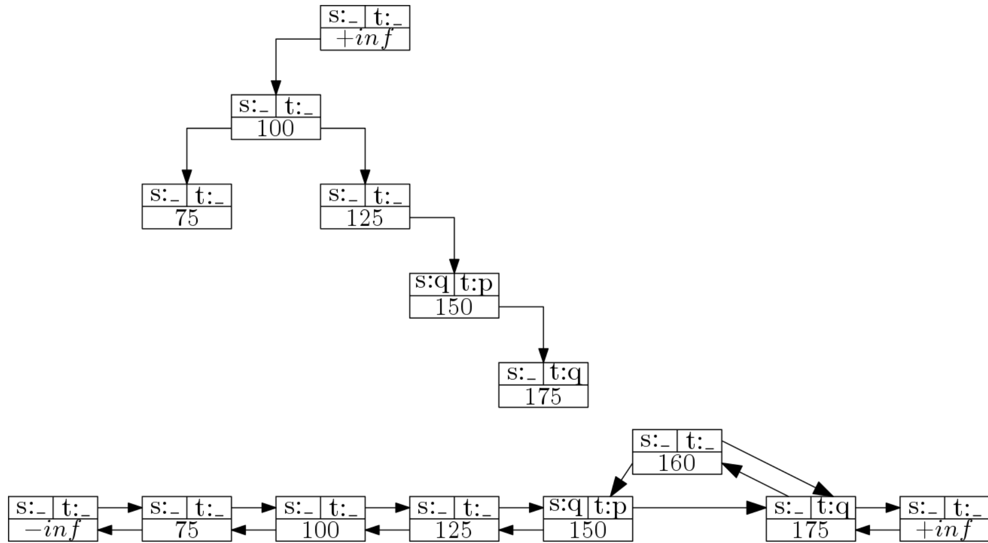

Figure 4.3: Drachler tree after partial insertion of 160

A third thread r does the contains operation in Algorithm 1 for the key 160 to completion. r searches the tree as per Line 1 in Algorithm 1 below, then backtracks to find 160. This operation ignores all locks and returns true. This result implies the insertion of 160 must already have been linearized. However, in the original work, the linearization point has not yet been reached, and is therefore incorrect (line 16 of *insert* in the original work, where *pred.succ* is updated) as the change is observed here when it must be before this line (line 15 of *insert* is logical).

Algorithm 1 contains(k)

1: node = BSTsearch(k)
2: **while** *node.key* $>$ *k* **do** node = node.pred
3: **while** *node.key* $<$ *k* **do** node = node.succ
4: **return** *(node.key == k* ***and*** *!(node.mark))*

Next, a fourth thread, thread s, does the same operation as thread r (contains(160)), however it goes to sleep after it traverses to node 125.

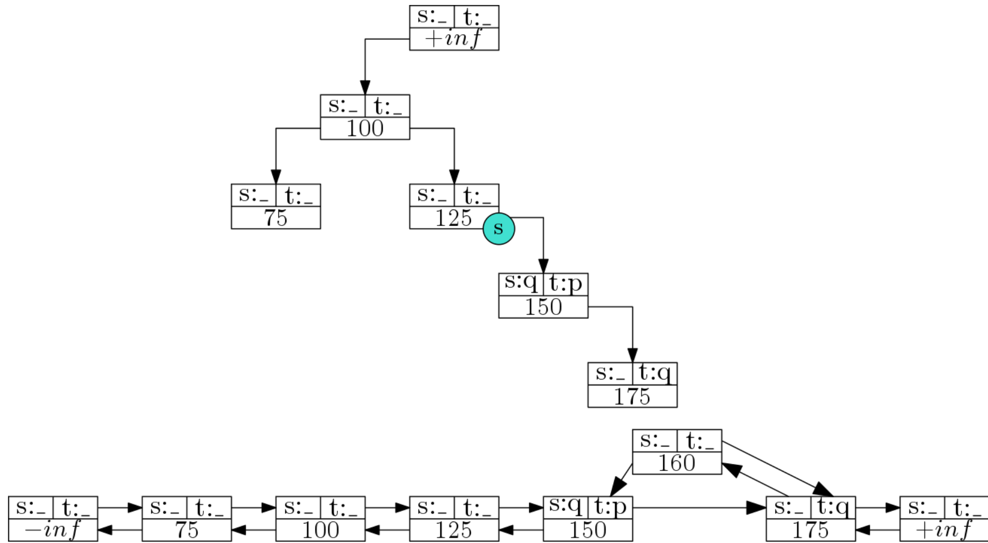

Figure 4.4: Drachler tree during thread p's rebalancing, during the search of thread s for 160

Thread p now wakes up and starts a rotation, this results in a call to function *rebalance(node, child)* with the node containing 125 as the "node" argument, and the node containing 150 as the "child" argument. Thread p already has the treeLock for 150, and can freely acquire the treeLocks for 125 and 100 (the other ones required for this rotation). Note here that thread q only holds the treeLock for 175. No threads hold succLocks at this point. p can proceed because it is not blocked by any currently held locks.

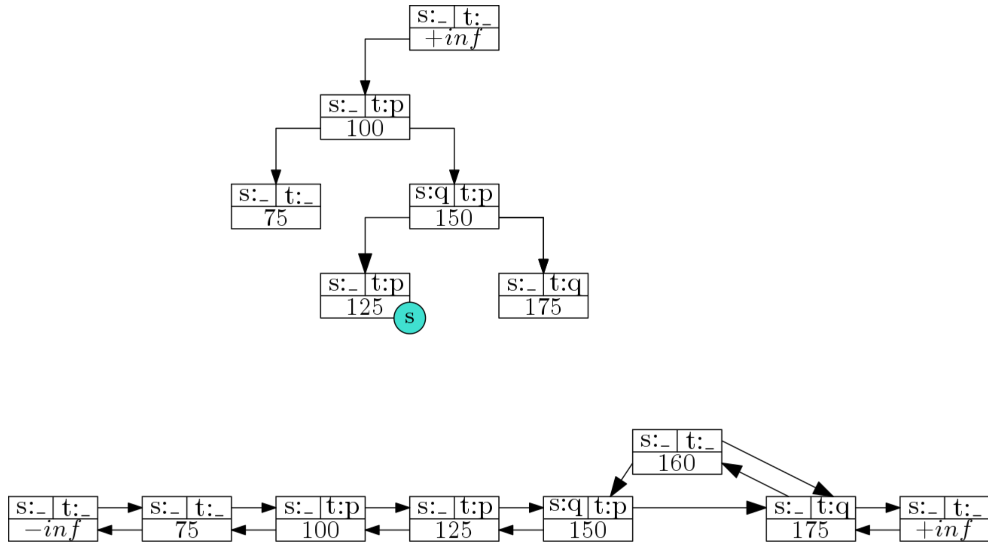

Figure 4.5: Drachler tree after thread s' invalid search for 160

Thread s (Figure 4.5) now wakes up after this rotation, detects that it is at a leaf node and completes its traversal of the tree. From there, it attempts to follow the logical order to discover if it missed an update. Line 2 of Algorithm 1 does not traverse the list as $node.key > k$ is false, but line 3 traverses (as $node.key < k$) until the node containing 175, sees that this key is not 160 and returns false. This result is invalid, as the linearization point for the insert(160) of thread q must have passed or else the result of the previous search for 160 must be incorrect.

Figure 4.6 shows the thread schedule of the executing threads and that the two contains operations cannot be linearized.

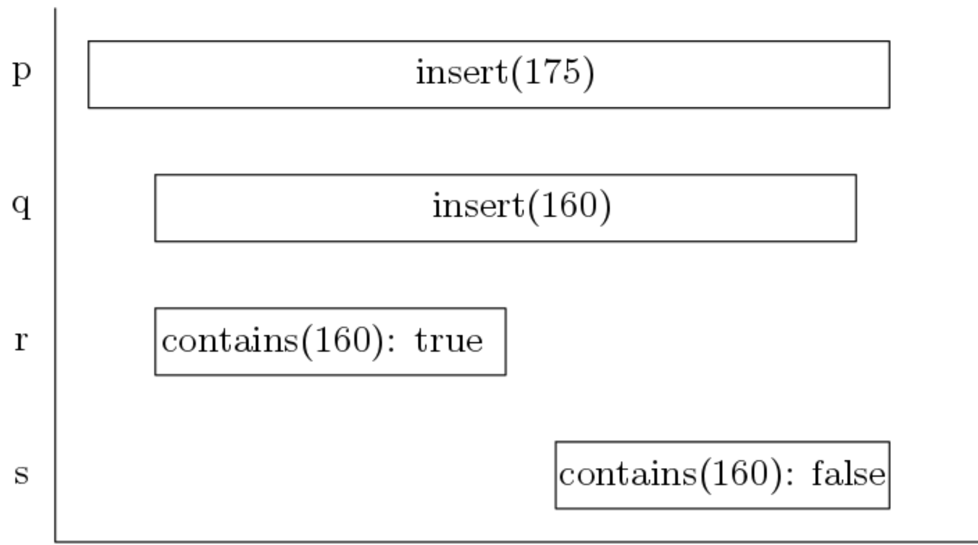

Figure 4.6: Invalid, non-linearizable, execution of the Drachler tree

4.2.2 Solution: Search Direction Swap

Algorithm 2 reverses line 2 and 3 in the Algorithm 1 contain operation:

Algorithm 2 contains(k)

1: node = BSTsearch(k)
2: **while** *node.key* $<$ *k* **do** node = node.succ
3: **while** *node.key* $>$ *k* **do** node = node.pred
4: **return** *(node.key == k* ***and*** *!(node.mark))*

Figure 4.7 shows the example above, up to the last state but using Algorithm 2.

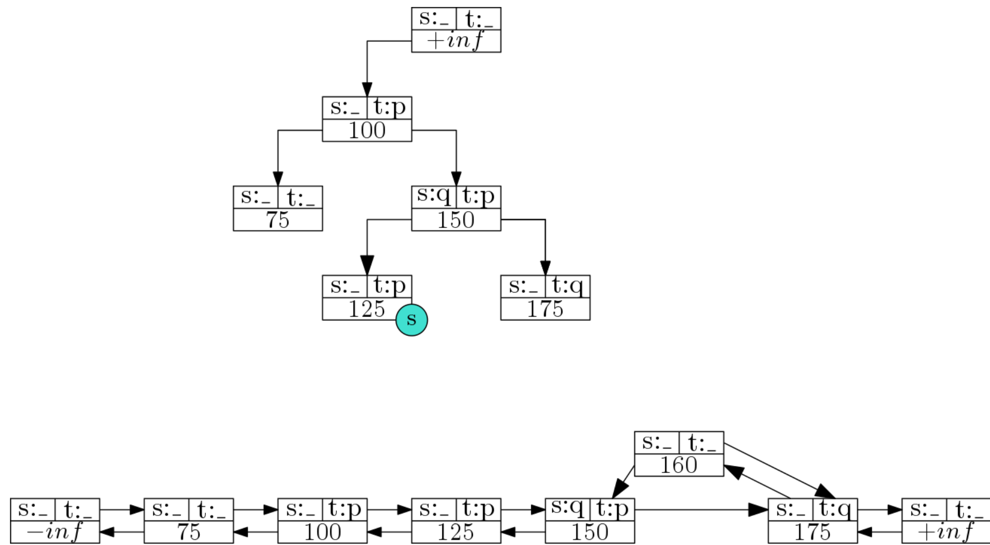

Figure 4.7: Drachler tree during thread *s*' search for 160, now recoverable with new search

Thread *s* now finds the key 160, and the insert operation can be linearized. Actually, *regardless of the node found after this search, the contains operation finds 160*. If a thread is left of the partially inserted node *n*, it will traverse succ pointers until it passes *n*, then it will follow a single pred pointer to *n*. If a thread is to the right of the partially inserted node, it will follow no succ pointers, but follow pred pointers until it reaches *n*.

Figure 4.8 shows the reverse case, before a right rotation occurs and Figure 4.9 shows after the rotation occurs. Now, *s* is searching for 85, arrives at 95 but then gets rotated down. If it searches right then left, it still finds 85. It is the same as the previous example, *regardless of what node contains terminates the BSTsearch, it finds 85.*

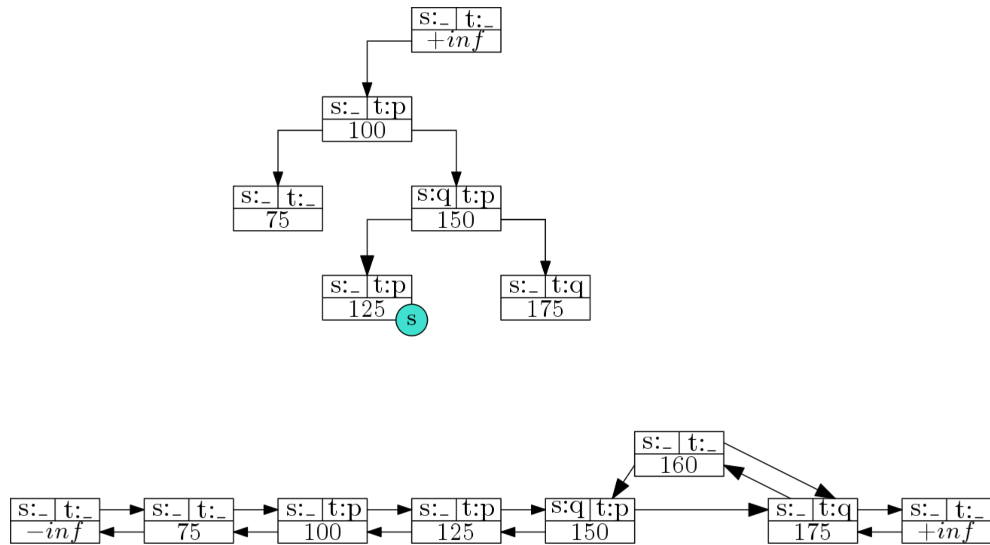

Figure 4.8: Drachler tree during thread s' search for 85, before the rotation that could cause an improper search

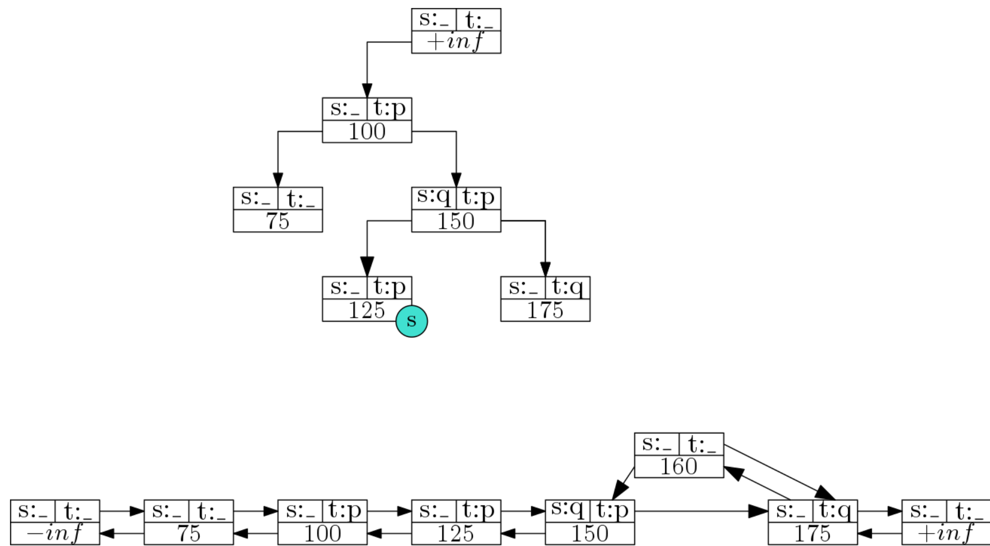

Figure 4.9: Drachler tree after thread s' search for 85, now recoverable with the new search

Chapter 5

Simplifying the use of KCAS

5.1 Interface

This section describes the C++ implementation of KCAS and compares it to those presented in prior work.

Algorithm 3 shows the operation *change* that modifies two fields of a node, called *ptr*, which is a node address or NULL, and *val*, which is an `integer` value. *change* operates on some node n, and returns false if $n \rightarrow ptr$ is NULL; otherwise *change* creates a new node with a *val* field equal to $n \rightarrow val + 1$ and a *ptr* field that points to n's address. Additionally, $n \rightarrow ptr$ is changed to the address of this new node.

Algorithm 3 *sequentialChange(n)*

1: **if** $n \rightarrow ptr \neq NULL$ **then return** *false*
2: $newNode = allocate(Node)$ ▷ Allocate a new node, return the address
3: $newNode \rightarrow ptr = n$
4: $newNode \rightarrow val = n \rightarrow val + 1$
5: $n \rightarrow ptr = newNode$
6: **return** *true*

If this operation is part of a concurrent data structure created using a KCAS implementation, such as in [18, 3], it would look something like Algorithm 4. Recall this KCAS implementation temporarily replaces values stored at fields with pointers to **descriptors**.[1] Additional work is required to determine if a value stored at a field is a program value or a descriptor, so all fields that can be modified via KCAS

[1] Certain operations also construct these descriptors, then pass them to an execute function

must be read and instantiated with special *kcasRead* and *kcasInit* functions. As a small optimization, fields that contain pointers are handled slightly differently by this implementation than fields that contain non-pointer values, and therefore require separate *kcasReadPtr* and *kcasInitPtr* functions. Finally, all fields manipulated by KCAS must be a single type defined by the KCAS implementation (such as in Algorithm 4, which uses `uintptr_t`), meaning the type of node fields is fixed, and fields must be cast to and from this KCAS type and their logical type.

Algorithm 4 *kcasChange(n)*

```
1: while true do
2:     ptrTemp = (Node)kcasReadPointer(&n→ptr)
3:     if ptrTemp ≠ NULL then return false
4:     desc = kcasGetDescriptor()
5:     newNode = allocate(Node)
6:     kcasSetInitialPointer(&newNode→ptr, (uintptr_t)n)
7:     valTemp = (int)kcasReadValue(&n→val)
8:     kcasSetInitialValue(&newNode→val, (uintptr_t)valTemp + 1)
9:     desc→add(&n→ptr, (uintptr_t)ptrTemp, (uintptr_t)newNode)
10:    desc→add(&n→val, (uintptr_t)valTemp, (uintptr_t)valTemp)
11:    if kcasExecute(desc) then return true
```

Following all these guidelines for using KCAS properly can be difficult, as small mistakes such as using the wrong *kcasRead* results in subtle logical errors that are difficult to find and debug. To make this process simpler, we introduce a new interface as part of this work that removes much of the burden from the programmer. Concurrent data structures created using this interface are closer to their sequential equivalent and explicitly prevent many misuses of KCAS, by translating such mistakes into compilation errors.

Our interface uses C++ language features in order to avoid many of the issues with traditional KCAS implementations. All modified fields used by KCAS must use the generic type `casword`, which is a C++ **templated**[2] type with a single template parameter: the logical type of this field (such as an integer, or a boolean). By

which carries out the KCAS. This approach is an alternative to having a large *kcas()* call at the end of an operation as outlined in the ADT (which is described in Section 3.1), making it easier to handle branches that affect the fields included as part of a KCAS.

[2] A template type is a C++ polymorphic feature generalizing across types that supply a common set of operations, e.g., type `stack<T>`, has operators `top`, `push`, and `pop`, independent of the type stored in the stack. A template-type field in an object is a *placeholder* for type T, such as `object<int>` and `object<bool>`, in which the placeholder type is `object`, respectively.

overloading the operators[3] on this type, we can carry out *kcasReads* implicitly, allowing the programmer to interact through `casword` as if it is the logical type. This approach also enables several features that help avoid programmer mistakes. For example, `casword` disables the assignment operator (`=`) so a value of a field cannot be modified without performing a KCAS or calling an initialization function. The interface is also modular in the sense that swapping the KCAS implementation requires virtually no changes to the data structures that use it. An implementation of the toy operation *change* using our *new* interface is shown above in Algorithm 5 (c.f. the old interface in Algorithm 4).

Algorithm 5 *kcasNewChange(node)*

```
 1: while true do
 2:     ptrTemp = n→ptr                                  ▷ implicit kcasRead happens here
 3:     if ptrTemp ≠ NULL then return false
 4:     kcas::start()                                    ▷ KCAS is now a C++ namespace
 5:     newNode = allocate(Node)
 6:     newNode→ptr.init(n)                              ▷ init works on both pointers and values
 7:     valTemp = n→val
 8:     newNode→val.init(valTemp + 1)
 9:     kcas::add(&n→ptr, ptrTemp, newNode,
10:               &n→val, valTemp, valTemp)
11:     if kcas::execute() then return true              ▷ No descriptor handling, happens implicitly
```

5.2 Implementation

The data structures within this work use a two-path KCAS implementation that uses HTM as a fast path and uses a lock-free implementation of KCAS [18, 3] as a fallback path. For a fixed number of attempts, our KCAS implementation uses HTM to carry out the KCAS and avoid the heavier cost of the CAS-based lock-free implementation.

[3]Operator overloading is a C++ polymorphic feature that allows function names to be repeated and selection is based on parameter type and number of parameters. For example, the member access operator `->` can be changed so additional work (such as invoking *KCASRead*) can be performed *before* a value is returned.

Algorithm 6 kcas::execute()

```
1: desc = getDescriptor()                                    ▷ Gets the current thread's descriptor
2: for tries = 0; tries < 5; tries++ do
3:     if (status = _xbegin()) == _XBEGIN_STARTED) then
4:         for index = 0; index < desc→entries; index++ do
5:             val = *desc→entries[index].addr
6:             if val ≠ desc→entries[index].oldval then
7:                 if isKCASDescriptor(val) then _xabort(READ_DESCRIPTOR)
8:                 _xabort(BAD_OLD_VAL)
9:         end for
10:        for index = 0; index < desc→entries; index++ do
11:            *desc→entries[index].addr = desc→entries[index].newval
12:        end for
13:        _xend()
14:        return true
15:    else if _XABORT_EXPLICIT & status then
16:        if _XABORT_CODE(status) == READ_DESCRIPTOR then break
17:        else if _XABORT_CODE(status) == BAD_OLD_VAL then return false
18: end for
19: return kcasLockFree(desc)
```

Algorithm 6 shows the HTM implementation. It first checks all the fields added to the KCAS descriptor to ensure these fields hold the old values from the descriptor. If any of these fields contain an incorrect old value, the transaction is explicitly aborted. If the incorrect value is a KCAS descriptor, the slow path is taken. We cannot simply return false, as we do not know the logical value of this field. If the value is not a KCAS descriptor, then the KCAS returns false, as it contained a value other than the one in the descriptor.

If all the fields contain old values in the descriptor, they are in the *read set* of the transaction. Changes to any of these fields after these reads results in an abort, and the transaction retries up to a fixed number of attempts. If the transaction succeeds in writing all the new values to the fields with no conflicts, the transaction commits, and *true* is returned.

A similar approach is followed in [34]. Several details of the KCAS algorithm (such as the sorting of descriptors for progress) are omitted from Algorithm 6 for simplicity.

Chapter 6

AVL Tree via KCAS

This section now introduces the framework used to implement the data structures within this work, and illustrate it via a concurrent AVL tree using a hands-on example. The data structure outlined in this chapter is the first (to the best of my knowledge) lock-free implementation of a concurrent, balanced BST that is *fully-internal*.

6.1 Overview

The tree implements a dictionary, which supports the following operations: *insertIfAbsent(key)*, *erase(key)* and *contains(key)*. *insertIfAbsent(key)* returns true and inserts *key* into the dictionary if it is not already present; otherwise, false is returned. *erase(key)* returns true and removes *key* from the dictionary if it is present; otherwise, false is returned. *contains(key)* returns true if *key* is in the dictionary; otherwise, false is returned. As in a traditional BST, nodes have the following fields: a key (*key*), a left child (*left*), a right child (*right*) and a value (*val*).

To facilitate correct synchronization, nodes are augmented with two additional fields: the version number of the node (*ver*) and its least significant bit is used as a mark to represent a deleted node, as shown in Algorithm 7. In other words, nodes with odd version numbers are treated as marked, and hence logically removed from the tree. The use of version numbers follows these rules:
Rule 1: *Before* reading any other field of a node, the *version number* must be read and saved in thread-local memory.
Rule 2: Nodes that are changed by a KCAS operation must also have their *version*

numbers included in the KCAS. (The *old value* of the version number should be the value saved in **Rule 1**. The new value increments the old value by *two*[1].)

Algorithm 7 Node type definition

```
template<typename K, typename V>
struct Node {
    casword<uint64_t> ver;
    casword<K> key;
    casword<Node<K, V> *> left;
    casword<Node<K, V> *> right;
    casword<Node<K, V> *> parent;
    casword<int> height;
    casword<V> val;
};
```

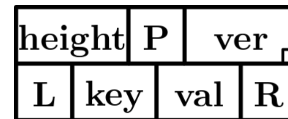

Figure 6.1: Node Layout

Atomically incrementing version numbers of a node alongside every update enforces a strict sequential ordering on updates **to that node**. Every update operation must increment the version number, and at most one operation can increment the version number from a specific value, making concurrent updates to the same node impossible. (Perhaps surprisingly, even with this limitation, this approach can obtain highly competitive implementations, as experiments show in Section 6.6.) Additionally, any changes to a field that occur between when the node's version number is read, and when a KCAS is subsequently performed, causes the KCAS to fail, preventing many incorrect (non-atomic) updates.

Note, however, that to implement a data structure update atomically using KCAS, it may not be sufficient to include *only* the fields that are changed by the update in the KCAS. For example, the search part of an insertion in a binary search tree determines if an insertion should occur and where, but only a small number of the nodes/fields read during the search are changed by the insertion. If only the fields to be *changed* are included in the KCAS, then it is possible for a KCAS to succeed, even if many nodes on the search path have changed—possibly even if the location being changed by KCAS is *no longer the correct location where this key should be inserted*.

To prevent these sorts of problems in an AVL tree, a set D of nodes is identified such that, *if a KCAS should no longer be performed*, e.g., because it would insert in

[1]Unless this node is being *marked* as part of the operation, then the old value is incremented by one.

the wrong place, then some node in D has changed since we read its contents. We call these *dependency nodes*. We add the *version numbers* of these dependency nodes to the KCAS (setting the old and new values to the same value—the version number read to satisfy **Rule 1**). As a result, any change that *should* cause the update to fail and possibly some other benign changes, as well, changes a dependency node, causing the KCAS to fail.[2] For an AVL tree, it turns out that D should consist of the predecessor and successor of the key being inserted. (The intuition is that a node containing a key *between* the predecessor and successor must change in order for the location where the key *belongs* to change. Such a change requires a change to the predecessor or successor. See Drachsler et al. [14] for details.)

Two more fields are required to rebalance the tree: *height* and *parent* (see Figure 6.1). *Rebalancing* steps are performed when the heights of a node's children differ by two or more. When a rotation is performed on a node n, it may be necessary to rebalance its parent p, which is reachable via n's parent field.

To avoid special cases, the tree always contains two *sentinel* nodes with keys $-\infty$ and $+\infty$. Consequently, every node with key $k \in (-\infty, +\infty)$ always has both *predecessor* and *successor* nodes. The sentinel with key $+\infty$, called the *max-root*, is the root of the entire tree. The sentinel with key $-\infty$, called the *min-root*, is the left child of *max-root*. No field of *max-root* is ever changed. The *min-root* can have its right child pointer and version number changed, but is never rebalanced. All keys in $(-\infty, +\infty)$ are always found in the *right subtree* of *min-root*.

The rest of this section gives an overview of the tree operations, and the following sections offer a proofs for correctness, progress, and balance.

6.2 Algorithm Design

6.2.1 Searching

The *search* function (Algorithm 8) performs a traditional BST search until it reaches a *NULL* node, or finds a node containing the key k it is searching for. Whenever a node n is visited by *search*, a pointer to n and n's current version number are recorded in the *path* and *vers* local array variables, respectively. *search* returns a

[2] In many cases, I have found that including dependencies in the KCAS with a *per-node* granularity by including *version numbers* actually leads to a *reduction* in the number of fields needed in a KCAS compared to including a tighter set of dependencies with a *per-field* granularity, which can yield significant performance improvements.

tuple of five items: two nodes, two version numbers and a boolean in the following format: $\langle node, version, node, version, boolean \rangle$.

If k is found, search returns pointers to the node containing k and the parent of that node, the version numbers read in those nodes and *true*: ⟨*node, nodeVersion, parent, parentVersion, true*⟩. If k is not found, then a function called *validatePath* is invoked to *validate* the contents of *path* and *vers* (Algorithm 9). This is done to ensure that concurrent modifications to the tree did not cause the search to *miss* key k in the tree (explained further below). If *validatePath* returns true, validation *succeeds*. In this case, search returns k's *predecessor* and *successor* nodes, which contain the largest key smaller than k and the smallest key larger than k, respectively. The version numbers read in those nodes and false: ⟨*predecessor, predecessorVersion, successor, successorVersion, false*⟩ or ⟨*successor, successorVersion, predecessor, predecessorVersion, false*⟩ — *predecessor* and *successor* are returned in the order they are encountered during *search*. Otherwise, *validatePath* returns false, and validation *fails*. In this case, the search is *restarted*.

validatePath ensures the search path followed by *search* is correct at some time during the search. More specifically, *validatePath* rereads the version number of each node in *path*, and returns true if: no version number has changed since reading and storing it in *vers*, **and** no node in *path* is *marked*. Otherwise, false is returned. If a search is performed with a subsequent successful *validatePath*, then a time has been established when *path* was an *atomic snapshot* of the search path. (This approach is essentially the classical *double collect* algorithm [1].)

Algorithm 8 search(k)

```
 1: while true do                                                    ▷ Retry loop
 2:     path[]; vers[];
 3:     path[0] = maxRoot
 4:     vers[0] = maxRoot→ver;                                       ▷ Note: uses KCASRead
 5:     n = root→left                                                ▷ Note: uses KCASRead
 6:     sz = 1                                                       ▷ Number of nodes in path
 7:     predIx = -1; succIx = 0;                                     ▷ Index of k's pred/succ in path
 8:     while true do
 9:         if n == NULL then                                        ▷ Reached a leaf
10:             if validatePath(path, vers, sz) then
11:                 a = min(predIx, succIx)                          ▷ The shallower of pred/succ (ancestor)
12:                 return ⟨path[a], vers[a], path[sz - 1], vers[sz - 1], false⟩
13:             else break                                           ▷ Failed validation
14:         path[sz] = n
15:         vers[sz] = n→ver                                         ▷ Note: uses KCASRead
16:         currKey = n→key                                          ▷ Note: uses KCASRead
17:         sz = sz + 1
18:         if key > currKey then
19:             predIx = sz - 1
20:             n = n→right                                          ▷ Note: uses KCASRead
21:         else if key < currKey then
22:             succIx = sz - 1
23:             n = n→left                                           ▷ Note: uses KCASRead
24:         else return ⟨path[sz - 1], vers[sz - 1], path[sz - 2], vers[sz - 2], true⟩
```

Note that paths are only validated when k is not found, because if a node n with the key k **is** found during a traversal, then there was a time during the operation where the k was within the tree, regardless of whether n is marked. This observation is true because no operation in this tree ever changes a marked node. (Recall that a node is marked if and only if its version number is odd. Before performing a KCAS, it is verified that the old values for all version numbers are even.) Hence, the fields of unmarked nodes never contain the addresses of marked nodes because they are unlinked and marked in the same atomic step. If a node is unlinked before a search, then the search cannot reach it. Nodes that are unlinked *during* the search are possibly reachable, but existed within the tree at some time during the search. Hence, any node reached during a search existed in the tree at some time during the search.

Algorithm 9 validatePath(*path, vers, sz*)

```
1: for i in 0 ... sz - 1 do
2:     if path[i]→ver ≠ ver[i] or isMarked(ver[i]) then
3:         return false
4: end for
5: return true
```

If the search did not find a node containing k, and the path is successfully validated by *validatePath*, then there is a time during the search when k is not in the tree. As discussed previously, a validated search path is the result of an atomic search at *some time* during the operation (in fact, the moment just before the invocation of *validatePath* is such a time). Since this atomic search does not find k, there is a time during the search when k is not in the tree.

The *contains* operation (not shown) simply calls *search* and returns whether the key is found or not.

6.2.2 Insertion

The *insertIfAbsent(k, v)* operation first identifies the location to insert the new key k by performing a search. If *search* finds k (the last item in the tuple returned by *search* is *true*), then there is a time where k exists within the tree, and *false* can be returned (line 3). If *search* does not find k (the last item in the tuple returned by *search* is *false*), the predecessor and successor nodes (along with their version numbers read during search) are returned. These nodes a, which is the first of the two nodes traversed in *search*, and p, which is the **final** node traversed in *search*, are critical to the insertion of k. At the time the search is validated, the correct location for the new node n containing k to be inserted is a child of p, so the KCAS attempts to add n as a child of p. a is not directly involved in the KCAS, but it is a dependency node for this operation (explained further below), so its version number is added to the KCAS.

A depiction of the operation is shown in Figure 6.2. The left image shows the state of the tree before the insertion occurs, and the right image shows the state of the tree after the insertion. In this example p and a are the predecessor and successor nodes of k, respectively (the opposite is also possible, but only changes the child pointer of p that is changed). (Note that a always is an ancestor of p.) Node fields are colored according to their role in the KCAS: orange is a changing field

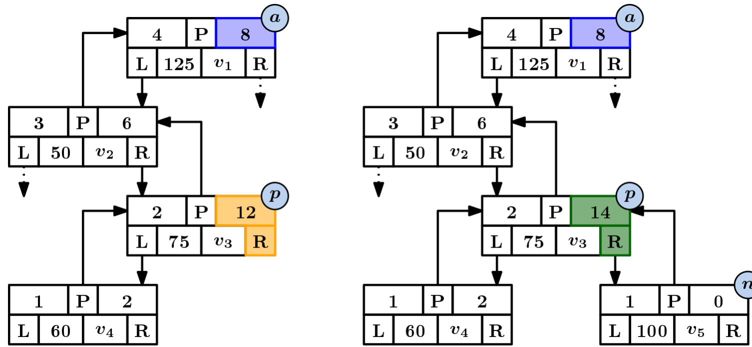

Figure 6.2: AVL insert

before the KCAS is executed, green is a changing field after a KCAS was successful, and blue is the version number of a dependency node.

Note that both p and a's marks were already checked as part of *validatePath*, and do not need to be checked again as part of this operation. If p or a are marked between the time the validation occurred and the KCAS is executed, their version number has changed, ensuring the KCAS fails.

Algorithm 10 insertIfAbsent(k, v)

1: **while** *true* **do**
2: $\quad\langle a, aVer, p, pVer, res\rangle = search(k)$
3: $\quad$**if** *res* **then return** *false*;
4: $\quad$*kcas::start()*
5: $\quad n = createNode(p, k, v)$
6: $\quad$**if** $k > p\rightarrow key$ **then** *kcas::add(&p→right, NULL, n)*
7: $\quad$**else if** $k < p\rightarrow key$ **then** *kcas::add(&p→left, NULL, n)*
8: $\quad$**else continue**
9: $\quad$*kcas::add(&a→ver, aVer, aVer,*
10: $\qquad\qquad$*&p→ver, pVer, pVer + 2)*
11: $\quad$**if** *kcas::execute()* **then**
12: $\qquad$*rebalance(p)*
13: $\qquad$**return** *true*

At the time the *search* is validated, the correct location n to be inserted is a child of p. However, concurrent operations can change the tree such that this is no longer correct. For example, another operation can insert a new key between a and p's keys, a and p can be removed, or a and p can be rebalanced. These operations, however, always change either a or p (or both), changing their version numbers from

the values observed during *search* and guaranteeing the KCAS fails (See proof in Section 6.3 of this chapter for more details).

6.2.3 Deletion

The *erase* operation (Algorithm 11) first searches for the key to be removed similar to *insertIfAbsent*. If *search* does not find k, then false is returned. If *search* finds k, the node n containing k is returned, along with the parent of n, p. Since a search that finds k does not validate the path taken, *erase* ensures these nodes provided by search are not marked.[3] If an unmarked node with the key is found, the number of children n is counted and used to determine the erase operation used. Note that it is possible for the wrong operation to be chosen due to a concurrent update, e.g., n may gain or lose children during or after this step but before the KCAS is executed. However, since the version number of n was read as part of the search, any change also increments the version number of n, meaning the KCAS to remove it is guaranteed to fail, and the operation is retried.

Algorithm 11 erase(k)

```
1: while true do
2:     ⟨n, nVer, p, pVer, res⟩ = search(k)
3:     if not res then return false;
4:     if isMarked(pVer) or isMarked(nVer) then continue
5:     l = n→left
6:     r = n→right
7:     if l == NULL or r == NULL then res = eraseSimple(k, n, nVer, p, pVer)
8:     else res = eraseTwoChild(n, nVer, p, pVer)
9:     if res == SUCCESS then return true
```

The traditional two-child delete for a sequential BST is followed: the removed node is replaced by another node that preserves the properties of the tree. To reduce the number of changing fields, the *key* and *value* fields of the node to be removed are replaced, rather than replacing the actual node itself. To remove a key k in a BST, either the successor or predecessor of k can be promoted in its place, removing the old node that used to contain the predecessor or successor.

[3] In other concurrent BSTs false can be returned when a marked node containing k is observed, as this is sufficient to establish a time where k is not within the tree: the thread actually observed k's concurrent removal. However, in this tree, keys can be promoted up the tree by two-child erase, therefore the marking of a node containing k does not always mean that key k is erased. Hence, the operation must be retried.

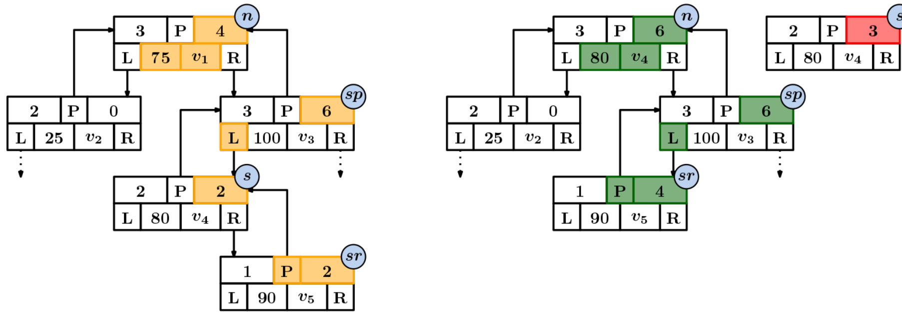

Figure 6.3: AVL Two-Child Erase

Algorithm 12 eraseTwoChild(n, $nVer$, p, $pVer$)

1: *kcas::start()*
2: $\langle s, sVer, sp, spVer, ret\rangle$ = *getSuccessor(n)*
3: **if not** *ret* ***or*** *isMarked(sVer)* ***or*** *isMarked(spVer)* **then**
4:     **return** *RETRY*
5: $sr = s \rightarrow right$
6: **if** $sr \neq NULL$ **then**
7:     $srVer = sr \rightarrow ver$
8:     **if** *isMarked(srVer)* **then return** *RETRY*
9:     *kcas::add(&sr→parent, s, sp,*
10:             *&sr→ver, srVer, srVer + 2)*
11: **if** $sp \rightarrow right == s$ **then** *kcas::add(&sp→right, s, sr)*
12: **else if** $sp \rightarrow left == s$ **then** *kcas::add(&sp→left, s, sr)*
13: **else return** *RETRY*
14: *kcas::add(&n→val, n→val, s→val,*
15:         *&n→key, key, s→key,*
16:         *&s→ver, sVer, sVer + 1,*
17:         *&sp→ver, spVer, spVer + 2)*
18: **if** $sp \neq n$ **then**   ▷ n and sp can be the same Node
19:     *kcas::add(&n→ver, nVer, nVer + 2)*
20: **if** *kcas::execute()* **then**
21:     *rebalance(sp)* ; **return** SUCCESS
22: **return** RETRY

It is arbitrary whether the predecessor or successor of k is chosen; this implementation promotes the successor. *getSuccessor* searches for the successor of n, which is the left-most node in the right subtree of n. *getSuccessor*'s return value is in the same format as *search*, returning two nodes, two version numbers, and a boolean. If the path to the successor found is successfully validated, *getSuccessor* returns the

successor node s, the parent of that node sp, the version numbers of those two nodes, and *true*: $\langle s, sVer, sp, spVer, true \rangle$. If the path is not successfully validated, the final item in the tuple is *false*, and the other values are ignored. Once a candidate node s is located, the path from n to s must be validated to establish a time where s is the successor node of n. Concurrent operations after this validation could insert keys between k and s' key, meaning s' key is no longer a valid replacement for k in n. However, if these operations change either n or s, the KCAS fails.

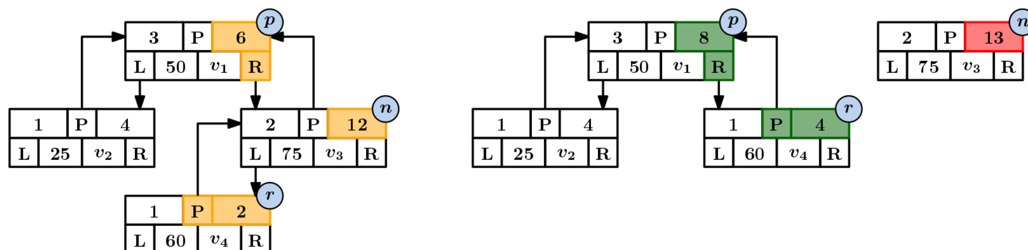

Figure 6.4: AVL One-Child Erase

To actually perform the removal of k, a KCAS is formed to change the key and value of n to the key and value of s, and perform the marking and unlinking of s. If s has one child, s is unlinked by changing the child field in sp that points to s so it points to s' child. If s is a leaf, it is unlinked by changing the child field of sp to NULL. The KCAS for this operation performs the removal of s and the promotion of s' fields to n in one atomic step.[4]

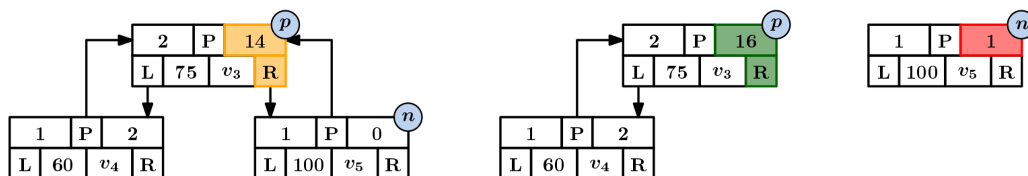

Figure 6.5: AVL Leaf Erase

Leaf nodes and nodes with a single child are erased by *eraseSimple* (Algorithm 13), which is much simpler than *eraseTwoChild*. In fact, this operation is almost identical to the removal of s in *eraseTwoChild* (Lines 11-17).

[4]Similar to the example presented in the introduction, other threads searching for s' key could be between n and s at the time of this operation, failing to find s' key during their traversal, despite it being present the entire time. However, path validation catches this issue: the version number of n changes with the change of its key, causing the search to try again.

Algorithm 13 eraseSimple(*key, n, nVer, p, pVer*)

```
1: kcas::start()
2: if n→left ≠ NULL then r = n→left
3: else if n→right ≠ NULL then r = n→right
4: else r = NULL
5: if r ≠ NULL then                                   ▷ n has one child
6:     rVer = r→ver
7:     if isMarked(rVer) then return RETRY
8:     kcas::add(&r→parent, n, p,
9:               &r→ver, rVer, rVer + 2)
10: if p→right == n then kcas::add(&p→right, n, r)
11: else if p→left == n then kcas::add(&p→left, n, r)
12: else return RETRY
13: kcas::add(&p→ver, pVer, pVer + 2,
14:           &n→ver, nVer, nVer + 1)
15: if kcas::execute() then
16:     rebalance(p)
17:     return SUCCESS
18: return RETRY
```

6.2.4 Rebalancing

This tree is approximately balanced in order to improve performance. Whenever *insertIfAbsent* or *erase* are successful, *rebalance(n)* (Algorithm 14) is invoked on the node n that either gained or lost children as part of the update. The rotations of a traditional sequential AVL tree are followed. A node is **balanced** if the logical heights of its children differ by less than 2. The entire tree is balanced if this holds for all nodes within the tree.

rebalance determines whether to perform a rotation on n, update its height, or do nothing based on its *apparent balance*. n's apparent balance is calculated by checking the heights of its children (Line 12). If n requires rebalancing, i.e., the apparent balance is $\geq +2$ or is ≤ -2, a direction is determined: a positive apparent balance indicates that n's left subtree is larger and requires a rotation in the opposite direction, and vice-versa. Depending on the balance of n's children, a single rotation may be insufficient to repair the imbalance of n. If this is the case, a double rotation is used, which involves applying a single rotation to one of n's children, and another one to n.

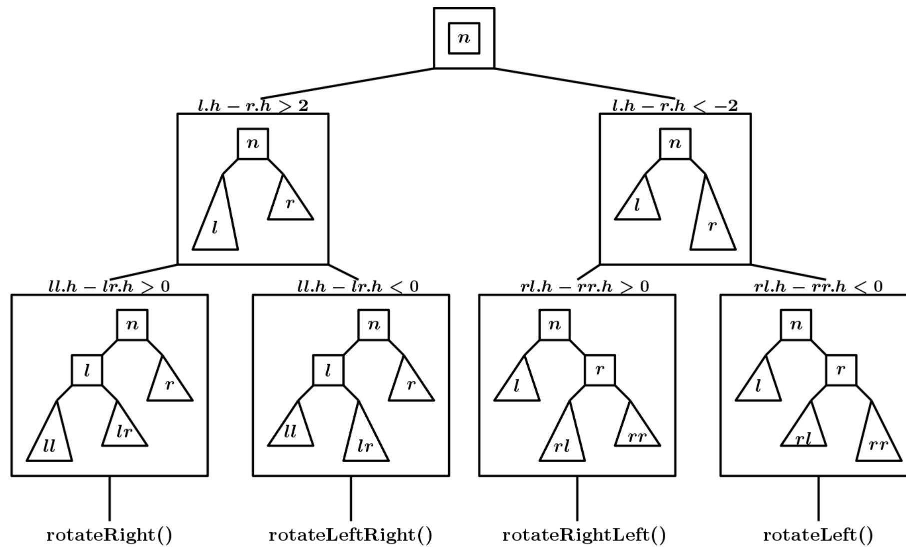

Figure 6.6: AVL Possible Rotations, *height* field is abbreviated to h for brevity

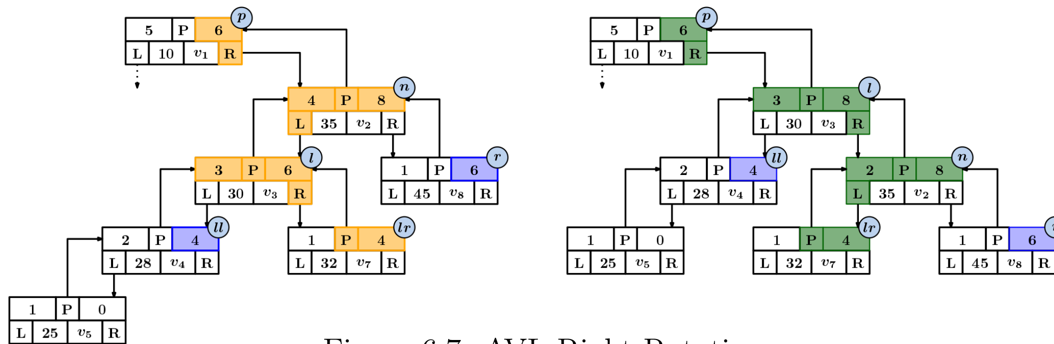

Figure 6.7: AVL Right Rotation

To carry out this rebalancing, the tree has the following operations: *fixHeight(node)*, *rotateLeft(node)*, *rotateRight(node)*, *rotateLeftRight(node)*, and *rotateRightLeft(node)*. *fixHeight(node)* updates the height field of a node that does not require rebalancing to be equal to the largest height of its children + 1, propagating updated height information up the tree. *rotateLeft(node)* and *rotateRight(node)* improve balance by shifting nodes in a particular direction. *rotateLeftRight(node)* and *rotateRightLeft(node)* are double rotations, as discussed above.

Algorithm 14 rebalance(n)

```
1: while n ≠ minRoot do
2:     nVer = n→ver
3:     if isMarked(nVer) then return
4:     p = n→parent
5:     pVer = p→ver
6:     if isMarked(pVer) then continue
7:     l = n→left
8:     if l ≠ NULL then lVer = l→ver
9:     r = n→right
10:    if r ≠ NULL then rVer = r→ver
11:    if isMarked(lVer) or isMarked(rVer) then continue
12:    lh = (l == NULL ? 0 : l→height); rh = (r == NULL ? 0 : r→height);
13:    nBalance = lh - rh
14:    if nBalance ≥ 2 then
15:        ll = l→left
16:        if ll ≠ NULL then llVer = ll→ver
17:        lr = l→right
18:        if lr ≠ NULL then lrVer = lr→ver;
19:        if isMarked(llVer) or isMarked(lrVer) then continue
20:        llh = (ll == NULL ? 0 : ll→height)
21:        lrh = (lr == NULL ? 0 : lr→height)
22:        lBalance = llh - lrh
23:        if lBalance < 0 then
24:            if rotateLeftRight(p, pVer, n, nVer, l, lVer, r, rVer, lr, lrVer) then
25:                rebalance(n); rebalance(l); rebalance(lr);
26:                n = p
27:        else if rotateRight(p, pVer, n, nVer, l, lVer, r, rVer) then
28:            rebalance(n); rebalance(l);
29:            n = p
30:    else if nBalance ≤ -2 then
31:        {...}                         ▷ Same as nBalance ≥ 2, l and r reversed
32:    else
33:        if (res = fixHeight(n, nVer)) == FAILURE then continue
34:        else if res == SUCCESS then n = n→parent
35:        else return
36: return
```

We note that the *height* field of a node is not always accurate, in that it is not always a correct representation of a node's logical height. However, Bougé et al. [5] prove that applying these rotation rules to a tree until none apply results in a strictly balanced AVL tree, even if some (or all) height fields of nodes are inaccurate. Our tree is strictly balanced in a quiescent state, i.e., when there are no ongoing

operations.

The simpler case in which only a single rotation is required (Line 24) is carried out by *rotateRight* (Algorithm 15, Figure 6.7) or *rotateLeft* (not shown, symmetric to *rotateRight*). In this example, *rotateRight*, moves n's left child, l, to n's position. Additionally, n replaces its left child pointer to l with l's right child, lr. New heights for all nodes involved are calculated based on this rotation and updated as part of the KCAS, except for p, which is checked for imbalance after this operation. Note that in Figure 6.7, the version numbers of ll and r are blue as they are dependency nodes not directly affected by the rotation. This dependency is because the heights of these nodes are used to calculate the new heights for other nodes, and the changing of these nodes can result in a rotation that does not improve the balance of the tree, if they change.

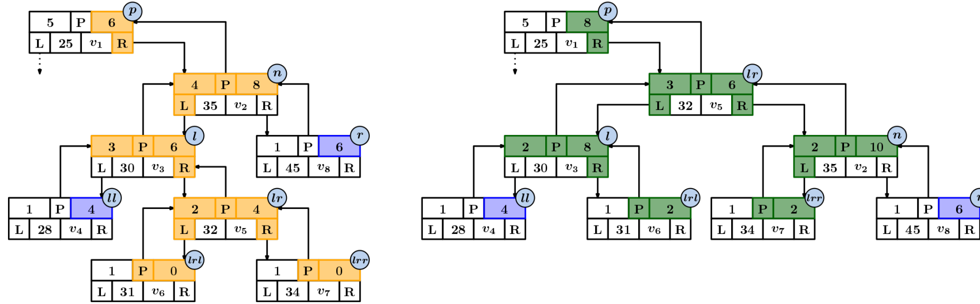

Figure 6.8: AVL Left-Right Rotation

Algorithm 15 rotateRight(*p, pVer, n, nVer, l, lVer, r, rVer*)

```
1: kcas::start()
2: if p→right = n then kcas::add(&p→right, n, l)
3: else if p→left = n then kcas::add(&p→left, n, l)
4: else return false
5: lr = l→right
6: lrHeight = 0
7: if lr ≠ NULL then
8:     lrVer = lr→ver; lrHeight = lr→height;
9:     if isMarked(lrVer) then return false
10:    kcas::add(&lr→parent, l, n,
11:              &lr→ver, lrVer, lrVer + 2)
12: ll = l→left
13: llHeight = 0
14: if ll ≠ NULL then
15:    llVer = ll→ver; llHeight = ll→height;
16:    if isMarked(llVer) then return false
17:    kcas::add(&ll→ver, llVer, llVer)
18: rHeight = 0
19: if r ≠ NULL then
20:    rVer = r→ver; rHeight = r→height;
21:    kcas::add(&r→ver, rVer, rVer)
22: oldNHeight = n→height
23: oldLHeight = l→height
24: newNHeight = 1 + max(lrHeight, rHeight)
25: newLHeight = 1 + max(llHeight, newNHeight)
26: kcas::add(&l→parent, n, p,
27:           &n→left, l, lr,
28:           &l→right, lr, n,
29:           &n→parent, p, l,
30:           &n→height, oldNHeight, newNHeight,
31:           &l→height, oldLHeight, newLHeight,
32:           &p→ver, pVer, pVer + 2,
33:           &n→ver, nVer, nVer + 2,
34:           &l→ver, lVer, lVer + 2)
35: if kcas::execute() then return true
36: return false
```

When a double rotation is required *rotateLeftRight* or *rotateRightLeft* is called. For simplicity, double rotations are combined into a single large KCAS. By combining these updates the need for updating the version number (and other fields) of nodes twice is removed.

If n does not require rebalancing, the accuracy of the height to its children is

ensured via *fixHeight*. If n's height needs to be updated, it is done via a simple KCAS, and *SUCCESS* is returned, meaning that rebalancing up the tree must continue as higher nodes must be made aware of this height change. If a time can be established when the height of n is accurate, then *fixHeight* returns *UNNECESSARY*, indicating that a balanced node has been reached and rebalancing can end. Rebalancing also ends if n is marked, or n is the *minRoot*.

insertIfAbsent and *erase* only affect the balance of a single node and its direct ancestors, which are reachable via the parent field of the node. However, rotations move nodes off of this parent pointer path, potentially making it impossible for threads to reach nodes need for rebalancing. Therefore, after a rotation is successful, *rebalance* is called on every node that had its *left* or *right* fields change as part of the rotation.

Algorithm 16 fixHeight(n, $nVer$)

```
1:  kcas::start()
2:  l = n→left
3:  r = n→right
4:  rVer = r→ver
5:  if l ≠ NULL then
6:      lVer = l→ver
7:      kcas::add(&l→ver, lVer, lVer)
8:  if r ≠ NULL then
9:      rVer = r→ver
10:     kcas::add(&l→ver, rVer, rVer)
11: oldHeight = n→height
12: newHeight = 1 + max(l→height, r→height)
13: if oldHeight == newHeight then            ▷ If the height is correct, no need to update
14:     if n→ver == nVer and (l == NULL or l→ver == lVer) and (r == NULL or r→ver ==
    rVer) then
15:         return UNNECESSARY
16:     else return FAILURE
17: kcas::add(&n→height, oldHeight, newHeight,
18:             &n→ver, nVer, nVer + 2)
19: if kcas::execute() then return SUCCESS
20: return FAILURE
```

Algorithm 17 rotateLeftRight(*p, pVer, n, nVer, l, lVer, r, rVer, lr, lrVer*)

```
1: kcas::start()
2: if p→right = n then kcas::add(&p→right, n, lr)
3: else if p→left = n then kcas::add(&p→left, n, lr)
4: else return RETRY
5: lrl = lr→left; lrlHeight = 0;
6: if lrl ≠ NULL then
7:     lrlVer = lrl→ver; lrlHeight = lrl→height;
8:     if isMarked(lrlVer) then return RETRY
9:     kcas::add(&lrl→parent, lr, l,
10:                &lrl→ver, lrlVer, lrlVer + 2)
11: lrr = lr→right; lrrHeight = 0;
12: if lrr ≠ NULL then
13:     lrrVer = lrr→ver; lrrHeight = lrr→height;
14:     if isMarked(lrrVer) then return RETRY
15:     kcas::add(&lrr→parent, lr, n,
16:                &lrr→ver, lrrVer, lrrVer + 2)
17: rHeight = 0;
18: if r ≠ NULL then
19:     rVer = r→ver; rHeight = r→height;
20:     kcas::add(&r→ver, rVer, rVer)
21: ll = l→left; llHeight = 0;
22: if ll ≠ NULL then
23:     llVer = ll→ver; llHeight = ll→height;
24:     if isMarked(llVer) then return RETRY
25:     kcas::add(&ll→ver, llVer, llVer)
26: oldNHeight = n→height; oldLHeight = l→height; oldLRHeight = lr→height
27: newNHeight = 1 + max(lrrHeight, rHeight)
28: newLHeight = 1 + max(llHeight, lrlHeight)
29: newLRHeight = 1 + max(newNHeight, newLHeight)
30: kcas::add(&lr→parent, l, p,
31:            &lr→left, lrl, l,
32:            &l→parent, n, lr,
33:            &lr→right, lrr, n,
34:            &n→parent, p, lr,
35:            &l→right, lr, lrl,
36:            &n→left, l, lrr,
37:            &n→height, oldNHeight, newNHeight,
38:            &l→height, oldLHeight, newLHeight,
39:            &lr→height, oldLRHeight, newLRHeight,
40:            &lr→ver, lrVer, lrVer + 2,
41:            &p→ver, pVer, pVer + 2,
42:            &n→ver, nVer, nVer + 2,
43:            &l→ver, lVer, lVer + 2)
44: if kcas::execute() then return SUCCESS
45: return RETRY
```

6.3 Correctness Proof

Definition 6.3.1. The **search path** to a key k is the path an atomic search(k) traverses.

Definition 6.3.2. The **pred node** (resp., succ node) of a key k is the node containing the largest key smaller than k (resp., smallest key larger than k).

Definition 6.3.3. A node is **in the data structure** if it is reachable from maxRoot.

Observation 6.3.4. In a BST that does not contain k, the search path to k contains the pred and succ nodes of k. One of these nodes is the last node on the search path.

Observation 6.3.5. In an internal BST, a node containing key k is inserted as a child of the pred or succ node of k.

See [13] and [14] for more details on Observation 6.3.4 and 6.3.5.

Lemma 6.3.6. No successful KCAS ever modifies the fields of marked nodes.

Proof. This follows from inspection of the code. Before reading any other field of a node, the version number of this node is read. If this node is to be involved in the KCAS, this version number is checked to ensure the node is not marked, and the version number is included in the KCAS. If the node is marked, the operation is retried. If the node is not marked when it is checked, but is marked between this check and the KCAS, the KCAS fails, i.e., the marking of a node directly involves the changing of its version number. □

Lemma 6.3.7. Our implementation of a relaxed AVL tree satisfies the following claims.

1. The node *maxRoot* always has *minRoot* as its left child, and NULL as its right child. The node *minRoot* has NULL as its left child. (Remark: the right child points to the rest of the tree.)

2. Consider any search, where $r_1 ... r_l$ is the sequence of nodes visited by it so far. For each r_i in $r_1 ... r_l$, there is a time during the search when r_i is in the data structure.

3. Consider an invocation I of *validatePath(path, vers, size)* in an *insertIfAbsent(k, v)*, *erase(k)*, or *contains(k)*. If I returns true, then *path* is the search path to k just before I.

4. (a) The tree rooted at the right child of *minRoot* is always a relaxed AVL tree.
   (b) Any *insertIfAbsent* or *erase* operation that performs a successful KCAS returns the same value if it is performed atomically at its linearization point (the successful KCAS).
   (c) Any *insertIfAbsent* or *erase* operation that terminates without performing a successful KCAS returns the same value if it is performed atomically at its linearization point.

Proof. Consider an arbitrary execution E. We prove these claims together by induction on the sequence of steps $s_1, s_2, ...$ in E, which can be shared memory reads, atomic KCASRead operations, or atomic KCAS operations.

Base case: Before any KCAS is successful, the tree is in its initial state where two nodes exist: *minRoot* and *maxRoot*. *maxRoot* has the key $+\infty$, its left child is *minRoot*, and its right child is NULL. *minRoot* has the key -∞, and both its children are NULL.

Inductive step: suppose the claims all hold before step s. We prove they hold after step s.

Claim 1. This configuration is the initial state of the tree, except the right child of *minRoot* can no longer be NULL. No operation can modify this state. *minRoot* and *maxRoot* contain keys that are never part of any operation: no operation searches for, attempts to remove, or attempts to insert these keys. Additionally, these nodes are never rebalanced: it is explicit in the code that rebalancing stops when it reaches the *minRoot*, which also means *maxRoot* also is never rebalanced.

Claim 2. The only step that can affect this claim is a KCASRead in a search that traverses to a new node, by reading a pointer from r_l to some r_{l+1}. This pointer is then added it to *path[]*, the sequence of nodes visited so far. So, s is a KCASRead at line 20 or 23.

By the inductive hypothesis r_l is in the data structure at some time before s during the search. If r_l still points to r_{l+1} at step s, then since r_l is in the data structure and points to r_{l+1}, so is r_{l+1}. Otherwise, r_l is deleted before s during the search, and right before this deletion r_l pointed to r_{l+1} (Lemma 6.3.6). Therefore, r_{l+1} is in the data structure at the time of this deletion, which is during the search.

Claim 3. Only *successful* KCAS operations can affect this claim, as reads do not modify the structure of the tree. A search reads and stores the version numbers of all nodes encountered during a search. If any of these nodes change between reading as part of the search and validation as part of *validatePath*, *validatePath* returns false. If *validatePath* returns true, it is guaranteed that no modification occurred to **any** node along the path between when it is read and when it is validated as part of *validatePath*. At time t, just before I, all nodes are read as part of the search but not validated. Hence, if I returns true, all the nodes still had the same version numbers read as part of the search at t and this path is an *atomic snapshot* of the search path to k, just before the invocation of I, i.e., before any nodes were validated.

Claim 4a. Only *successful* KCAS operations that change the layout of the tree can affect this claim. KCAS operations are performed by *insertIfAbsent*, *eraseSimple*, *eraseTwoChild*, *rotateLeft*, *rotateRight*, *rotateLeftRight*, and *rotateRightLeft*. We proceed by cases.

Case 1: Suppose s is a successful KCAS at line 11 of *insertIfAbsent*. This KCAS is only executed if a successful path validation occurred. Let t be the time just before the successful invocation of *validatePath* started. Claim 3 means that this path **is** the search path to k at time t. Let $pred_t$ be the predecessor node of k a time t, and $succ_t$ be the successor node of k at time t. The node we modify to insert this new node is the final node on the search path at time t, denoted by $last_t$. Note that, from Observation 6.3.4 and 6.3.5, $last_t$ is either $pred_t$ or $succ_t$. The version numbers of both $pred_t$ and $succ_t$ are added to the KCAS, so if they change after t, the KCAS fails. We wish to prove that a successful KCAS inserts this new node as a child of $last_s$, i.e., the last node on the search path to k at s. In other words, we wish to prove that because the KCAS succeeds, $last_t == last_s$. Suppose that $last_t \neq last_s$ to obtain a contradiction. If the correct place to insert the new node changes from as a child of $last_t$ to as a child of some other node $last_s$, this must happen as a result of an *erase*, *insertIfAbsent* or one of the rotations (searches do not change the data structure). We show that any of these operations that make it so $last_s$ is any node other than $last_t$, then that results in a failed KCAS.

Subcase 1: Suppose an *insertIfAbsent* operation inserts a key outside the range of $[pred_t \rightarrow \text{key}, succ_t \rightarrow \text{key}]$. If this operation modifies any field of $pred_t$ or $succ_t$, the KCAS fails. If it does not, from Observation 6.3.4 and 6.3.5, $pred_t$ and $succ_t$ still contain the predecessor and successor of k at s ($pred_t = pred_s$ and $succ_t = succ_s$). Therefore, k should still be inserted as a child of one of these nodes. At the time of validation, one of these nodes is $last_t$, and the only one of the two to have a NULL

pointer in the appropriate child field for the insertion of k. For example, if $pred_t$ $= last_t$, then $pred_t$→right = NULL and $succ_t$→left ≠ NULL. Conversely, if $succ_t$ = $last_t$, then $succ_t$→left = NULL and $pred_t$→right ≠ NULL. This insertion does not change these nodes, so $last_s$ is still the one of the two nodes that k can be inserted as a child of at t. In other words, $last_t = last_s$. If a key in the range [$pred_t$→key, $succ_t$→key] is inserted, by Observation 6.3.5 this must be added as a child of either $pred_t$ or $succ_t$. This operation must modify $pred_t$ or $succ_t$, incrementing their version number, causing the KCAS to fail.

Subcase 2: Suppose an *erase* operation removes a key outside the range of [$pred_t$→key, $succ_t$→key]. This scenario follows the exact same argument for an *insertIfAbsent* operation inserting a key outside this range, in Subcase 1 above. If an *erase* removes a key within the range of [$pred_t$→key, $succ_t$→key], it must be either $pred_t$→key or $succ_t$→key, as they are the only keys within this range at t (or something else was inserted, see Subcase 1). This removal would modify $pred_t$ or $succ_t$, causing the KCAS to fail.

Subcase 3: Performing a rotation that does not modify either $pred_t$ or $succ_t$ does not change where k should be inserted. No such rotation changes the predecessor or successor nodes of k, so k should still be inserted as a child of either $pred_t$ or $succ_t$ at s. As stated in Subcase 2, only one of $pred_t$ and $succ_t$ have a NULL pointer in the proper child field to allow for the insertion of k at t, which is node $last_t$. A rotation can cause this situation to be reversed, meaning if $last_t = pred_t$ then $last_s = succ_t$ (or vice versa), but this modifies $pred_t$ and $succ_t$, causing the KCAS to fail.

We prove this for each possible modification inductively, and conclude that because the KCAS is successful, $last_t = last_s$ after any number of steps between t and s.

Case 2: Suppose s is a successful KCAS at line 20 of *eraseTwoChild* (we omit the KCAS of *eraseSimple* as it is strictly easier). In *erase*, search locates a node n that contains the key k at t, and the version number of n at t is added to the KCAS. Let t' be the time just before the invocation of *validatePath* in *getSuccessor*. *getSuccessor* locates the successor of k at t', $succ_{t'}$, by traversing the tree from n and validating the path taken. We want to prove that because the KCAS succeeds, n is a two child node containing k and that we replace the key of s with the successor node of k at s ($succ_s$) meaning $succ_{t'} = succ_s$. If n or the successor of k changes, this must happen as a result of an *erase*, *insertIfAbsent* or one of the rotations (searches do not change the data structure). We show that any of these operations that makes n's key not equal to k, changes n's number of children, or makes it so $succ_s$ is any node other than $succ_{t'}$, causing the KCAS to fail.

Subcase 1: Suppose an *insertIfAbsent* operation inserts a key outside the range of [n_t→key, $succ_{t'}$→key]. This action does not change the successor of k, meaning $succ_{t'} = succ_s$. If this operation modifies any field of n or $succ_{t'}$, the KCAS fails, and n still contains k and has two children. Suppose an *insertIfAbsent* operation inserts a key inside the range of [n_t→key, $succ_{t'}$→key]. From Observation 6.3.5, this new key must be inserted as a child of either n or $succ_{t'}$, which modifies either n or $succ_{t'}$, causing the KCAS to fail.

Subcase 2: Suppose an *erase* operation removes or modifies n, meaning k is either removed or k is promoted to another node. This operation modifies n and causes the KCAS to fail. Similarly, any erase that modifies $succ_{t'}$ also causes the KCAS to fail. If an erase does not modify n or $succ_{t'}$, then n still contains k and has two children. If there are no new keys between k and $succ_{t'}$→key, then $succ_{t'} = succ_s$. We know that there are no such keys inserted (Subcase 1, above).

Subcase 3: Suppose a rotation modifies n or $succ_{t'}$, this modification causes the KCAS to fail. If the rotation does not modify n or $succ_{t'}$, n still contains k and has two children. Additionally, rotations do not add or remove keys, meaning $succ_{t'} = succ_s$.

We prove this for each possible modification inductively, and conclude that because the KCAS is successful, $succ_{t'} = succ_s$ after any number of steps between *t'* and *s*.

Case 3: Suppose *s* is a successful KCAS of any rotation. We want to prove that these rotations preserve the in-order traversal of all nodes modified by this rotation and their subtrees. Note that all the above cases are difficult because we had to prove that the result of a search at time t (or *t'*) still applied at time *s*. However, there is no search portion of these rotations, and they are simply sequential AVL tree rotations made atomic via KCAS. It is possible that fields could change after they are read to determine which rotation to apply, but as part of the rotations *every node read is involved in the KCAS, so any changes result in a failed KCAS.*

CLAIM 4B. In Claim 4a, we actually proved that all operations that execute a successful KCAS are atomic at time *s*, which is the time when the KCAS is executed.

CLAIM 4C. *insertIfAbsent* only returns without performing a KCAS if *search* locates a node that has the key k. From Claim 2, the key k that we are trying to insert is in the data structure at some time t during the search. Time t is during *insertIfAbsent*, so we linearize this operation at t, returning false.

erase only returns without performing a KCAS if *search* does not locate a node that has the key k. By Claim 3, the key k that we are trying to remove is not in the data structure at some time t during the search. Time t is during *erase*, so we linearize this operation at t, returning false.

The linearization points for the operations are as follows:

- *insertIfAbsent*
  - returning *true*: at the successful KCAS at line 11 of *insertIfAbsent*
  - returning *false*: the time t during search where k is in the data structure, from Claim 2
- *erase*
  - returning *true*: at the successful KCAS of *eraseSimple* at line 15 or *eraseTwoChild* at line 20, whichever is used
  - returning *false*: the time t just before the invocation of *validatePath* in search, from Claim 3
- *contains*
  - returning *true*: the time t during search where k is in the data structure, from Claim 2
  - returning *false*: the time t just before the invocation of *validatePath* in search, from Claim 3

□

Theorem 6.3.8. Our relaxed AVL tree implements a linearizable dictionary.

Proof. Lemma 6.3.7 proves all the tree operations are atomic and do not violate any relaxed AVL-tree properties. Furthermore, rebalancing steps preserve the in-order traversal of the tree and do not add or remove any keys from the data structure. Therefore, searches are equivalent to an atomic search at some time t during *search*. *contains* is simply linearized at this time t during *search*. □

6.4 Progress Proof

We wish to prove that, if processes take steps infinitely often, then the tree operations succeed infinitely often. In essence, this conjecture is shown by the lock free nature of the KCAS implementation, the fact that operations are only forced to retry when they observe a change, and the bounded number of rebalancing steps required to resolve imbalance in the tree, as shown by [5].

THEOREM 6.4.1. The relaxed AVL-Tree is lock free

Proof. Consider some configuration C after which some threads continue to take steps, but no operations complete. Eventually, in this scenario, the tree stops changing, as the only operations that can change the tree are successful operations and rebalancing steps. [5] shows that regardless of how imbalanced the tree is or how inaccurate the height fields of nodes are, the number of rebalancing steps to completely balance an arbitrary tree is bounded. Hence, at some point, rebalancing must end and the state of the tree must stop changing.

In this static tree, *search* must succeed after a finite number of steps: no concurrent operations can invalidate the paths taken in a search, i.e., all paths taken to key k are the correct search path to k. Hence, no *contains* operations are invoked after C, as this is the only required part of this operation. Thus, eventually, only *insertIfAbsent* and *erase* continue to make steps, continually looping in their retry loops. However, the only way for these operations to be forced to retry is if the value read at some address during the operation changes at the time of the KCAS, causing the KCAS to fail. The KCAS of operations in this algorithm only fail if some value read during the operation changes between the read of the operation and the execution of the KCAS. This change, however, implies that other operations are succeeding. In other words, in order for infinitely many KCAS operations to fail, infinitely many KCAS operations must succeed.

□

6.5 Balance Proof

A node n has a **violation** if:

- n→left→height - n→right→height > 2; **or**

- n→left→height - n→right→height < -2; **or**
- n→height ≠ 1 + max(n→left→height, n→right→height)

We wish to prove that, when all operations on the tree are completed, the tree is a **strict** AVL-tree. We make no guarantees about the balance of the tree while operations are ongoing. The rotations used within this tree are originally discussed in [5], where it is proven that a generic AVL tree is balanced if applicable rotations or *fixHeight* are applied to all nodes with violations until no violations exist. Therefore, we need to prove that whenever a violation is created, the thread that created it repairs this violation, or another thread repairs it for them.

THEOREM 6.5.1. The tree is a strict AVL-tree when in a quiescent state.

Proof. Operations that can create violations are *insertIfAbsent*, *erase*, rotations, and *fixHeight*.

insertIfAbsent, *erase*, rotations, and *fixHeight* can only create violations on the single node directly above where the operation takes place. For *insertIfAbsent* it is the parent of the new node n inserted. For *erase* it is the parent of the node removed, where the node removed is either the actual node containing k or the successor node of k that is removed in its place. Rotations and *fixHeight* either move a new node into n's place with a different height or update n's height to a new value, both of which can create a violation at whatever node is the parent of n before the operation is executed. No other nodes generate violations due to any operation. After all these operations, *rebalance* is called on the node that could potentially contain a violation and it is fixed if it exists.

Rotations change the layout of the tree and nodes often lose **ancestors** (nodes along their parent pointer path to the root) as part of these rotations. These nodes could potentially have unresolved violations, but are no longer be reachable by the threads doing operations lower in the tree that caused the violation. To resolve this issue, any rotation that removes an ancestor from a set of nodes immediately calls *rebalance* on that node fixing all the violations of that node and all its ancestors. This action effectively takes responsibility for fixing the violations in that parent pointer chain to the root starting at that node. Once a thread begins rebalancing, it does not stop until it has fixed violations at every node it is responsible for, and nodes it is no longer responsible for are to be rebalanced by another thread. □

6.6 Evaluation

In this section we evaluate the performance of the AVL tree. The benchmark environment is a Non-Uniform Memory Architecture (NUMA) system with four CPUs (Intel Xeon Platinum 8160 3.7GHz). Each CPU has 24 cores with two hardware threads per core (hyperthreading) for a total of 192 hardware threads. Cores on the same CPU share a 33MB L3 cache. We reserve 2 hardware threads for system processes, to avoid unnecessary context switching during experiments. Threads are *pinned* to cores and only run on one physical CPU (NUMA node) at thread counts up to 48, on two physical CPUs at thread counts up to 96, and so on. For all algorithms, jemalloc 5.0.1-25 is used to allocate memory. Code is compiled with GCC 7.4.0-1, with the highest optimization level (-O3) We used `numactl --interleave=all` to distribute memory pages evenly across NUMA nodes. Memory is reclaimed for all data structures using DEBRA, a fast epoch-based reclamation algorithm [7].

The data structures compared in the experiments are as follows:

- **BRONSON** [6] is a partially-external lock-based BST that uses optimistic concurrency control (OCC) for searches.
- **NATARAJAN** [28] is a lock-free external BST created using CAS and bit test and set (BTS).
- **ELLEN** [15] is a lock-free external non-blocking BST using CAS.
- **DRACH** [13] is a partially external lock-free BST (in which we found a correctness bug as part of this work).
- **DAVID** [12] is an external BST that uses ticket locks.
- **BROWN** [10] is a chromatic tree implemented using LLX/SCX.
- **KCAS** is the AVL tree presented within this work, using the KCAS implementation from [3].
- **KCASHTM** is the AVL tree presented within this work, using a hardware transactional memory based KCAS.

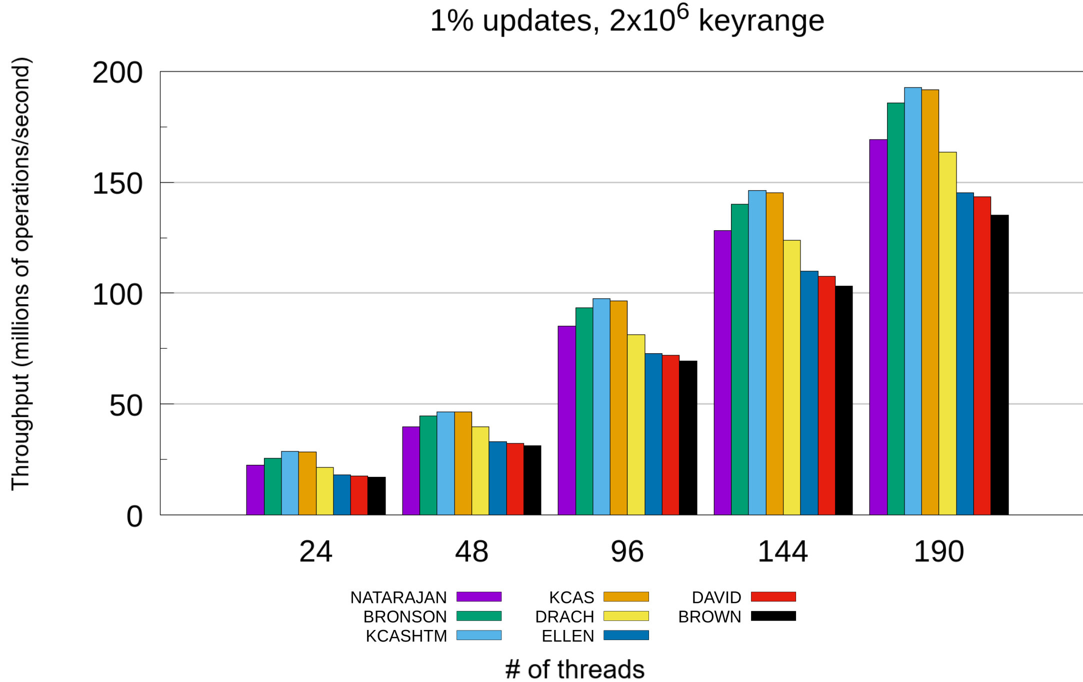

Figure 6.9: **Throughput** comparison with additional BSTs.

6.6.1 Throughput

The AVL tree was compared against all the implementations outlined above, as shown in Figure 6.9. For all workloads, BRONSON, NATARAJAN and the KCAS trees from this work outperformed all other trees. For readability, only those experiments are included in the rest of the results. (Other trees are consistently outperformed by BRONSON or NATARAJAN.)

These four implementations were tested with a variety of update rates (0%, 10% and 40%) and key-range sizes ($2 * 10^6$, $2 * 10^7$, $2 * 10^8$, $2 * 10^9$). Updates are evenly distributed between *insertIfAbsent* and *erase*. For example, in a 10% update workload, 5% of operations are *insertIfAbsent* and 5% of operations are *erase*, the rest are *contains*. Figures 6.10 - 6.13 shows these experiments, showing how all algorithms scale under these workloads with additional threads.

These experiments show, despite the higher cost of KCAS, that the KCAS implementation outperforms Natarajan in all workloads (except for the non-HTM KCAS

with keyrange size of $2 * 10^6$ and an update rate of 40%) and are competitive with Bronson in all workloads, slightly surpassing Bronson in lower update-rate cases. The KCAS implementation suffers when contention is high, specifically in high update-rate cases with smaller key ranges.

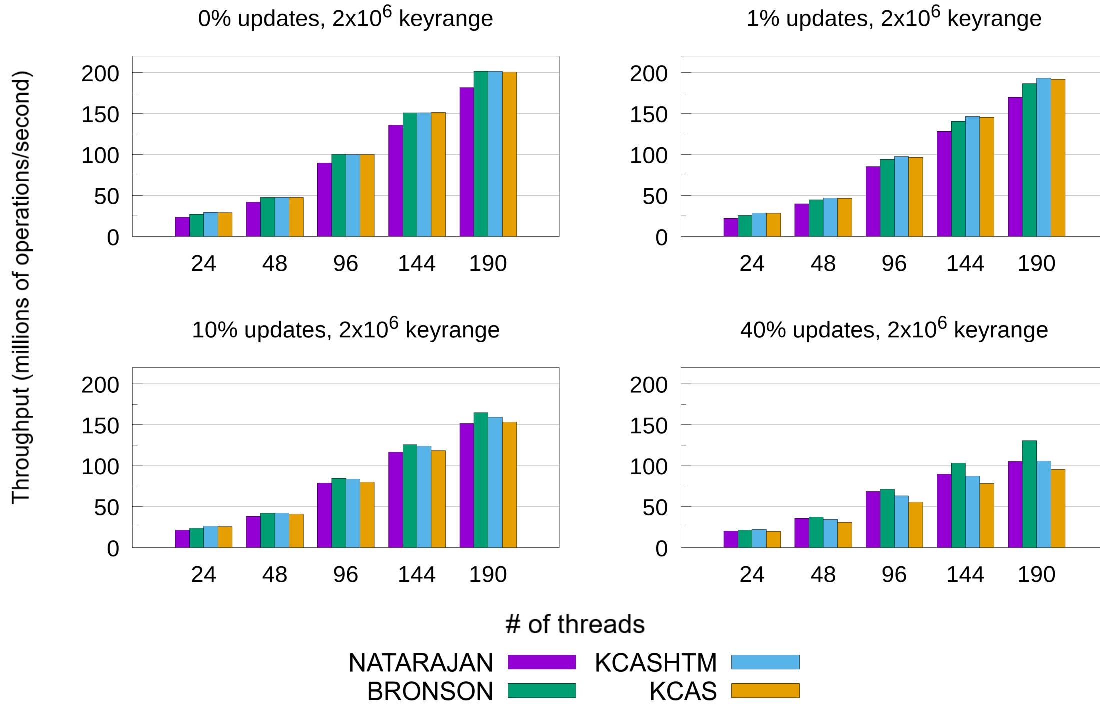

Figure 6.10: AVL operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^6$

6.6.2 Key Depth and Cache Misses

The expectation is that KCAS would be significantly slower than a fine-grained technique, as it requires significantly more writes. However, due to the internal nature of this tree, the average key depth tree is lower than other trees, meaning searches require fewer reads to locate a specific key, and therefore less cache misses are expected. The number of last level cache (LLC) misses per operation is shown in Figure 6.14.

Managing paths in the KCAS implementations is not free, as threads must read and save several fields as they traverse, something the other trees do not require.

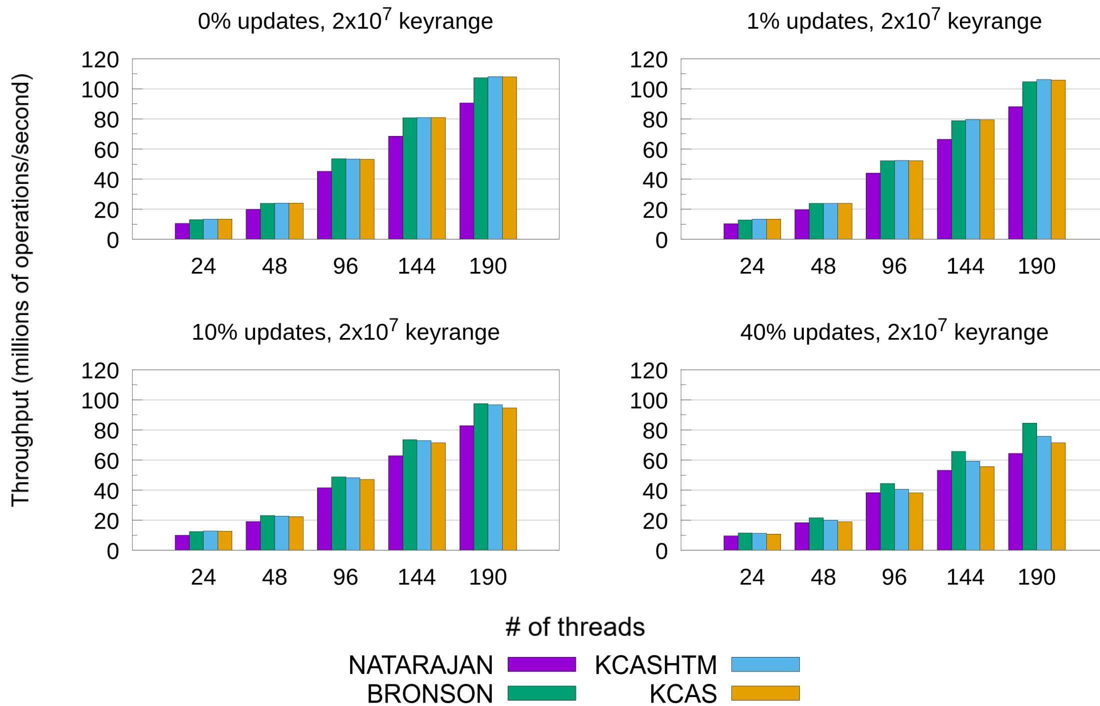

Figure 6.11: AVL operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^7$

However, whenever Bronson traverses a new node for a search, it involves a recursive call so that searches can be partially recovered rather than requiring a full restart. As a result, every traversal requires the creation of a stack frame to do the recursive call, which comes at a significant cost. Conversely, the overall cost of *validatePath*, is quite low: it is expected that the fields that are validated as part of *validatePath* are already in L1 cache, as they are read to do the search. *validatePath* also returns on the first node that has a changed version number. It is expected that a cache miss would occur on this first node, but then the search would be retried. Note that Bronson is also susceptible to an L2 prefetching issue caused by its validation technique, which causes additional cache misses [4].

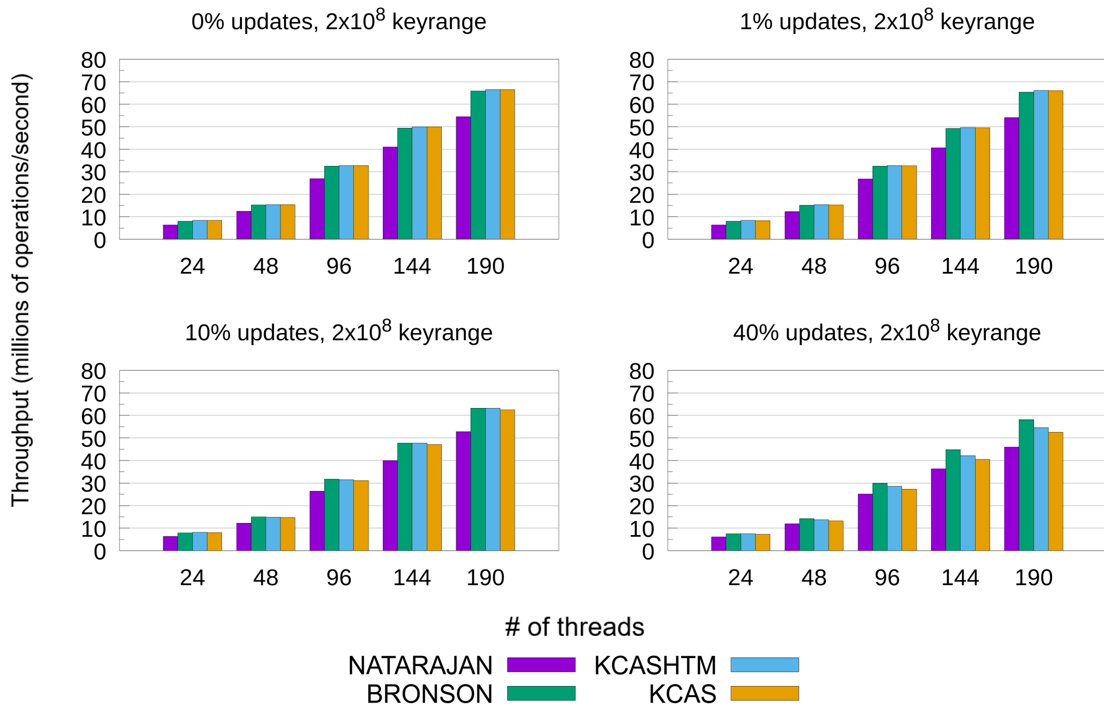

Figure 6.12: AVL operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^8$

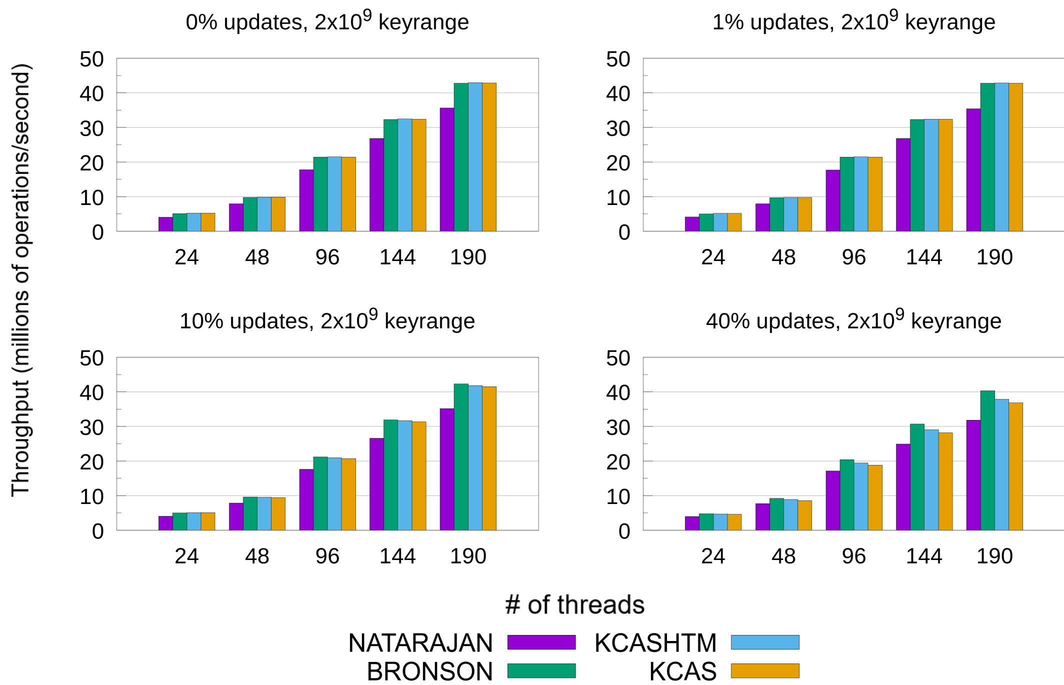

Figure 6.13: AVL operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^9$

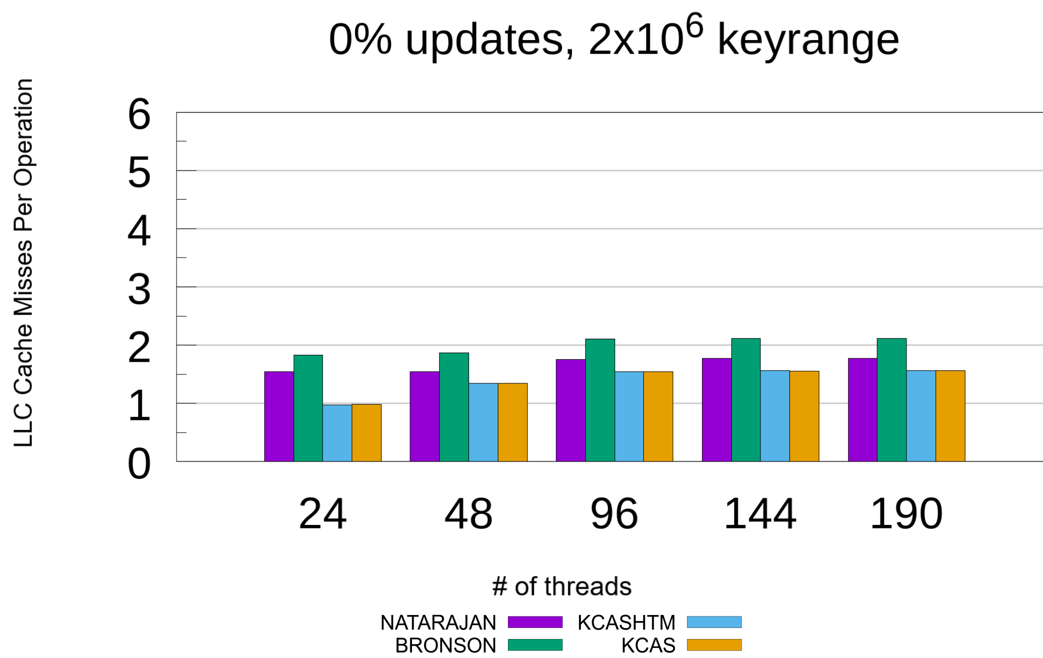

Figure 6.14: AVL LLC misses per operation comparison, lower is better

Chapter 7

Dynamic Connectivity via KCAS

The approach outlined in the previous chapter can be applied to a variety of node-based data structures. As an example, this chapter gives an overview of the first lock-free concurrent solution to the dynamic connectivity problem on undirected acyclic graphs. To the best of our knowledge, this is the first concurrent implementations of such a data structure.

7.1 Overview

The dynamic connectivity problem involves maintaining a graph containing a set of fixed vertices and a dynamic set of edges. Solving dynamic connectivity requires implementing three operations: *connected(v, w)*, *link(v, w)* and *cut(v, w)*. *connected(v, w)* returns true if there exists a path from node v to w. Otherwise, it returns false. If there is no path between v and w, *link(v, w)* creates an edge between them and returns true. Otherwise, it returns false.[1] Finally, if there is an edge between v and w, *cut(v, w)* removes this edge and returns true. Otherwise, it returns false.

We follow the same general approach used to implement the AVL tree: all nodes have version numbers, and follow **Rule 1** and **Rule 2**, and the *validatePath* operation is used to implement atomic searches.

[1] You cannot link two nodes if there already exists a path between them, as this creates a cycle in the graph. This restriction is a common limitation of *sequential* data structures for dynamic connectivity, which does not relate to our method.

7.2 Algorithm Design

In the sequential setting, Euler Tours [32] are typically used to implement dynamic connectivity. An Euler Tour starts at an arbitrary node and visits each edge exactly once (interpreting undirected edges as two directed edges for our purposes) recording each visit to a node as they are traversed.

In the classical Euler tour data structure, the tour is stored in a BST. However, this work stores the tour in a skip list rather than a BST, as in [36], which makes it easier to split and merge while maintaining (probabilistic) balance, and having each node of the skiplist represent an *edge* in an Euler tour rather than a node [33]. To clarify, there is a graph comprised of graph nodes, and a skiplist comprised of list nodes, which represent *edges* in the graph, and each graph node has pointers to the list nodes for each of its incident edges. That paper also adds an additional self-edge for each node, which appears as a list node pointed to by the appropriate graph node. It turns out that this self edge greatly simplifies the data-structure operations in a concurrent setting. Figure 7.1 shows an example of how such tours can be represented. We omit the upper levels of the skip-list to save space, and draw the graph representation above the list. To avoid special cases, towers of sentinel nodes are added to be beginning and the end of every tour list.

Just as we used version numbers to impose a sequential ordering on modifications to a single BST node in our AVL tree, the version number of the leftmost sentinel node at the bottom level of the list (the **minimum sentinel**) is used to impose a sequential ordering on modifications to an individual *Euler tour list*, i.e., a single version number protects the *entire* tour list. More precisely, all updates increment the version number of the minimum sentinel, allowing only a single update on any list at a time. This approach might seem like a concurrency bottleneck, but care is taken to avoid the possibility of cycles being introduced by concurrent *link*s. Additionally, every graph node is initially in *its own* Euler tour tree, allowing plenty of concurrency. We now sketch how the operations are implemented.

7.2.1 Connection Queries

The main purpose of this data structure is to answer connectivity queries: does a path between nodes v and w exist or, equivalently, do v and w belong to the same connected component or tour list? Consider the self-edges from v to v, and from w to w. Let L_v and L_w be the list nodes that represent these self-edges. If a time can be established when L_v and L_w are in the *same* tour list, then a path exists between

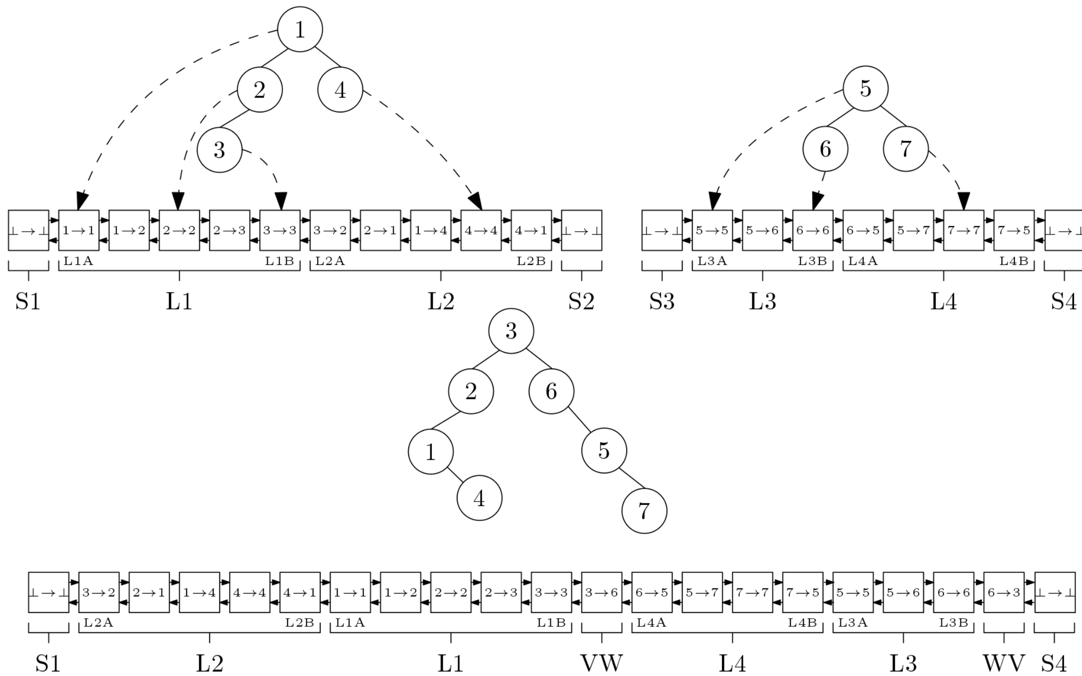

Figure 7.1: Operation *link*$(3, 6)$ on (simplified) Euler tour lists

those two graph nodes at that time. Conversely, if a time can be established when these list nodes are in **different** tour lists, then no path exists between the graph nodes at that time.

This observation is simple to determine: starting from L_v and L_w, the tour list(s) are traversed left towards the minimum sentinel. These traversals do the reverse of a traditional skiplist search, traversing up and left in the list until a sentinel node is reached. Once a sentinel node is reached, the node is traversed down towards the bottom level of the list to locate the minimum sentinel. (We could simply have traversed left, without going upwards, but then our traversal time could be linear in the size of the tour list.) The paths taken by both traversals are then validated, and if either validation fails the entire operation is retried. If the *same* minimum sentinel is found by these validated searches then there is a time when these two graph nodes existed in the same subgraph and therefore a path existed between them, so true is returned. If they are different, a time exists where the graph nodes are in different subgraphs and no path existed between them, so false is returned.

7.2.2 Link

To simplify the presentation, each tour list is presented as a doubly-linked list (rather than skiplist), then is extended to explain how this changes when skiplists are used. The goal of *link(v,w)* is to add a link from graph node v to graph node w, absorbing the subgraph (equivalently, tour list) of w into the subgraph of v. v and w can only be linked if they are not part of the same subgraph, so this operation begins by performing the algorithm for *connected(v,w)*. Let M_v and M_w be the minimum sentinels located while performing this algorithm. If M_v and M_w are the same, then *link(v,w)* can safely return false: a time is determined where they were already in the same subgraph. If M_v and M_w are different, then the operation can proceed. We include M_v and M_w in our (eventual) KCAS operation, using it to increment both of their version numbers.

We explain the next steps with an example. Consider the case *link(3, 6)* presented in Figure 7.1. The tour lists drawn at the top of that figure are logically split into two *sublists* each: sublist *L1* contains all nodes in the tour list to the left of and including 3's self-edge (excluding the minimum sentinel, called S1), and sublist *L2* contains all nodes to the right of 3's self-edge (excluding the **maximum sentinel**, S2). Similarly, in the tour list for node 6, sublist *L3* contains all nodes in the tour list to the left of and including 6's self-edge (excluding the minimum sentinel, called S3), and sublist *L4* contains all nodes to the right of 6's self-edge (excluding the maximum sentinel, S4). The labels L1, L2, L3 and L4 are suffixed with A and B to denote the beginning and end of sublists (i.e., L1A is the leftmost node of L1, and L1B is the rightmost node of L1). These nodes require updates as part of the operation.

Since *link* adds a new edge, that edge is added to the tour lists (twice, as it should be traversed in both directions). Two new list nodes are created (VW and WV), one for each direction. The resulting tour list can be constructed by arranging the sublists in the following order: [S1, L2, L1, VW, L4, L3, WV, S4],[2] which requires the operation to change the *left* or *right* pointers of the nodes on the ends of the sublists, as well as those of sentinel nodes. This arrangement effectively *rotates* the individual tours containing v and w such that they are rooted at v and w, respectively, and then links them together. Note that the graph nodes for 3 and 6 are also updated to add their new neighbour (6 and 3, resp.) to their adjacency lists.

All of these pointer changes are performed in a single KCAS. In other words, the

[2] It is possible for L2 and L4 to be empty, in which case, the same sequence sublist order works if empty sublists are omitted.

KCAS needs to update the left and right fields of all the list nodes at the ends of the sublists, add neighbours to the graph nodes, increment the version numbers of all nodes involved (crucially, including the minimum sentinel), and *mark* any nodes that are removed (S2 and S3, in this case). We use *marking* to avoid erroneous modifications to deleted nodes. Before a KCAS is performed, it is first verified that every node included in the KCAS is not marked. If a node marked, the entire operation is restarted.

In a skiplist, this list restructuring is simply repeated at *every level*, in one large KCAS. The relevant sublists are determined at *each level* by traversing starting from the bottom list, and are rearranged in the same order as the bottom list. Crucially, updates to a skiplist based tour list are still serialized on the same field: the version number of the minimum sentinel.

To determine a sublist at level $i + 1$ from level i, the skip-list is traversed *upwards* and *inwards* from the ends of the sublist at level i. For example, consider the top left image in Figure 7.1, let the bottom level of the list represented here be level 1. To determine the sublist L1 at level 2, the skip-list is traversed from L1A right until a node is encountered that has a node above it at level 2. Similarly, to determine the other end of the sublist, the skip-list is traversed from L1B *left* until a node is encountered that has a node above it at level 2. The two nodes found at level 2 are the ends of the sublist L1 at level 2. If these two traversals ever encounter the same node at some level i, this indicates there is no such sublist at level $i + 1$. By performing this traversal for every sublist, at every level, all the nodes that need to be modified can be found. This process is repeated until the maximum height of the skiplist is reached, or the sublist does not exist at some level, i.e., if a sublist does not exist at some level, it does not exist at any level above that as well.

7.2.3 Cut

The goal of our implementation of *cut(v, w)* is to remove the edge connecting v to w if it exists, and split their tour list into two. Graph nodes contain adjacency lists, so determining if two nodes are directly connected by an edge is easy. If they are not neighbours, then the operation returns false. Otherwise, the version numbers of these graph nodes should be added to our (eventually) KCAS. The minimum sentinel is located as in the previous operations but only a single traversal starting at one of v or w is required.

From the graph nodes in Figure 7.2, the list nodes representing the edges VW and WV can be found. These list nodes are removed as part of the operation and

the list nodes between them form one of the new tour lists. The list is separated into *three* sublists: L1, which contains all nodes to the right of the minimum sentinel and to the left of the edge VW, L2, which contains all nodes to the right of VW and to the left of WV, and L3, which contains all nodes to the right of WV and to the left of the maximum sentinel (S2). Two new sentinel nodes are created for the new list, S3 and S4. This operation simply removes L2 from the center of the list, creating two lists as a result: [S1, L1, L3, S2] and [S3, L2, S4].[3] The sublist L2 represents the nodes no longer reachable from v after the removal of w, since the only way v could reach these nodes is by first traversing w.

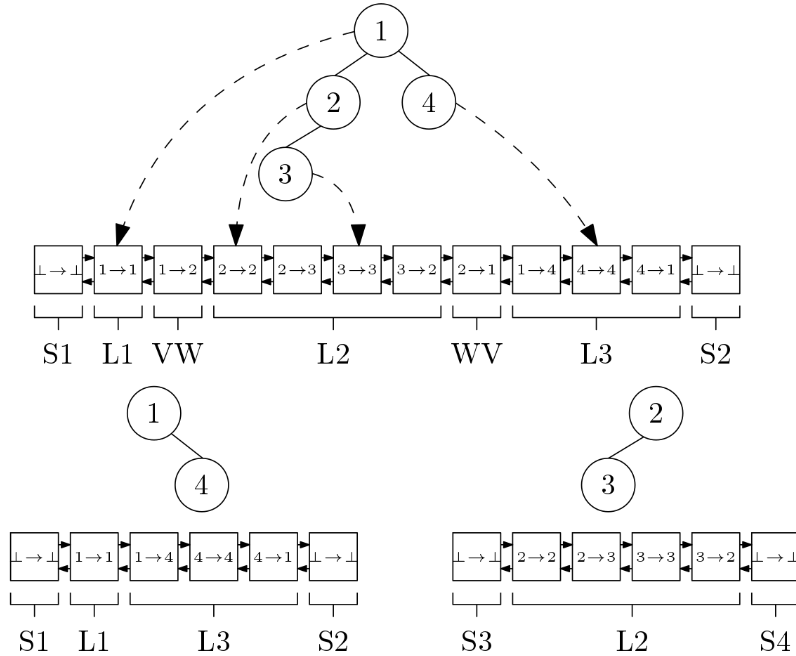

Figure 7.2: Operation $cut(3, 6)$ on (simplified) Euler tour lists

To extend this to a skiplist it is very similar to *link*. These sublists are formed at each level and linked together in the same order as the bottom list.

[3] One of L1 or L3 (but not both) could be empty, but the order presented here remains correct if empty lists are omitted.

7.3 Correctness Proof

Note: recall that to avoid special cases, an additional self-edge is added for each graph node. Once this is done, there is still a well defined Euler tour that follows each edge once, but now these self-edges appear in the Euler tour. If an Euler tour containing self-edges has all of its self-edges removed,an Euler tour of the original graph remains.

As in the avl tree, list nodes are marked at the time they are deleted, so it can be verified (purely syntactically) that before performing a KCAS that all nodes it modifies are unmarked, and in doing so guarantee that no deleted node is ever changed.

DEFINITION 7.3.1. This *fully-dynamic connectivity* data structure consists of a set of Euler tour skiplists and a set of graph nodes. Each graph node u participates in a single Euler tour skiplist (or tour list for short), and contains pointers to all of the (skip)list nodes that represent directed edges starting from u, including the self edge $u \to u$. In each tour list, the bottom level list nodes represent the sequence of edges visited in an **Euler tour** of the graph nodes that participate in the tour list.

Algorithm 18 *isConnected(v, w)*

```
1: while true do
2:     p_v = traversePathToMinSentinel(v)          ▷ A path p_v followed to a minimum sentinel
3:     p_w = traversePathToMinSentinel(w)          ▷ A path p_w followed to a minimum sentinel
4:     if not validatePath(p_v) then continue
5:     if not validatePath(p_w) then continue
6:     if p_v→minSent == p_w→minSent then return true    ▷ paths arrived at the same sentinel
7:     else return false                           ▷ paths arrived at different minimum sentinels
```

OBSERVATION 7.3.2. Any list node that has the value NULL in both its down and left field is the **minimum sentinel** of a tour list.

LEMMA 7.3.3. Our implementation satisfies the following claims:

1. *isConnected(v, w)* returns the same value as if performed atomically at its linearization point (just before the first validation) .

2. (a) The data structure is a fully-dynamic connectivity structure (see definition 7.3.1) .

(b) Any *link* or *cut* operation that performs a successful KCAS returns the same value as if it were performed atomically at its linearization point (the KCAS)

(c) Any *link* or *cut* operation that terminates without performing a successful KCAS returns the same value as if it were performed atomically at its linearization point

Proof. Consider an arbitrary execution E. We prove these claims together by induction on the sequence of steps $s_1, s_2, ...$ in E, which can be shared memory reads, atomic KCASRead operations, or atomic KCAS operations.

Base case: There are a finite number i graph nodes, and each is in its own tour list. These tour lists contain a single self-edge, and two sentinel towers on each side.

Inductive step: suppose the claims all hold before step s. We prove they hold after step s.

CLAIM 1. The only operations that can affect this claim are KCAS operations from *link* or *cut*. Reads do not change the data structure, hence they do not change the paths followed by the traversals in *isConnected*.

Subcase 1: Consider an invocation of *isConnected(v, w)* that returns true. In the final loop of this invocation, the following occurs: the traversal from v to a minimum sentinel that follows path p_v occurs, then the traversal from w to the same minimum sentinel follows path p_w. Let the time this second traversal ends be t_0. From Observation 7.3.2, we can statically check that the node reached by these traversals is a minimum sentinel. We then validate the path of the first traversal at t_1, and then validate the path of the second traversal at t_2. Therefore, $t_0 < t_1 < t_2$). This operation does not return unless *validatePath* returns true for both paths. Since we know that *validatePath(p_w)* returns true, there is no modifications to any nodes in p_w between t_0 and t_2. Hence, there is also no modifications to any node in p_w between t_0 and t_1, and *validatePath(p_w)* returns true if it executed at t_1. Consider a *link* or *cut* update that changes the configuration of the tour list during this operation. If this *link* or *cut* does not involve the current tour list, it does not change the minimum sentinel that is reached by either traversal. If these updates occur on the current tour list, it must include the version number of the minimum sentinel in the KCAS. If this update occurs during one of the traversals, then the traversal fails to validate, and this operation is retried. Additionally, if the update occurs between the traversals and one of the validations, the validation fails. Therefore, since both of these traversals end at the same minimum sentinel, they are in the same tour list just before t_1, which is where we linearize this operation.

Subcase 2: Consider an invocation of *isConnected(v, w)* that returns false. This argument is the same as Subcase 1, since *isConnected(v, w)* still must validated both paths, however the minimum sentinels are different.

CLAIM 2A. The only operations that affect this claim are the KCAS operations in *link* and *cut*, as reads to not change the data structure.

Subcase 1: Suppose s is a successful KCAS of *link(v, w)*. Before this KCAS, two traversals occurred and validated that form the self-egdes of the two graph nodes v and w to two different minimum sentinels. As we proved in Claim 1, a time t exists where v and w are in different tours, and hence there is no path between them. From the inductive hypothesis, these two tour lists are well-formed before this operation. Hence, it is correct to perform a *link* operation on these two nodes at t. Since the version number of both minimum sentinels are part of this KCAS, there are no changes to either tour list between t and s. This observation means that no update modified any node in either tour list after the time they are validated. Therefore, the *link* operation is still applicable at s.

Subcase 2: Suppose s is a successful KCAS of *cut(v, w)*. This case is easier than Subcase 1, as only a single list is tracked for this operation.

CLAIM 2B. This claim is proven in Claim 2A, as we proved that both *link* and *cut* are atomic at s, which is when the KCAS is executed.

CLAIM 2C. This claim is proven in Claim 1, as both use the result of *isConnected* to determine if a KCAS should execute or not. If *link* calls *isConnected* and it returns true, *link* returns false, as a time t established a path already existed between the two nodes (right before the first validation of *isConnected*), we linearize this operation at t. The argument is identical for *cut*, but reversed.

□

7.4 Progress Proof

THEOREM 7.4.1. Our implementation of the fully-dynamic connectivity structure is lock-free.

Proof. Consider some configuration C, where threads continue to take steps, but after some time t no operations complete. All threads, therefore, must be stuck in retry loops, failing *validatePath* or KCAS operations. Since *validatePath* and KCAS operations only fail if a node is modified since its version number is last read, the only way these infinite retry loops fail is if there are infinitely many modifications

to the data structure. However, if operations stop completing after time t, then eventually the data structure must stop changing, since each operation performs at most one successful KCAS, and the data structure is changed exclusively by successful KCAS operations. This observation is a contradiction, the only way a thread fails an operation is if another thread has made progress.

□

Chapter 8

(a,b)-tree via KCAS

The implementation of the AVL tree in Chapter 6 performed worse relative to the competition when update rates were high (> 20%). There are data structures, however, where our approach can actually provide superior performance in high update-rate workloads relative to the current state of the art. As an example of such a data structure, we give a brief overview for an implementation of an (a,b)-tree using our framework that actually *improves* update performance.

8.1 Overview

An (a,b)-tree is a leaf-oriented search tree, similar to leaf-oriented B-trees, with the extra invariant that all leaves have between a and b keys and internal nodes have between a and b child pointers. The exception is the root, which has between 1 and b keys if it is a leaf or 2 and b child pointers if it is an internal node. This (a,b)-tree is also *relaxed* similar to the AVL tree within this work: the operations and subsequent rebalancing required are not done in a single atomic operation. Hence, during an execution, the tree may become imbalanced, but at the end of the execution, where there are no ongoing operations, the tree satisfies the normal B-tree balance invariant.

The original implementation of this tree [8] is implemented using a "tree update template", which relies on load-link-extended (LLX) and store-conditional-extended (SCX). The "tree update template" makes heavy use of the read-copy-update (RCU) paradigm in order to perform all updates. With KCAS, however, this is not necessary for all operations and directly results in performance benefits for some operations. LLX/SCX primitives are extended versions of the traditional load-link (LL) and

store-conditional (SC) primitives. LL returns the current value stored at a memory location. After this, a SC attempts to store a new value at the same memory location only if no changes have occurred to that memory location since the LL.[1] LLX can be called on multiple **data records** (such as a node) before an invocation of SCX. SCX verifies multiple memory locations have not changed since the associated invocation of LLX, and can mark any number of data records, but it can only change a single memory location after the verification.

The focus of the rest of this chapter is to show how changing an implementation of a data structure to use KCAS, and making some other small changes to conform to our framework, can actually **improve** the performance of updates. Most operations of the tree follow the same semantics as the LLX/SCX version, however do so via KCAS. For brevity, how every operation is carried out is not outlined, but only the changes required to the original implementation to use KCAS, and how some operations changed in order to accommodate KCAS.

8.2 Algorithm Design

Beause LLX/SCX only change a single field at a time, the original implementation relies heavily on a read-copy-update style for making changes to a data structure, allocating new nodes even if the new nodes allocated are very similar to those being replaced. One common operation is insertion. Consider an insertion of a new key-value pair into a leaf that has space for it, i.e., no split or join operations are required to insert this key. The original implementation carried out this insertion case as depicted in Figure 8.1.

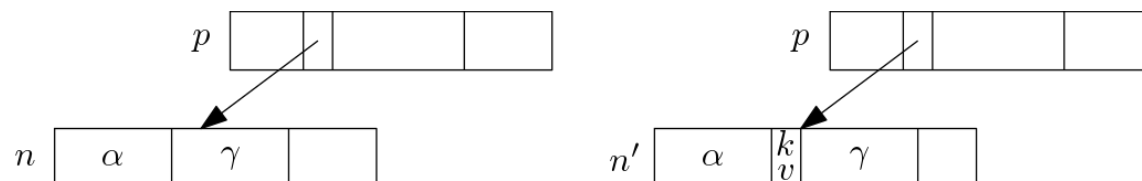

Figure 8.1: Simple insert case on the LLX/SCX implementation of the (a,b)-tree

Node n is the leaf node in which the new key-value pair (k,v) is to be inserted.

[1]This is similar to CAS, however CAS only ensures the memory location has the same value as the one provided, while LL/SC will fail if **any change** occurs to that memory location, therefore being resilient to ABA problems.

The key-value pairs in this diagram are split into two sets: α, which are the key-value pairs with keys less than the new key k to be inserted, and γ, which are the key-value pairs with keys greater than the new key k to be inserted. In essence, this operation attempts to insert (k,v) between these two sets. LLX/SCX is insufficient to insert the multi-word new pair in-place, as it can only change a single field in memory. Hence n is replaced with a new node. In the current state of the tree n's parent, node p has a pointer to n. This pointer to n is changed to point to a new node, n, that contains all the keys and values of n and the new pair (k,v) in the appropriate location. Even in the case where the set γ is empty, LLX/SCX still cannot avoid this allocation because the key and value are two separate fields.

In the updated implementation using KCAS, a slight change to how searches work to avoid special cases is required. The keys of leaves are left unsorted[2], meaning the insertion discussed above can be carried out as depicted in Figure 8.2.

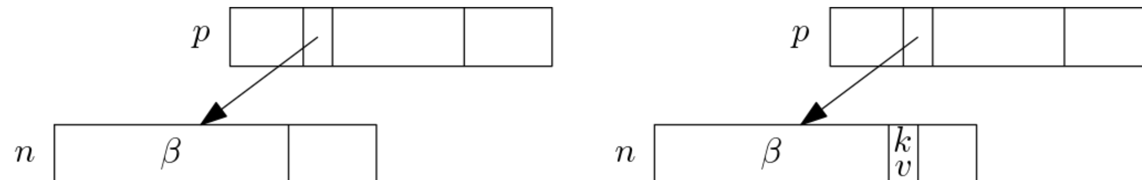

Figure 8.2: Simplest insert case on the KCAS implementation of the (a,b)-tree

Since keys are unsorted, the set β of keys is simply all the keys of n in no particular order. To carry out this insertion, the new pair (k,v) is simply inserted into the node n and n is not replaced, avoiding a new node allocation. On the surface, this seems like a minimal change. However, as seen in the next section, avoiding this allocation actually has substantial impact on performance.

To accommodate the unsorted nature of leaf nodes, a small change to the searching algorithm is required. When a leaf node is encountered when searching for a key, all keys in the node must be read and compared to the key searched for, until it is found or no keys remain. We also make changes to this data structure in order to follow our framework, augmenting every node with a version number and validating searches as in the AVL tree. All updates increment this version number, and searches are validated by re-reading version numbers of nodes encountered during a search.

The only operation changed drastically is the singular insert case and searches. Other operations are changed to be carried out via KCAS, but are semantically the

[2]Due to the bounds on the size of nodes here (between a and b keys), and the fact that only leaves are left unsorted, this does not change the time complexity of any operation.

same as the LLX/SCX version, except they also increase a version number. The correctness arguments are very similar to the AVL tree (actually, they are strictly simpler) and are therefore not restated here.

8.3 Evaluation

The KCAS and LLX/SCX implementations are compared to show the performance differences under different workloads. A similar approach is followed to the experiments from the AVL tree, however we change some of the update-rates in order to highlight the benefits of using KCAS.

In Figures 8.3 - 8.6 a comparison between the two implementations is shown. It can be seen clearly that the searches in the LLX/SCX implementations are somewhat faster, which is most likely due to the extra cycles spent performing KCAS reads in our implementation, and the small cost of path validations. However, as the update-rate increases the performance of the LLX/SCX implementation fails to scale as well as the KCAS implementation. The essential difference in the KCAS variant is that it removes the need for the allocation of a new node for the insert case outlined above, and the copying of all the values from the old node to the new one. The following experiments show that while the additional overhead of KCAS relative to LLX/SCX does affect of performance as expected, in high update-rate scenarios avoiding the RCU-style updates has a much larger impact.

8.4 Possible Extensions

It should be noted that it is possible to avoid validating searches in this data structure and use a similar argument to show searches are correct from the original implementation. However, omitting this validation greatly increases the complexity of the required proof, making it similar to that of the original LLX/SCX implementation, while only providing a small benefit to performance (<1% in all tested workloads).

Additional operations could be changed more drastically thanks to the increased power of KCAS, allowing almost all updates to many fields of a node to be changes rather than replacements. This approach would require algorithmic changes and an increase in complexity relative to the original implementation, though could result in performance benefits.

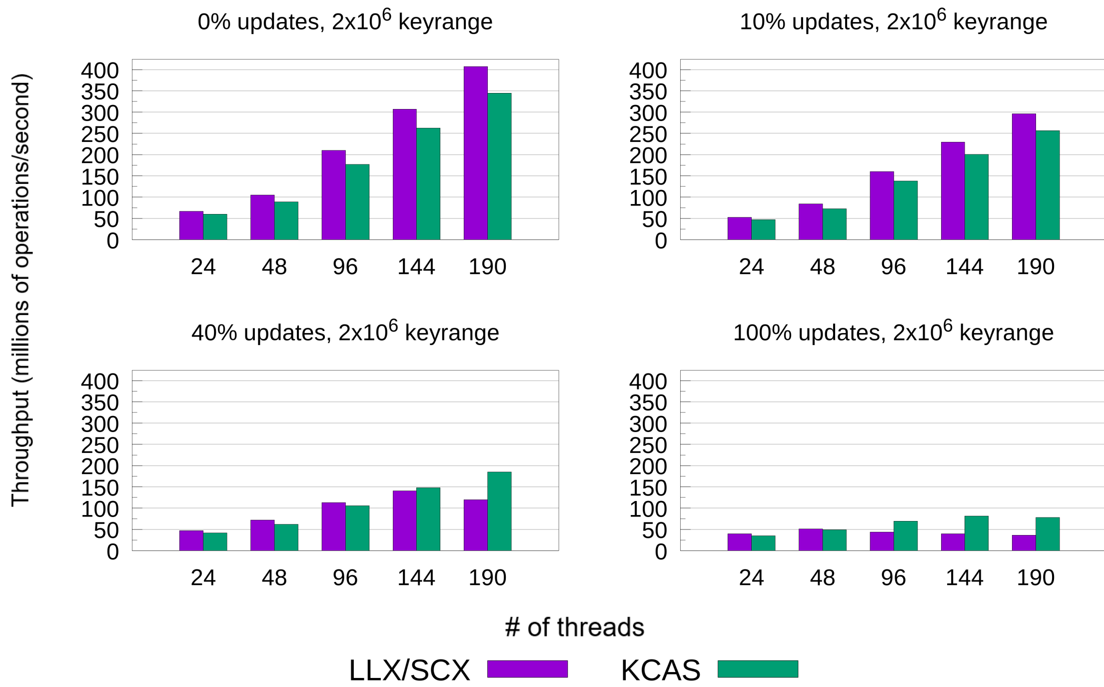

Figure 8.3: (a,b)-tree operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^6$

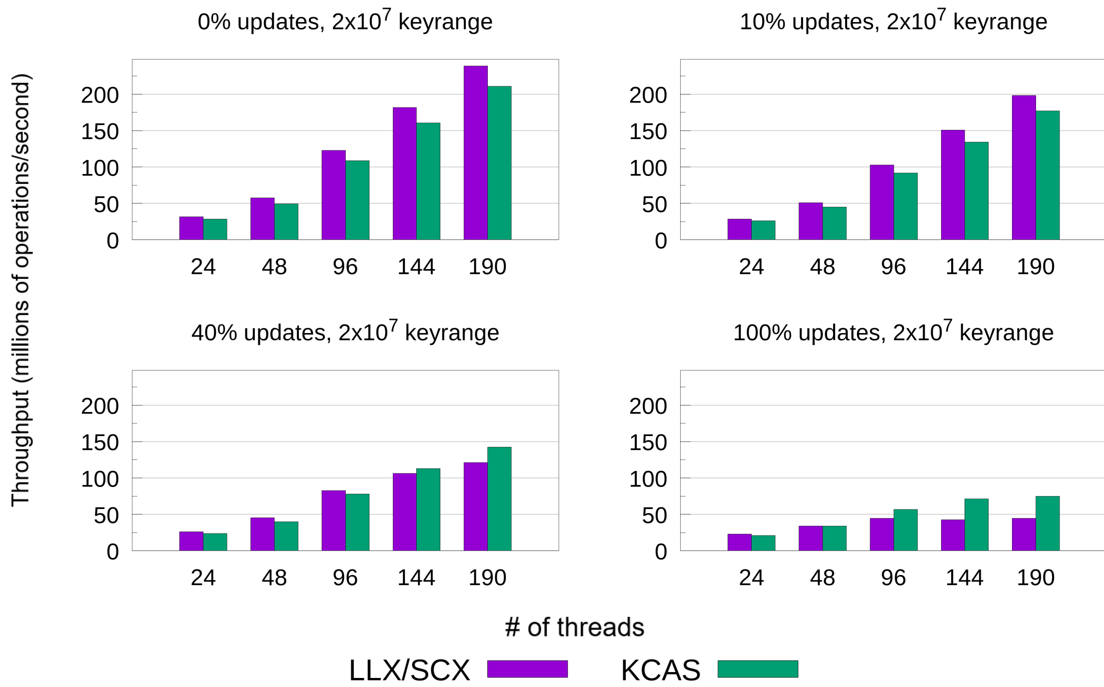

Figure 8.4: (a,b)-tree operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^7$

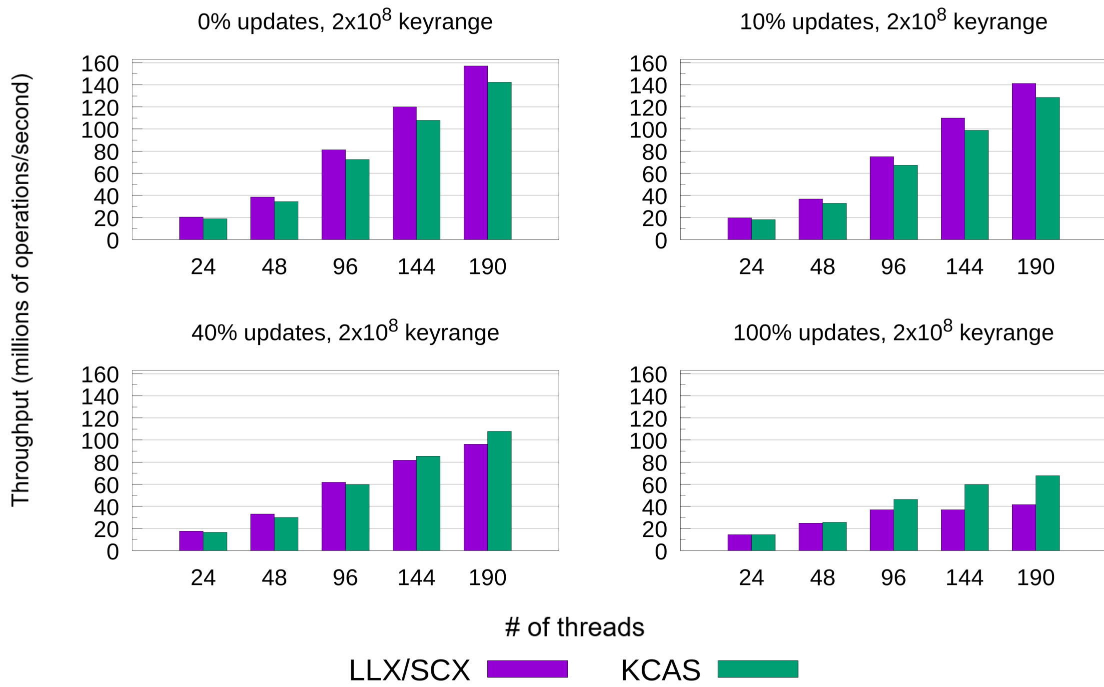

Figure 8.5: (a,b)-tree operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^8$

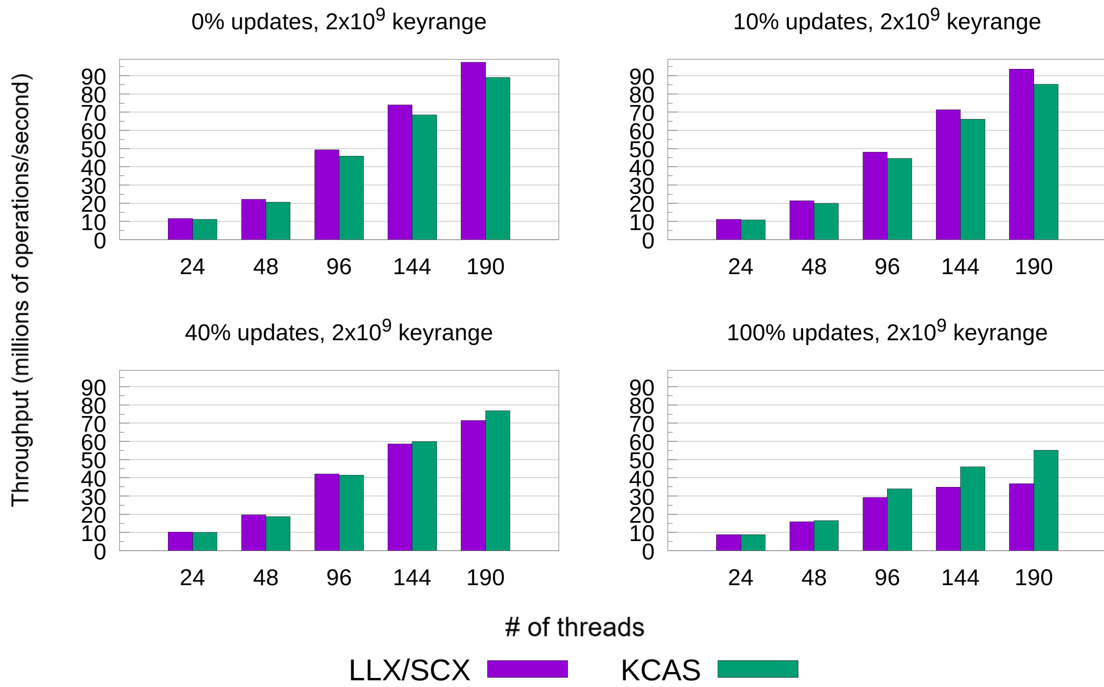

Figure 8.6: (a,b)-tree operation throughput comparison at varying thread counts, in millions of operations per second, higher is better. Keyrange: $2 * 10^9$

Chapter 9

Conclusion

9.1 Summary

This thesis introduces a KCAS-based framework for designing efficient concurrent data structures, and showed how it can be applied to a variety of data structures, two of which are the first of their kind. The AVL tree presented is competitive with the state-of-the-art in concurrent BSTs, despite being significantly simpler and using a higher-level primitive like KCAS. The first implementation of a concurrent fully-dynamic connectivity data structure is also presented, notably achieving lock-free progress. Finally, a KCAS variant of the (a,b)-tree implemented with LLX/SCX presented in [4] is outlined, which is able to achieve higher update performance by avoiding the use of RCU-style updates from the original implementation.

9.2 Future Work

9.2.1 Node-based KCAS

The use of KCAS and this approach can be integrated, further abstracting the use of KCAS via a *node-based* KCAS implementation. In this implementation, the user would interact with data structure nodes exclusively through the KCAS interface (including allocation and reclamation). When preparing a KCAS operation, the user would provide a set of nodes they wish to change, the changes to those nodes they wish to apply, and the set of dependency nodes for the operation. The use of version

numbers can be abstracted away from the user, making the implementation of these data structures significantly simpler, and potentially have performance benefits for certain data structures.

9.2.2 Combining Data Structures

Combining data structures is an approach used to achieve specific time complexities for certain workloads. For example, a hash-list (the combination of a hash-table and a doubly-linked-list) provides expected constant-time removal and membership tests, at the cost of more expensive insertions. Creating such data structures concurrently can be difficult with fine-grained approaches, as both data structures must be synchronized. Using KCAS, however, operations on both data structures can be combined relatively simply, resulting in simple implementations of very specialized data structures. For example, to carry out an insertion operation on a hash-list, it would be as simple as constructing the KCAS for insertion for both the hash-table and linked-list separately, then combining those KCAS operations into one single larger KCAS.

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993.

[2] James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, page 184–193, New York, NY, USA, 1995. Association for Computing Machinery.

[3] Maya Arbel-Raviv and Trevor Brown. Reuse, don't recycle: Transforming lock-free algorithms that throw away descriptors. In *Proceedings of the 31st ACM Symposium on Distributed Computing*, 2017.

[4] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. Getting to the root of concurrent binary search tree performance. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 295–306, Boston, MA, July 2018. USENIX Association.

[5] Luc Bougé, Joaquim Gabarró Vallés, Xavier Messeguer Peypoch, and Nicolas Schabanel. Height-relaxed avl rebalancing: a unified, fine-grained approach to concurrent dictionaries. 1998.

[6] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, New York, NY, USA, 2010. ACM.

[7] Trevor Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 261–270, New York, NY, USA, 2015. ACM.

[8] Trevor Brown. *Techniques for Constructing Efficient Lock-free Data Structures.* PhD thesis, University of Toronto, 2017.

[9] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, pages 13–22, 2013.

[10] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 329–342, 2014.

[11] Intel Corporation. Intel architecture instruction set extensions programming reference, chapter 8, 2013.

[12] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 631–644, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *PPoPP '14 Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 343–356, 2014.

[14] Dana Drachsler-Cohen, Martin Vechev, and Eran Yahav. Practical concurrent traversals in search trees. *SIGPLAN Not.*, 53(1):207–218, February 2018.

[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] Rachid Guerraoui and Vasileios Trigonakis. Optimistic concurrency with optik. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 18:1–18:12, New York, NY, USA, 2016. ACM.

[17] Tim L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 300–314, 2001.

[18] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, page 265–279, Berlin, Heidelberg, 2002. Springer-Verlag.

[19] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[20] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[21] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[22] Shane Howley and Jeremy Jones. A non-blocking internal binary search tree. *ACM Symposium on Parallelism in Algorithms & Architectures*, 06 2012.

[23] Robert Kelly, Barak A. Pearlmutter, and Phil Maguire. Concurrent robin hood hashing. In Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira, editors, *22nd International Conference on Principles of Distributed Systems (OPODIS), Hong Kong, China.*

[24] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. 2006.

[25] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[26] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical report.

[27] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and POWER8. In Deborah T. Marr and David H. Albonesi, editors, *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 144–157. ACM, 2015.

[28] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Notices*, volume 49, pages 317–328. ACM, 2014.

[29] Niloufar Shafiei. *Non-Blocking Data Structures Handling Multiple Changes Atomically*. PhD thesis, York University, 7 2015.

[30] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, page 204–213, New York, NY, USA, 1995. Association for Computing Machinery.

[31] Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *J. Parallel Distrib. Comput.*, 68(7):1008–1020, July 2008.

[32] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *25th Annual Symposium on Foundations of Computer Science*, pages 12–20, Oct 1984.

[33] Robert E. Tarjan. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. *Math. Programming*, 78:169–177, 1997.

[34] Shahar Timnat, Maurice Herlihy, and Erez Petrank. A practical transactional memory interface. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, pages 387–401, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[35] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[36] Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. Batch-parallel euler tour trees. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 92–106, 2019.

[37] John D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, 5 1995.