CONCURRENT BINARY SEARCH TREES:

DESIGN AND OPTIMIZATIONS

by

Arunmoezhi Ramachandran

APPROVED BY SUPERVISORY COMMITTEE:

Neeraj Mittal, Chair

Balaji Raghavachari

Venkatesan Subbarayan

Rob F. Van Der Wijngaart

Copyright © 2016

Arunmoezhi Ramachandran

All rights reserved

*To my parents and sister*

# CONCURRENT BINARY SEARCH TREES: DESIGN AND OPTIMIZATIONS

by

ARUNMOEZHI RAMACHANDRAN, BE, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May 2016

ACKNOWLEDGMENTS

I would like to thank my PhD advisor, Professor Neeraj Mittal, for guiding me throughout my PhD journey. I am also grateful to Dr. Rob Van Der Wijngaart who was my mentor during my internship with Intel. I enjoyed working with him on my first research project.

I am thankful to Professor R. Chandrasekaran who has always been kind and helpful to me. I also have to thank my PhD committee, Professors Balaji Raghavachari and Venkatesan Subbarayan, for their advice and suggestions in general.

December 2015

# PREFACE

This dissertation was produced in accordance with guidelines which permit the inclusion as part of the dissertation the text of an original paper or papers submitted for publication. The dissertation must still conform to all other requirements explained in the "Guide for the Preparation of Master's Theses and Doctoral Dissertations at The University of Texas at Dallas." It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts which provide logical bridges between different manuscripts are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the student's contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the dissertation attest to the accuracy of this statement.

CONCURRENT BINARY SEARCH TREES:

DESIGN AND OPTIMIZATIONS

Publication No. _____

Arunmoezhi Ramachandran, PhD
The University of Texas at Dallas, 2016

Supervising Professor: Neeraj Mittal

Over the last decade processor clock speeds have hit a wall but the demand for performance improvements has continued to grow. So there has been a major shift towards multi-core and many-core processors. This motivates the design of concurrent data structures. In such a data structure, multiple processes may need to operate on overlapping regions of the data structure simultaneously.

Designing a concurrent data structure is far more challenging than its sequential counterpart because processes executing concurrently may interleave in many ways. For all such interleavings, a concurrent data structure should manage the contention among the processes in such a way that all operations complete correctly and leave the data structure in a valid state. Concurrent algorithms may be *blocking* or *non-blocking*. Blocking algorithms are usually designed using locks. While a process is holding a lock, it blocks other processes from accessing the portion of the data structure protected by the lock. In a non-blocking algorithm, a suspended process will not prevent other processes from making progress. This is usually achieved by a concept called *helping*, where a process always leaves enough informa-

tion about its operation, so that even if it gets suspended, another process (which conflicts with the suspended process) can help finish the operation without waiting for the suspended process to resume.

In this work, we present a blocking and a non-blocking algorithm for concurrent manipulation of a binary search tree in an asynchronous shared memory system. Binary search trees are ubiquitous in computer science and are commonly used to implement dictionary abstract data type. We also provide a general optimization technique to improve performance of existing concurrent binary search trees. In this approach processes can recover from failures due to contention more efficiently using *local recovery*. Our approach is sufficiently general in the sense that it can be applied to a variety of concurrent binary search trees based on both blocking and non-blocking approaches. Moreover, we also present several techniques to make search operations on such binary search trees terminate in a finite number of steps. Our techniques ensures that search operations never have to restart due to a failure.

Experiments indicate that our algorithms perform best in most cases. And our optimization technique improves performance of our algorithms and other existing algorithms.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

With the growing prevalence of multi-core, multi-processor systems, concurrent data structures are becoming increasingly important. In such a data structure, multiple processes may need to operate on the data structure at the same time. Contention between different processes must be managed in such a way that all operations complete correctly and leave the data structure in a valid state.

Concurrency is often managed using locks. A lock can be used to achieve mutual exclusion, which can then be used to ensure that any updates to the data structure or a portion of it are performed by one process at a time. This makes it easier to design a lock-based concurrent data structure and reason about its correctness. Moreover, this also makes it easier to implement a lock-based data structure and debug it than its lock-free counterpart. Lock-based algorithms for concurrent versions of many important data structures for storing and managing shared data have been developed including linked lists, queues, priority queues, hash tables and skip lists (*e.g.*, [17, 18, 24, 25, 26, 28]).

However, locks are blocking; while a process is holding a lock, no other process can access the portion of the data structure protected by the lock. If a process stalls while it is holding a lock, then the lock may not be released for a long time. This may cause other processes to wait on the stalled process for extended periods of time. As a result, lock-based implementations of concurrent data structures are vulnerable to problems such as deadlock, priority inversion and convoying [18].

Non-blocking algorithms avoid the pitfalls of locks by using special (hardware-supported) *read-modify-write* instructions such as *load-link/store-conditional (LL/SC)* and

*compare-and-swap (CAS)* [18]. Non-blocking implementations of many common data structures such as queues, stacks, linked lists, hash tables and search trees have been proposed (*e.g.*, [3, 4, 10, 11, 12, 15, 18, 20, 26, 30, 31, 32]).

Binary search tree is one of the fundamental data structures for organizing *ordered* data that supports search, insert and delete operations [9]. It is commonly used to implement dictionary abstract data type. Many variants of binary search trees exists and they are used in different applications. For example, $B^+$tree [8] is used to implement indexes in databases. Quadtree [14], a two-dimensional version of binary search tree is used in image processing, geographical informational systems and in robotics. Similarly octtrees [21] are used to represent three dimensional objects and are used in 3D computer graphics.

A binary search tree may be unbalanced (different leaf nodes may be at very different depths) or balanced (all leaf nodes are at roughly the same depth). A balanced binary search tree provides better worst-case guarantees about the cost of performing an operation on the tree. However, in many cases, the overhead of keeping the tree balanced, especially in a concurrent environment, may incur significant overhead. As a result, in many cases, an unbalanced binary search tree outperforms a balanced binary search tree in practice. In this work, our focus is on developing efficient concurrent algorithms for an *unbalanced* binary search tree.

Concurrent algorithms for unbalanced binary search trees have been proposed in [2, 6, 10, 11, 12, 20, 31]. Algorithms in [2, 10] are blocking (or lock-based), whereas those in [6, 11, 12, 20, 31] are non-blocking (or lock-free). Also, algorithms in [2, 6, 10, 20] use internal representation of a search tree in which all nodes store data, where as those in [11, 12, 31] use an external representation of a search tree in which only leaf nodes store data (data stored in internal nodes is used for routing purposes only).

Algorithms that use internal representation of a search tree have to address the problem that arises due to a key moving from one location in the tree to another. This occurs when

the key undergoing deletion resides in a binary node, which requires it to be either replaced with its predecessor (next smallest key) or its successor (next largest key). As a result, an operation traversing the tree may fail to find the target key both at its old location and at its new location, even though the target key was continuously present in the tree. Different algorithms use different approaches to handle the problem arising due to key movement. The algorithm by Drachsler *et al.* in [10] maintains a *sorted* linked list of all the keys in the tree. If the traversal of the tree fails to find a given key, then an operation traverses the linked list to look for the key. The algorithm by Arbel and Hattiya [2] uses the RCU (Read-Copy-Update) framework (first employed in Linux kernels) to allow reads to occur concurrently with updates.

Most of the concurrent algorithms (for BSTs) that have proposed so far use a naïve approach and simply restart the traversal from the root of the tree [2, 10, 11, 20, 31]. This is especially undesirable if the tree has large height, and the overhead of repeatedly traversing the tree may dominate all other overheads of performing an operation.

Recently, a few algorithms have been proposed in which an operation attempts to recover from a failure *locally* [6, 12]. The algorithm in [12], which is based on external representation and builds upon the algorithm in [11], maintains a stack of the nodes visited during the traversal of the tree and simply restarts from the last "unmarked" node (a node is marked before it is removed from the tree). Intuitively, this works because, in an external search tree, keys *do not* move from one location in the tree to another. However, in a search tree based on internal representation, keys may move from one location (in the tree) to another. This occurs when the key undergoing deletion resides in a binary node, which requires it to be either replaced with its predecessor (next smallest key) or its successor (next largest key). This causes two problems. First, an operation traversing the tree may fail to find the target key both at its old location and at its new location, even though the target key was continuously present in the tree. Second, it is not always safe to simply restart from the

last unmarked node in the stack since the key may have moved to an ancestor of such a node. Their restart approach is, however, sufficiently general that it can be applied to other concurrent search trees based on external representation [31].

The algorithm in [6], which is based on internal representation, uses *backlink pointers* to find its way and recover from a failure while executing an operation. Their restart approach appears to be customized for their concurrent search tree and is not clear how it can be extended to other concurrent search trees based on internal representation.

### 1.1 Contributions

First we present a new *lock-based* algorithm for concurrent manipulation of a binary search tree in an asynchronous shared memory system that supports search, insert and delete operations. Our algorithm is based on an internal representation of a search tree as in [2, 10]. However, as in [31], it operates at edge-level (locks edges) rather than at node-level (locks nodes); this minimizes the contention window of a write operation and improves the system throughput. Further, in our algorithm, (i) a search operation uses only read and write instructions, (ii) an insert operation does not acquire any locks, and (iii) a delete operation only needs to lock up to four edges in the absence of contention. Our experiments indicate that our lock-based algorithm outperforms existing algorithms for a concurrent binary search tree—blocking as well as non-blocking—for medium-sized and larger trees, achieving up to 46% higher throughput than the next best algorithm.

Second we extend the previous algorithm to develop a new *lock-free* algorithm. It combines ideas from two existing lock-free algorithms, namely those by Howley and Jones [20] and Natarajan and Mittal [31], and is especially *optimized for the conflict-free scenario*. Like Howley and Jones' algorithm, it uses internal representation of a search tree in which all nodes store keys. Also, like Natarajan and Mittal's algorithm, it operates at edge-level rather than node-level and does not use a separate explicit object for enabling coordination

among conflicting operations. As a result, it inherits benefits of both the lock-free algorithms. Specifically, when compared to modify operations of Howley and Jones' internal binary search tree, its modify operations (a) have a smaller contention window, (b) allocate fewer objects, (c) execute fewer atomic instructions, and (d) have a smaller memory footprint. Our experiments indicate that our new lock-free algorithm outperforms other lock-free algorithms in most cases, providing up to 27% improvement in some cases over the next best algorithm.

Third, we present a general approach for local recovery that enables a process to quickly recover from a failure while performing an operation by restarting the traversal from a point "close" to the operation's window rather than the root of the tree. Our approach can be applied to many existing concurrent algorithms for maintaining binary search trees using internal representation—blocking as well as non-blocking—such as those in [2, 10, 20, 34]. Our local recovery approach uses only local variables and does not require modifying a tree node (of the original algorithm) to store any additional information. Using experimental evaluation, we demonstrate that our local recovery approach can yield significant speed-ups for many concurrent algorithms.

Finally, we present two light-weight techniques to make search operations for concurrent binary search trees based on internal representation, such as those in those in [2, 10, 20, 33, 34], *wait-free* with low additional overhead. Both of our techniques have the desirable feature that a search operation does not need to perform any write instructions on the share memory thereby minimizing the cache coherence traffic.

## 1.2 Dissertation Roadmap

This dissertation is organized as follows. We first describe the key concepts and techniques related to concurrent data structures in Chapter 2. We describe the system model in Chapter 3. Then we describe our lock-based algorithm for a binary search tree in Chapter 4

followed by our lock-free algorithm in Chapter 5. Our general technique for local recovery is described in Chapter 6. Then we discuss *wait-free* techniques for search operations in Chapter 7. The experimental evaluation of different concurrent algorithms for a binary search tree is described in Chapter 8. Finally Chapter 9 concludes the dissertation and outlines directions for future research.

# CHAPTER 2

# PRELIMINARIES

Processors were originally developed with only one core. So the data structures designed to run on then were sequential in nature. But as the processing speed of a single processor saturated, there was a shift towards multi-core processors. These shared-memory multiprocessors concurrently execute multiple threads. These threads communicate and synchronize through data structures in shared memory. The efficiency of these data structures are crucial to the performance. So developing concurrent versions of the sequential data structure became important. As threads in a multiprocessor can interleave in exponential number of ways, the challenge is to ensure that a concurrent data structure preserves its sequential specification for all possible interleavings.

## 2.1 Designing Concurrent Data Structures

The following example illustrates the design of a simple concurrent data structure. Let $x$ be a *shared counter* which can be incremented using a function fetchAndIncrement(). Figure 2.1 shows various implementations of this counter. In the sequential version, the *counter* $x$ is first loaded into a register, then the register value is incremented and finally it is stored back into $x$. Each of these three steps are atomic. But put together they are no longer atomic. For instance consider this scenario. Let the initial value of $x$ be 0. Two threads $\alpha$ and $\beta$ read its value and copied it into their local registers. Then, both of them increment their local registers. Now when they try to write their register value back into the shared counter, there is a race condition. Without loss of generality let us assume that thread $\alpha$ made the first write. Now $x$ becomes 1. Then when $\beta$ issues a write, the value of $x$ is overwritten.

But the new value of $x$ is still 1. Ideally, it has to be 2 as two threads have incremented the shared counter. This is a classic *lost-update* or *write-after-write* problem.

```
r1 = x;
inc(r1);
x = r1;
```

**fetchAndIncrement:** sequential

```
acquire(lock);
r1 = x;
inc(r1);
x = r1;
release(lock);
```

**fetchAndIncrement:** Using locks

```
repeat
    rOld = x;
    rNew = rOld+1;
until (x.compareAndSwap(rOld,rNew));
```

**fetchAndIncrement:** using atomic instructions

Figure 2.1. Various implementations of a Shared Counter

A simple way to avoid such race conditions is to wrap them with locks. Consider the same scenario again. Without loss of generality, let thread $\alpha$ obtain the lock first. Thread $\beta$ cannot do any operation on the shared counter until thread $\alpha$ releases the lock. When $\alpha$ releases the lock, the value of $x$ is 1. Now when $\beta$ obtains the lock it reads the value of $x$ to be 1. It increments the value of $x$ and releases the lock. The final value of $x$ is 2 which is as expected.

Another way to solve the race conditions is to use *Read-Modify-Write* (RMW) instructions like *test-and-set*, *fetch-and-add* and *compare-and-swap*. They essentially read a memory location and write a new value into it atomically. As shown in Figure 2.1, A compare-and-swap instruction takes three arguments: *address*, *old* and *new*; it compares the contents of a memory location (*address*) to a given value (*old*) and, only if they are the same, modifies the contents of that location to a given new value (*new*). The same shared counter can be implemented using a compare-and-swap instruction. For the same scenario, let both $\alpha$ and

$\beta$ threads read the value of $x$ into their local register $rOld$. They compute $rNew$ to be 1 and both of them try to perform a CAS(Compare-And-Swap) operation. Without loss of generality let $\alpha$ precede $\beta$. The operation by $\alpha$ would succeed as the the value of $x(0)$ is equal to $rOld(0)$. But when $\beta$ tries to perform a CAS operation, the value of $x$ has already changed to 1. So it would fail, as the value of $x(1)$ is not equal to $rOld(0)$. So $\beta$ would retry the operation. This time it reads the updated value of $x$ into $rOld$ and the CAS operation would succeed leaving the final value of $x$ to be 2.

Each of the two solutions described above handled contention in a different way. The one using locks are used in developing blocking algorithms and the one using atomic operations are used to develop non-blocking algorithms.

## 2.2 Blocking algorithms

The granularity of the locks used in blocking algorithms can be *coarse* or *fine*. Coarse grained locks allows only one thread to operate on a data structure. But fine grain locks allows multiple threads to operate on different portions of the data structure concurrently. In general, fine-grained locking approaches provides better performance. Though blocking algorithm are relatively easier to design than their non-blocking counterpart, they have their own limitations. They are often prone to deadlocks and priority inversion. Also they provide weaker progress guarantees. For instance consider two threads $\alpha$ and $\beta$ trying to obtain lock on a shared counter. Let $\alpha$ obtain the lock. Assume that before it increments the counter it gets swapped out of context by the operating system. Now $\beta$ will not be able to obtain the lock until $\alpha$ releases it. Though there is no deadlock in this state, the system as a whole is not making any progress.

## 2.3 Non-blocking algorithms

Non-blocking algorithms uses atomic (Read-Modify-Write) instructions to resolve contention. The also provide stronger progress guarantees. A non-blocking algorithm is either *lock-free* or *wait-free*. An algorithm is lock-free if at least one thread is able to make progress over an infinite period of time in a system. It is wait-free if every thread is able to complete its operations in a finite number of steps over an infinite period of time in a system. The notion of lock-freedom is at the algorithm level and not at the system level as the atomic operations still uses locks at the hardware level. In a blocking algorithm a thread owns a lock but in a non-blocking algorithm an operation being performed by a thread owns a lock.

Lock-freedom and wait-freedom are usually achieved using the concept of *helping*. When a thread performing its operation sees that some other thread is in its way, then it first *helps* the other thread before continuing its own operation. To enable helping, every thread, for all its operations on the shared data structure, leaves enough information in the shared memory so that even if it gets swapped out of context by the operating system, some other contending thread might still be able to help finish its operation. This way the system as a whole is able to make progress even though some threads might be stalled in the middle of its operations.

## 2.4 Linearizability

Proving correctness of concurrent data structures are as hard as designing them. For the simple shared counter using locks and atomic operations it is easy to prove the correctness. But for complex data structures like trees it is much more difficult. So we need a formal way to define correctness.

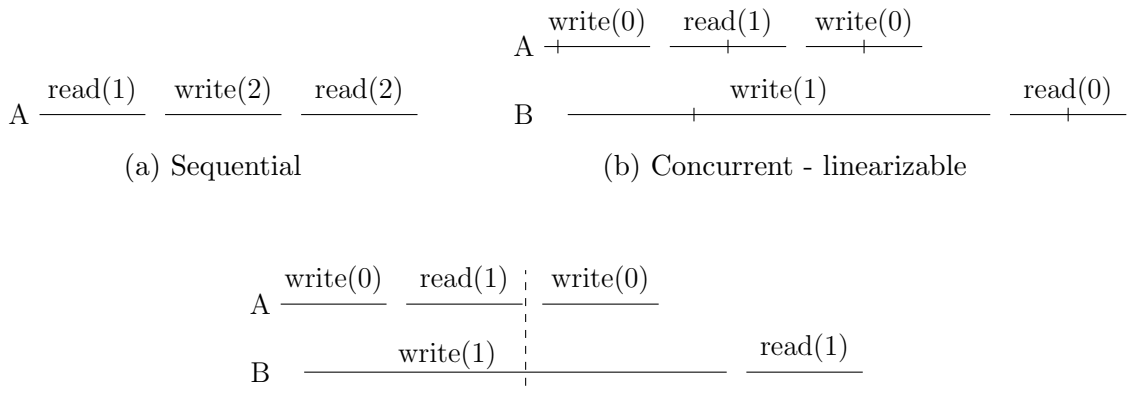
A sequential object has a state and a set of methods which operate on the object making the object move from one valid state to another. A sequence of method invocations and

responses is called a *history*. Every method has a set of pre-conditions and post-conditions. Pre-conditions captures the state of the object before the method is invoked and post-conditions captures the state after the method returns. In a sequential specification of an object, each method is described in isolation. The interactions among methods are captured by the side-effects on the object state. So a sequential object needs a meaningful state only between method calls. Also as methods are described in isolation, new methods can be added without modifying the existing methods. As the methods are executed one-at-a-time, proving the correctness requires that the any history (a sequence of method invocations and response) respects the sequential specification of the object.

For a concurrent object, multiple methods might be executing concurrently. Since method calls overlap, an object might never be between method calls and hence we cannot define a valid state. Also we must consider all the exponential possible interactions among method calls. Proving the correctness of a concurrent object requires that some total ordering of any history respects the sequential specification of the object.

Two well known consistency conditions for shared memory systems are *sequential consistency* [23] and *linearizability* [19]. Sequential consistency is not composable while linearizability is. Traditionally software systems as they are built by composing multiple subsystems. So linearizability is preferred in designing concurrent data structures.

Linearizability requires two properties: (i) the object (or data structure) be sequentially consistent (ii) the total ordering which makes it sequentially consistent respect the *real-time ordering* among the operations in the execution. Respecting real-time ordering means that if an operation $op_1$ completed before another operation $op_2$, then $op_1$ must be ordered before $op_2$. In other words, linearizability requires us to identify a distinct point between a method invocation and response where the method appears to have taken effect instantaneously. This point is called a *linearization point*. Now if we order the method calls based on their linearization points then the resulting order should be in the sequential specification of the object.




(a) Sequential

(b) Concurrent - linearizable

(c) Concurrent - not linearizable

Figure 2.2. A sample history of an object

Figure 2.2(a) shows a sequential execution of reads and writes on a register. Figure 2.2(b) shows an execution history of a concurrent object. The linearization points are chosen such that *write(0)* $\rightarrow$ *write(1)* $\rightarrow$ *read(1)* $\rightarrow$ *write(0)* $\rightarrow$ *read(0)*. Figure 2.2(c) shows a history which is not linearizable. Since thread $A$ reads a value 1, the *write()* of thread $B$ must precede the second *write()* of thread $A$. So the *read()* by thread $B$ cannot read a value 1.

As linearizability is intuitive and composable, most of the concurrent data structure implementations use it as a correctness condition. Moir and Shavit [29] provides a detailed literature survey of various concurrent data structure implementations. For each of the algorithms we present in this dissertation we prove that they are linearizable.

# CHAPTER 3

# SYSTEM MODEL

We assume an asynchronous shared memory system that, in addition to read and write instructions, also supports compare-and-swap (CAS) atomic instruction. The CAS instruction is commonly available in many modern processors such as Intel 64 and AMD64.

We also use locks and assume that the following properties hold true about the locks

(a) safe: it satisfies the mutual exclusion property, *i.e.*, at most one process can hold the lock at any time, and

(b) live: it satisfies the deadlock freedom property, *i.e.*, if the lock is free and one or more processes attempt to acquire the lock, then some process is eventually able to acquire the lock.

## 3.1 Binary Search Tree

We assume that a binary search tree (BST) implements a dictionary abstract data type and supports *search*, *insert* and *delete* operations. For convenience, we refer to the insert and delete operations as *modify* operations. A search operation explores the tree for a given key and returns true if the key is present in the tree and false otherwise. An insert operation adds a given key to the tree if the key is not already present in the tree. Duplicate keys are not allowed in our model. A delete operation removes a key from the tree if the key is indeed present in the tree. In both cases, a modify operation returns true if it changed the set of keys present in the tree (added or removed a key) and false otherwise.

A binary search tree satisfies the following properties:

(a) the left subtree of a node contains only nodes with keys less than the node's key,

(b) the right subtree of a node contains only nodes with keys greater than or equal to the node's key, and

(c) the left and right subtrees of a node are also binary search trees.

### 3.2 Correctness conditions

To demonstrate the correctness of our algorithms, we use *linearizability* [19] for the safety property. Broadly speaking, linearizability requires that an operation should appear to take effect instantaneously at some point during its execution. For our lock-based algorithm we use *deadlock-freedom* [18] for the liveness property. Deadlock-freedom requires that some process with a pending operation be able to complete its operation eventually. And for our lock-free algorithm we use *lock-freedom* [18] for progress guarantees. Lock freedom ensures over an infinite period of time in a system, at least one thread is able to make progress.

# PART I

# DESIGN

# CHAPTER 4

# LOCK BASED CONCURRENT BINARY SEARCH TREE

We first provide an overview of our algorithm. We then describe the algorithm in more detail and also give its pseudo-code. For ease of exposition, we describe our algorithm assuming no memory reclamation, which can be performed using the well-known technique of hazard pointers [27].

## 4.1 Overview of the Algorithm

Every operation in our algorithm uses *seek* function as a subroutine. The seek function traverses the tree from the root node until it either finds the target key or reaches a non-binary node whose next edge to be followed points to a null node. We refer to the path traversed by the operation during the seek as the *access-path*, and the last node in the access-path as the *terminal node*. The operation then compares the target key with the stored key (the key present in the terminal node). Depending on the result of the comparison and the type of the operation, the operation either terminates or moves to the execution phase. In certain cases in which a key may have moved upward along the access-path, the seek function may have to restart and traverse the tree again; details about restarting are provided later. We now describe the next steps for each of the type of operation one-by-one.

**Search:** A search operation starts by invoking seek operation. It returns true if the stored key matches the target key and false otherwise.

**Insert:** An insert operation starts by invoking seek operation. It returns false if the target key matches the stored key; otherwise, it moves to the execution phase. In the execution

phase, it attempts to insert the key into the tree as a child node of the last node in the access-path using a CAS instruction. If the instruction succeeds, then the operation returns true; otherwise, it restarts by invoking the seek function again.

**Delete:** A delete operation starts by invoking seek function. It returns false if the stored key does not match the target key; otherwise, it moves to the execution phase. In the execution phase, it attempts to remove the key stored in the terminal node of the access-path. There are two cases depending on whether the terminal node is a binary node (has two children) or not (has at most one child). In the first case, the operation is referred to as *complex delete operation*. In the second case, it is referred to as *simple delete operation*. In the case of simple delete, the terminal node is removed by changing the pointer at the parent node of the terminal node. In the case of complex delete, the key to be deleted is replaced with the *next largest* key in the tree, which will be stored in the *leftmost node* of the *right subtree* of the terminal node.

### 4.2 Details of the Algorithm

As in most algorithms, to make it easier to handle special cases, we use sentinel keys and sentinel nodes. The structure of an empty tree with only sentinel keys (denoted by $\infty_1$ and $\infty_2$ with $\infty_1 < \infty_2$) and sentinel nodes (denoted by $\mathbb{R}$ and $\mathbb{S}$) is shown in Figure 4.1.

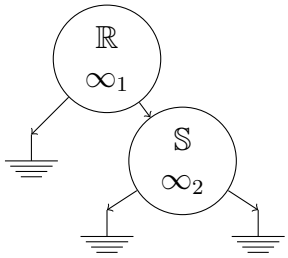



Figure 4.1. Sentinel keys and nodes ($\infty_1 < \infty_2$)

Our algorithm, like the one in [31], operates at edge level. A delete operation obtains ownership of the edges it needs to work on by locking them. To enable locking of an edge, we

steal a bit from the child addresses of a node referred to as *lock-flag*. We also steal another bit from the child addresses of a node to indicate that the node is undergoing deletion and will be removed from the tree. We denote this bit by *mark-flag*. Finally, to avoid the ABA problem, as in Howley and Jones [20], we use *unique* null pointers. To that end, we steal yet another bit from the child address, referred to as *null-flag*, and use it to indicate whether the address points to a null or a non-null address. So, when an address changes from a non-null value to a null value, we only set the null-flag and the contents of the address are not otherwise modified. This ensures that all null pointers are unique.

We next describe the details of the seek operation, which is executed by all operations (search as well as modify) after which we describe the details of the execution phase of insert and delete operations.

### 4.2.1 The Seek Phase

A seek function keeps track of the node in the access-path at which it took the last "right turn" (*i.e.*, it last followed a right edge). Let this "right turn" node be referred to as *anchor node* when the traversal reaches the terminal node. Note that the terminal node is the node whose key matched the target key or whose next child edge is set to a null address. For an illustration, please see Figure 4.2. In the latter case (stored key does not match the target key), it is possible that the key may have moved up in the tree. To ascertain that the seek function did not miss the key because it may have moved up during the traversal, we use the following set of conditions that are *sufficient* (but not necessary) to guarantee that the seek function did not miss the key. First, the anchor node is still part of the tree. (For an illustration, see Figure 4.3) Second, the key stored in the anchor node has not changed since it first encountered the anchor node during the (current) traversal. To check for the above two conditions, we determine whether the anchor node is undergoing deletion by examining it right child edge. We discuss the two cases one-by-one.

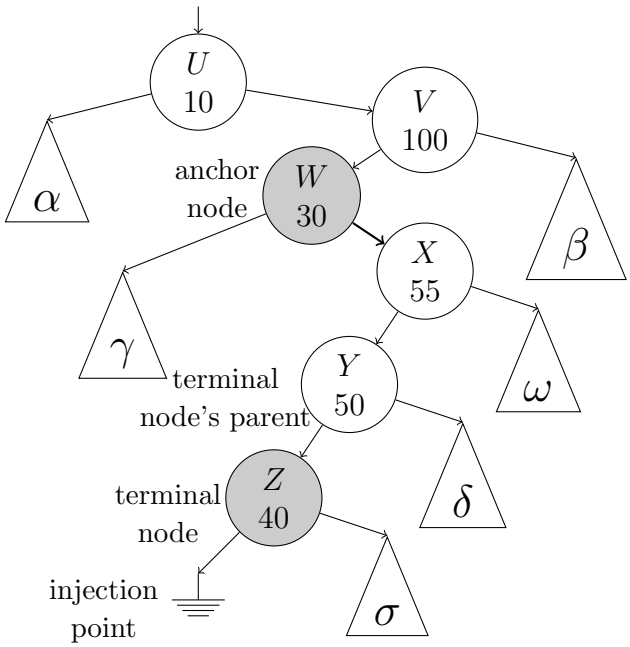



Figure 4.2. Nodes in the access path of seek

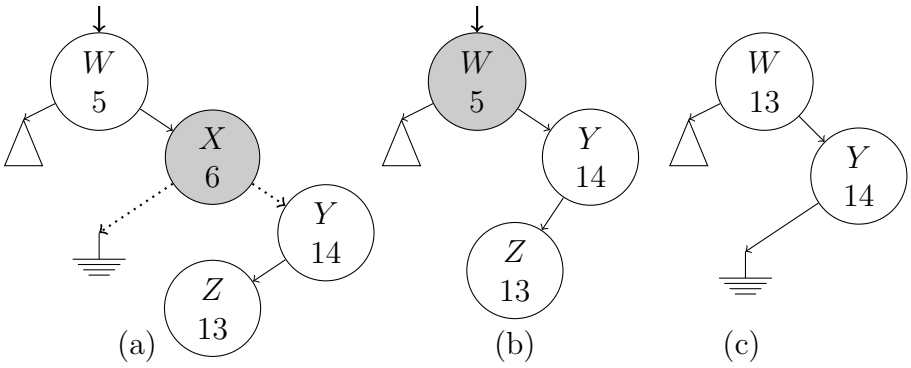



Figure 4.3. A case when last right turn node is no longer part of the tree

(a) *Right child edge not marked:* In this case, the anchor node is still part of the tree. We next check whether the key stored in the anchor node has changed. If the key has not changed, then the seek function returns the results of the (current) traversal, which consists of three addresses: (i) the address of the terminal node, (ii) the address of its parent, and (iii) the null address stored in the child field of the terminal node that caused the traversal to terminate. The last address is required to ensure that an insert operation works correctly (specifically to ascertain that the child field of the terminal node has not undergone any change since the completion of the traversal). We refer to it as the *injection point* of the insert operation. On the other hand, if the key has changed, then the seek function restarts from the root of the tree. A possible optimization is that the seek function restarts only if the target key is now less than the anchor node's key.

(b) *Right child edge marked:* In this case, we compare the information gathered in the current traversal about the anchor node with that in the previous traversal, if one exists. Specifically, if the anchor node of the previous traversal is same as that of the current traversal and the keys found in the anchor node in the two traversals also match, then the seek function terminates, but returns the results of the previous traversal (instead of that of the current traversal). This is because the anchor node was definitely part of the tree during the previous traversal since it was reachable from the root of the tree at the beginning of the current traversal. Otherwise, the seek function restarts from the root of the tree.

For insert and delete operations, we refer to the terminal node as the *target node*.

### 4.2.2 The Execution Phase of an Insert Operation

In the execution phase, an insert operation creates a new node containing the target key. It then adds the new node to the tree at the injection point using a CAS instruction. For an

illustration, see Figure 4.4. If the CAS instruction succeeds, then (the new node becomes a part of the tree and) the operation terminates; otherwise, the operation restarts from the seek phase. Note that the insert operations are lock-free.

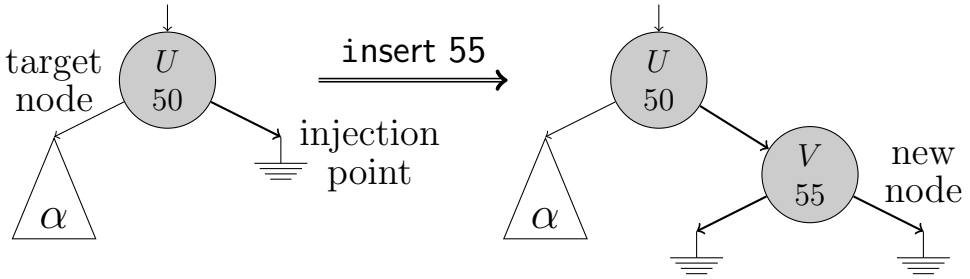



Figure 4.4. An illustration of an insert operation

### 4.2.3 The Execution Phase of a Delete Operation

The execution of a delete operation starts by checking if the target node is a binary node or not. If it is a binary node, then the delete operation is classified as complex; otherwise it is classified as simple.

For a tree node $X$, let $X.parent$ denote its parent node, and $X.left$ and $X.right$ denote its left and right child node, respectively. Also, hereafter in this section, let $T$ denote the target node of the delete operation under consideration.

(a) *Simple Delete:* In this case, either $T.left$ or $T.right$ is pointing to a null node. Note that both $T.left$ and $T.right$ may be pointing to null nodes in which case $T$ will be a leaf node. Without loss of generality, assume that $T.right$ is a null node. The removal of $T$ involves locking the following three edges: $\langle T.parent, T\rangle$, $\langle T, T.left\rangle$ and $\langle T, T.right\rangle$. For an illustration, see Figure 4.5.

A lock on an edge is obtained by setting the lock-flag in the appropriate child field of the parent node using a CAS instruction. For example, to lock the edge $\langle X, Y\rangle$, where $Y$ is the left child of $X$, the lock-flag in the left child of $X$ is set to one. If all the

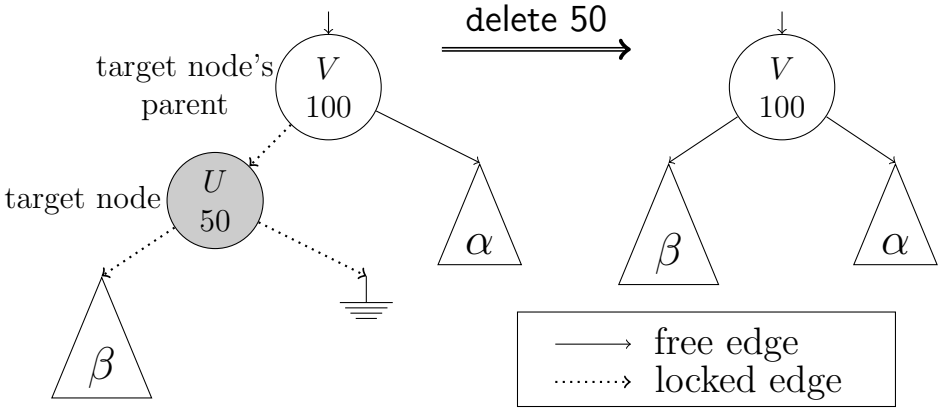



Figure 4.5. An illustration of a simple delete operation

edges are locked successfully, then the operation validates that the key stored in the target node still matches the target key. If the validation succeeds, then the operation marks both the children edges of $T$ to indicate that $T$ is going to be removed from the tree. Next, it changes the child pointer at $T.parent$ that is pointing to $T$ to point to $T.left$ using a simple write instruction. Finally, the operation releases all the locks and returns true.

(b) *Complex Delete:* In this case, both $T.left$ and $T.right$ are pointing to non-null nodes. The operation locates the next largest key in the tree, which is the smallest key in the subtree rooted at the right child of $T$. We refer to this key as the *successor key* and the node storing this key as the *successor node*. Hereafter in this section, let $S$ denote the successor node. Deletion of the key stored in $T$ involves copying the key stored in $S$ to $T$ and then removing $S$ from the tree. To that end, the following edges are locked by setting the lock-flag on the edge using a CAS instruction: $\langle T, T.right\rangle$, $\langle S.parent, S\rangle$, $\langle S, S.left\rangle$ and $\langle S, S.right\rangle$. For an illustration, see Figure 4.6. Note that the first two edges may be same which happens if the successor node is the right child of the target node. Also, since we do not lock the left edge of the target node, the left edge may change and may possibly start pointing to a null address. But, that does not impact the correctness of the complex delete operation.

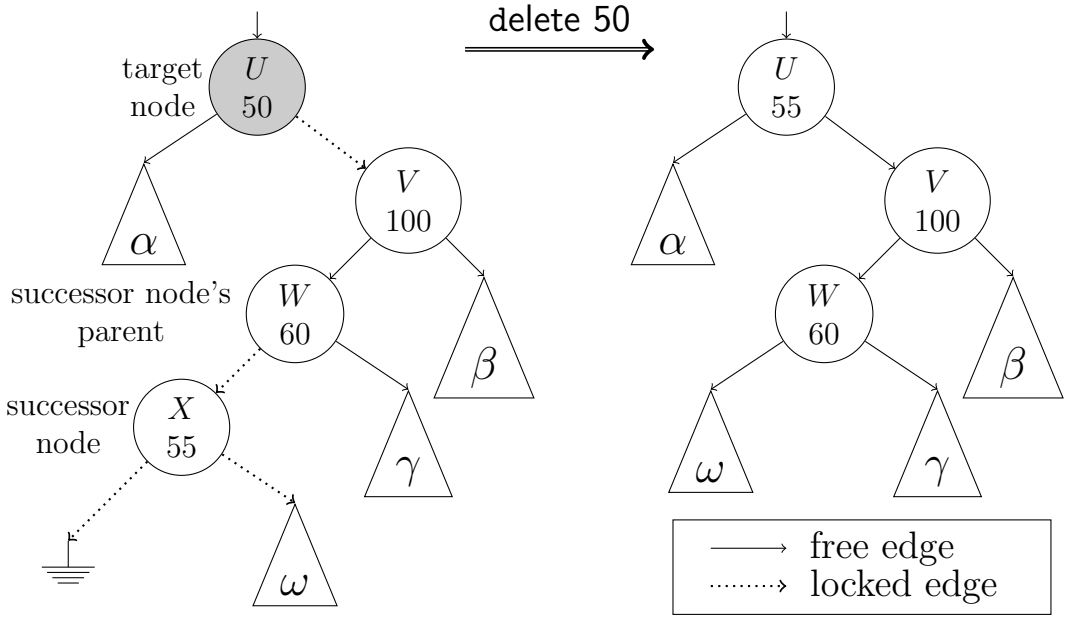



Figure 4.6. An illustration of a complex delete operation

If all the edges are locked successfully, then the operation validates that the key stored in the target node still matches the target key. If the validation succeeds, then the operation copies the key stored in $S$ to $T$, and marks both the children edges of $S$ to indicate that $S$ is going to be removed from the tree. Next, it changes the child pointer at $S.parent$ that is pointing to $S$ to point to $S.right$ using a simple write instruction. Finally, the operation releases all the locks and returns true.

In both cases (simple as well as complex delete), if the operation fails to obtain any of the locks, then it releases all the locks it was able to acquire up to that point, and restarts from the seek phase. Also, after obtaining all the locks, if the key validation fails, then it implies that some other delete operation has removed the key from the tree while the current execution phase was in progress. In that case, the given delete operation releases all the locks, and simply returns false. Note that using a CAS instruction for setting the lock-flag also enables us to *validate that the child pointer has not changed* since it was last observed in a single step.

### 4.3 Formal Description

We refer to our algorithm as CASTLE (Concurrent Algorithm for Binary Search Tree by Locking Edges).

**Algorithm 4:** Data Structures Used

```
   // a tree node
13 struct Node {
14     Key key;
15     { boolean, boolean, boolean, NodePtr } child[2];
       // each child field contains four subfields:  lFlag, mFlag, nFlag and address
16 };
   // used to store the results of a tree traversal
17 struct SeekRecord {
18     NodePtr node;
19     NodePtr parent;
20     NodePtr nullAddress;
21 };
   // used to store information about an anchor node
22 struct AnchorRecord {
23     NodePtr node;
24     Key key;
25 };
   // used to store information about an edge to lock
26 struct LockRecord {
27     NodePtr node;
28     enum { LEFT, RIGHT } which;
29     { boolean, NodePtr } addressSeen;
       // addressSeen contains two subfields:  nFlag and address
30 };
   // local seek record used when looking for a node
31 SeekRecordPtr seekTargetKey, seekSuccessorKey;
   // local array used to store the set of edges to lock
32 LockRecord lockArray[4];
```

A pseudo-code of our algorithm is given in Algorithms 4-10. Different data structures used in our algorithm are shown in Algorithm 4. Besides tree node, we use three additional records: (a) *seek record:* to store the outcome of a tree traversal both when looking for the target key and the successor key, (b) *anchor record:* to store information about the anchor node during the seek phase, and (c) *lock record:* to store information about a tree edge that needs to be locked.

**Algorithm 5:** Seek Function

```
33 SEEK( key, seekRecord )
34 begin
35     while true do
           // initialize the variables used in the traversal
36         pNode := ℝ;      cNode := 𝕊;
37         address := 𝕊→child[RIGHT].address;
38         anchorRecord := {ℝ, ∞1};
39         while true do
               // reached terminal node; read the key stored in the current node
40             cKey := cNode→key;
41             if key = cKey then
42                 seekRecord := {cNode, pNode, address};
43                 return;
44             which := key < cKey ? LEFT : RIGHT;
               // read the next address to dereference along with mark and null flags
45             ⟨∗, ∗, nFlag, address⟩ := cNode→child[which];
46             if nFlag then                    // the null flag is set; reached terminal node
47                 aNode := anchorRecord→node;
48                 if aNode→child[RIGHT].mFlag then
                       // the anchor node is marked; it may no longer be part of the tree
49                     if anchorRecord = pAnchorRecord then
                           // the anchor record of the current traversal matches that of
                              the previous traversal
50                         seekRecord := pSeekRecord;
51                         return;
52                     else break;
53                 else                    // the anchor node is definitely part of the tree
54                     if aNode→key < key then                    // seek can terminate now
55                         seekRecord := {cNode, pNode, address};
56                         return;
57                     else break;

               // update the anchor record if needed
58             if which = RIGHT then
                   // the next edge to be traversed is a right edge; keep track of
                      current node and its key
59                 anchorRecord := {cNode, cKey};

               // traverse the next edge
60             pNode := cNode;      cNode : =address;
```

**Algorithm 6:** Search Operation

```
61 boolean SEARCH( key )
62 begin
63     SEEK( key, seekTargetKey );
64     node := seekTargetKey→node;
65     if node→key = key then
66         return true;                                          // key found
67     else
68         return false;                                         // key not found
```

**Algorithm 7:** Insert Operation

```
69 boolean INSERT( key )
70 begin
71     while true do
72         SEEK( key, seekTargetKey );
73         node := seekTargetKey→node;
74         if node→key = key then
75             return false;                                     // key found
76         else
               // key not found; add the key to the tree
77             newNode := create a new node;
               // initialize its fields
78             newNode→key := key;
79             newNode→child[LEFT] := ⟨0_l, 0_m, 1_n, null⟩;
80             newNode→child[RIGHT] := ⟨0_l, 0_m, 1_n, null⟩;
               // determine which child field (left or right) needs to be modified
81             which := key < node→key ? LEFT : RIGHT;
               // fetch the address observed by the seek function in that field
82             address := seekTargetKey→nullAddress;
83             result := CAS( node→child[which],
                              ⟨0_l, 0_m, 1_n, address⟩,
                              ⟨0_l, 0_m, 0_n, newNode⟩ );
84             if result then
                   // new key successfully added to the tree
85                 return true;
```

**Algorithm 8:** Delete Operation

```
86  boolean DELETE( key )
87  begin
88      while true do
89          SEEK( key, seekTargetKey );
90          node := seekTargetKey→node;
91          if node→key ≠ key then                                   // key not found
92              return false;
93          else                  // key found; read contents of target node's children fields
94              lField := CLEARFLAGS( node→child[LEFT] );
95              rField := CLEARFLAGS( node→child[RIGHT] );
96              if lField.nFlag  or  rField.nFlag then              // simple delete operation
97                  parent := seekTargetKey→parent;
98                  if key < parent→key then  which := LEFT;
99                  else  which := RIGHT;
100                 lockArray[0] := {parent, which, ⟨0, node⟩};
101                 lockArray[1] := {node, LEFT, lField};
102                 lockArray[2] := {node, RIGHT, rField};
103                 result := LOCKALL( lockArray, 3 );
104                 if result then              // all locks acquired; perform the operation
105                     if node→key = key then      // key still matches; remove the node
106                         REMOVECHILD( parent, which ); match := true;
107                     else  match := false;
108                     UNLOCKALL( lockArray, 3 );
109                     return match;
110             else                  // complex delete operation; locate the successor node
111                 FINDSMALLEST(node, rField.address, seekSuccessorKey);
112                 sNode := seekSuccessorKey→node; sParent := seekSuccessorKey→parent;
                    // determine the edges to be locked
113                 lockArray[0] := {node, RIGHT, rField};
114                 if node ≠ sParent then
                        // successor node is not the right child of target node
115                     lockArray[1] := {sParent, LEFT, ⟨0, sNode⟩}; size := 4;
116                 else  size := 3 ;
117                 lField := CLEARFLAGS( sNode→child[LEFT] );
118                 rField := CLEARFLAGS( sNode→child[RIGHT] );
119                 lockArray[size − 2] := {sNode, LEFT, lField};
120                 lockArray[size − 1] := {sNode, RIGHT, rField};
121                 result := LOCKALL( lockArray, size );
122                 if result then              // all locks acquired; perform the operation
123                     if node→key = key then
                            // key still matches; copy key in successor node to target node
124                         node→key := sNode→key;
125                         REMOVECHILD( sParent, LEFT ); match := true;
126                     else  match := false;
127                     UNLOCKALL( lockArray, size );
128                     return match;
```

**Algorithm 9:** Lock and Unlock Functions

```
129 boolean LockAll( lockArray, size )
130 begin
131     for i ← 0 to size − 1 do
            // acquire lock for the i-th entry
132         node := lockArray[i].node;
133         which := lockArray[i].which;
134         lockedAddress := lockArray[i].addressSeen;
135         lockedAddress.lFlag := true;
            // set the lock flag in the child edge
136         result := CAS(node→child[which], lockArray[i].addressSeen, lockedAddress);
137         if not (result) then
                // release all the locks acquired so far
138             UnlockAll( lockArray, i − 1 );
139             return false;
140     return true;
141 UnlockAll( lockArray, size )
142 begin
143     for i ← size − 1 to 0 do
144         node := lockArray[i].node;
145         which := lockArray[i].which;
            // clear the lock flag in the child edge
146         node→child[which].lFlag := false;
```

The pseudo-code for the seek function is shown in Algorithm 5. The pseudo-codes for search, insert and delete operations are shown in Algorithm 6, Algorithm 7 and Algorithm 8, respectively. Algorithm 9 contains the pseudo-code for locking and unlocking a set of tree edges, as specified in an array. Finally, Algorithm 10 contains the pseudo-codes for three helper functions used by a delete operation, namely: (a) ClearFlags: to clear lock and mark flags from a child field, (b) FindSmallest: to locate the smallest key in a subtree, and (c) RemoveChild: to remove a given child of a node.

In the pseudo-code, to improve clarity, we sometimes use subscripts $l$, $m$ and $n$ to denote lock, mark and null flags, respectively.

**Algorithm 10:** Helper Functions used by Delete Operation

```
147 word ClearFlags( word field )
148 begin
149     newField := field with lock and mark flags cleared;
150     return newField;

151 FindSmallest( parent, node, seekRecord )
152 begin
        // initialize the variables used in the traversal
153     pNode := parent;      cNode := node;
154     while true do
155         ⟨∗, ∗, nFlag, address⟩ := cNode→child[LEFT];
156         if not (nFlag) then
                // visit the next node
157             pNode := cNode;      cNode := address;
158         else
                // reached the successor node
159             seekRecord := {cNode, pNode, address};
160             break;

161 RemoveChild( parent, which )
162 begin
        // determine the address of the child to be removed
163     node := parent→child[which];
        // mark both the children edges of the node to be removed
164     node→child[LEFT].mFlag := true;
165     node→child[RIGHT].mFlag := true;
        // determine whether both the child pointers of the node to be removed are null
166     if node→child[LEFT].nFlag  and  node→child[RIGHT].nFlag then
            // set the null flag only
167         parent→child[which].nFlag := true;
168     else
            // switch the pointer at the parent to point to its appropriate grandchild
169         if node→child[LEFT].nFlag then
170             address := node→child[RIGHT].address;
171         else  address := node→child[LEFT].address ;
172         parent→child[which].address := address;
```

## 4.4 Correctness Proof

It is convenient to treat insert and delete operations that do not change the tree as search operations. We call a tree node *active* if it is reachable from the root of the tree. We call a tree node *passive* if it was active earlier but is not active any more. Note that, before an active node is made passive by a delete operation, both its children edges are *marked*. Also, a CAS instruction performed on an edge (by either an insert operation or a delete operation as part of locking) is successful only if the edge is unmarked. As a result, clearly, if an insert operation completes successfully, then its target node was active when its edge was modified to make the new node (containing the target key) a part of the tree. Likewise, if a delete operation completes successfully, then all the nodes involved in the operation (up to three nodes) were active when their edges were locked.

### All Executions are Linearizable

We show that an arbitrary execution of our algorithm is linearizable by specifying the *linearization point* of each operation. Note that the linearization point of an operation is the point during its execution at which the operation appeared to have taken effect. Our algorithm supports three types of operations: search, insert and delete. We now specify the linearization point of each operation.

1. *Insert operation:* The operation is linearized at the point at which it performed the successful CAS instruction that resulted in its target key becoming part of the tree.

2. *Delete operation:* There are two cases depending on whether the delete operation is simple or complex. If the operation is simple delete, then the operation is linearized at the point at which a successful write step was performed at the parent of the target node that resulted in the target node becoming passive. Otherwise, it is linearized at the point at which the original key of the target node was replaced with its successor key.

3. *Search operation:* There are two cases depending on whether the target node was active when the operation read the key stored in the node. If the target node was not active, then the operation is linearized at the point at which the target node became passive. Otherwise, it is linearized at the point at which the read step was performed.

It can be easily verified that, for any execution of the algorithm, the sequence of operations obtained by ordering operations based on their linearization points is legal, *i.e.*, all operations in the sequence satisfy their specification.

Thus we have:

**Theorem 1.** *Every execution of our algorithm is linearizable.*

**All Executions are Deadlock-Free**

We say that the system is in a *quiescent state* if no modify operation completes hereafter. We say that the system is in a *potent state* if it has one or more pending modify operations. Note that quiescence is a *stable property*; once the system is in a quiescent state, it stays in a quiescent state. We show that our algorithm is deadlock-free by proving that a potent state is necessarily non-quiescent.

Note that, in a quiescent state, no edges in the tree can be marked. This is because a delete operation marks edges only after it has successfully obtained all the locks, after which it is guaranteed to complete. This also implies that the tree cannot undergo any changes now because that would imply eventual completion of a modify operation. Thus, once a system has reached a quiescent state, all modify operation currently pending repeatedly alternate between seek and execution phases. We say that the system is in a *strongly-quiescent state* if all pending modify operations started their most recent seek phase *after* the system became quiescent. Note that, like quiescence, strong quiescence is also a stable property. Now, once the system has reached a strongly quiescent state, the following can be easily verified. First,

for a given modify operation, every traversal of the tree in the seek phase returns the same target node. Second, for a given delete operation, the set of edges it needs to lock remains the same.

Now, assume that the system eventually reaches a state that is both potent and quiescent. Clearly, from this state, the system will eventually reach a state that is potent and strongly-quiescent. Note that a delete operation in our algorithm locks edges in a *top-down*, *left-right* manner. As a result, there cannot be a "cycle" involving delete operations. If a delete operation continues to fail in the execution phase, then it is necessarily because it tried to acquire lock on an already locked edge. (Recall that the set of edges does not change any more and there are no marked edges in the tree.) We can construct a chain of operations such that each operation in the chain tried to lock an edge already locked by the next operation in the chain. Clearly, the length of the chain is bounded. This implies that the last operation in the chain is guaranteed to obtain all the locks and will eventually complete. This contradicts the fact that the system is in a quiescent state.

Thus, we have:

**Theorem 2.** *Every execution of our algorithm is deadlock-free.*

# CHAPTER 5

# LOCK FREE CONCURRENT BINARY SEARCH TREE

In this chapter we present our lock-free algorithm for a binary search tree. We first provide an overview of our algorithm. And then we describe the algorithm in more detail and also give its pseudo-code. For ease of exposition, we describe our algorithm assuming no memory reclamation.

## 5.1 Overview of the Algorithm

Every operation in our algorithm uses *seek* function as a subroutine. The seek function traverses the tree from the root node until it either finds the target key or reaches a non-binary node whose next edge to be followed points to a null node. We refer to the path traversed by the operation during the seek as the *access-path*, and the last node in the access-path as the *terminal node*. The operation then compares the target key with the stored key (the key present in the terminal node). Depending on the result of the comparison and the type of the operation, the operation either terminates or moves to the execution phase. In certain cases in which a key may have moved upward along the access-path, the seek function may have to restart and traverse the tree again; details about restarting are provided later. We now describe the next steps for each of the operations one-by-one.

**Search** A search operation starts by invoking seek operation. It returns true if the stored key matches the target key and false otherwise.

**Insert** An insert operation (shown in Figure 5.1) starts by invoking seek operation. It returns false if the target key matches the stored key; otherwise, it moves to the execution

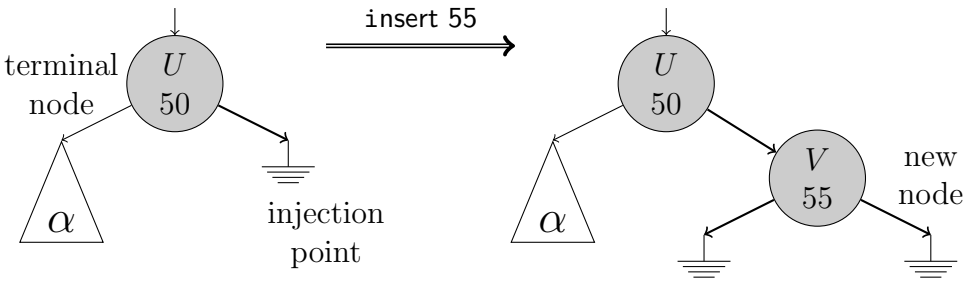



Figure 5.1. An illustration of an insert operation.

phase. In the execution phase, it attempts to insert the key into the tree as a child node of the last node in the access-path using a CAS instruction. If the instruction succeeds, then the operation returns true; otherwise, it restarts by invoking the seek function again.

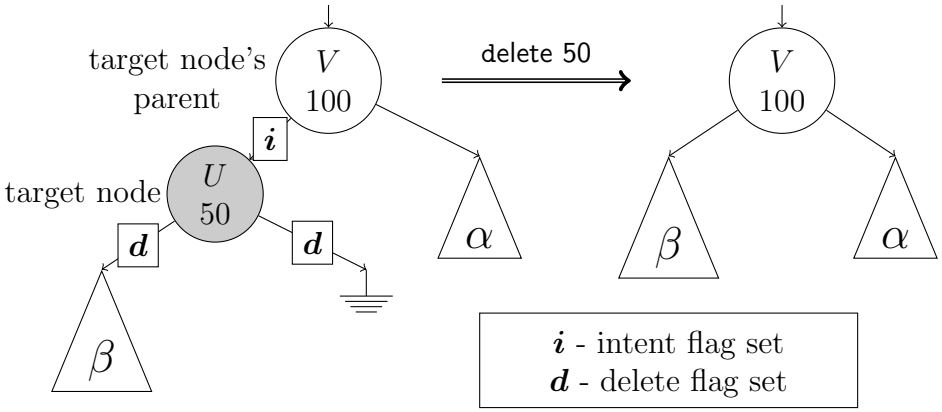



Figure 5.2. An illustration of a simple delete operation.

**Delete** A delete operation starts by invoking seek function. It returns false if the stored key does not match the target key; otherwise, it moves to the execution phase. In the execution phase, it attempts to remove the key stored in the terminal node of the access-path. There are two cases depending on whether the terminal node is a binary node (has two children) or not (has at most one child). In the first case, the operation is referred to as *complex delete operation*. In the second case, it is referred to as *simple delete operation*. In the case of simple delete (shown in Figure 5.2), the terminal node is removed by changing the pointer at the parent node of the terminal node. In the case of complex delete (shown in Figure 5.3),

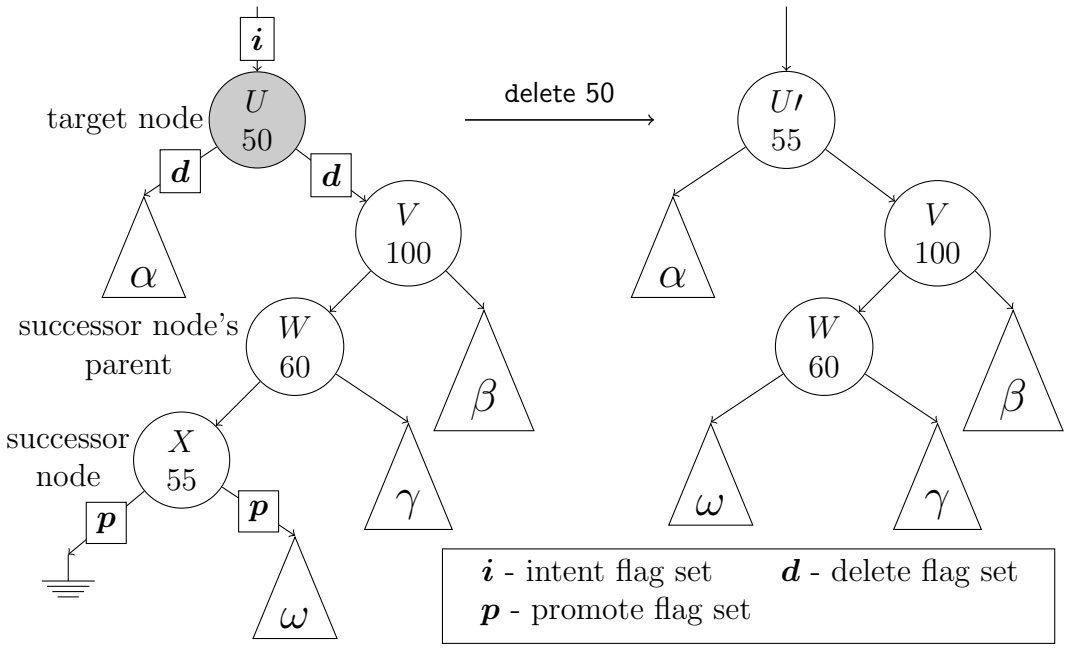



Figure 5.3. An illustration of a complex delete operation.

the key to be deleted is replaced with the *next largest* key in the tree, which will be stored in the *leftmost node* of the *right subtree* of the terminal node.

## 5.2 Details of the Algorithm

As in most algorithms, we use sentinel keys and three sentinel nodes to handle the boundary cases easily. The structure of an empty tree with only sentinel keys (denoted by $\infty_0$, $\infty_1$ and $\infty_2$ with $\infty_0 < \infty_1 < \infty_2$) and sentinel nodes (denoted by $\mathbb{R}$, $\mathbb{S}$ and $\mathbb{T}$) is shown in Figure 5.4.

Our algorithm, like the one in [31], operates at edge level. A delete operation obtains ownership of the edges it needs to work on by marking them. To enable marking, we steal bits from the child addresses of a node. Specifically, we steal *three* bits from each child address to distinguish between three types of marking: (i) marking for *intent*, (ii) marking for *deletion* and (iii) marking for *promotion*. The three bits are referred to as *intent-flag*, *delete-flag* and *promote-flag*. To avoid the ABA problem, as in Howley and Jones [20], we

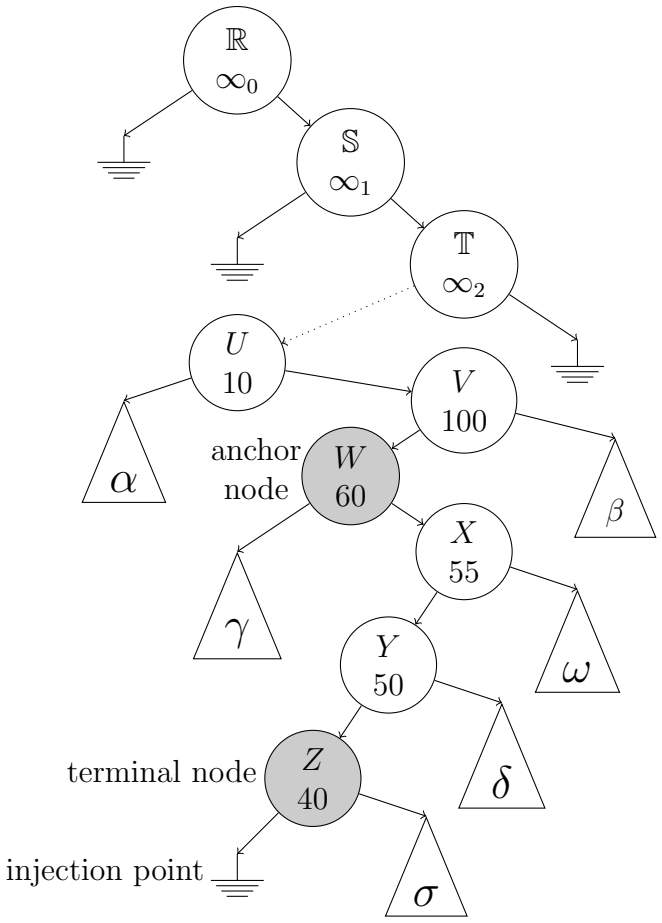



Figure 5.4. Nodes in the access path of seek along with sentinel keys and nodes ($\infty_0 < \infty_1 < \infty_2$)

use *unique* null pointers. To that end, we steal another bit from the child address, referred to as *null-flag*, and use it to indicate whether the address field contains a null or a non-null value. So, when an address changes from a non-null value to a null value, we only set the null-flag and the contents of the address field are not otherwise modified. This ensures that all null pointers are unique.

Finally, we also steal a bit from the key field to indicate whether the key stored in a node is the original key or the replacement key. This information is used in a complex delete operation to coordinate helping among processes.

We next describe the details of the seek function, which is used by all operations (search as well as modify) to traverse the tree after which we describe the details of the execution phase of insert and delete operations.

### 5.2.1 The Seek Phase

A seek function keeps track of the node in the access-path at which it took the last "right turn" (*i.e.*, it last followed a right edge). Let this "right turn" node be referred to as *anchor node* when the traversal reaches the terminal node. Note that the terminal node is the node whose key matched the target key or whose next child edge is set to a null address. For an illustration, please see Figure 5.4. In the latter case (stored key does not match the target key), it is possible that the key may have moved up in the tree. To ascertain that the seek function did not miss the key because it may have moved up during the traversal, we use the following set of conditions that are *sufficient* (but not necessary) to guarantee that the seek function did not miss the key. First, the anchor node is still part of the tree. Second, the key stored in the anchor node has not changed since it first encountered the anchor node during the (current) traversal. To check for the above two conditions, we determine whether the anchor node is undergoing removal (either delete or promote flag set) by examining its right child edge. We discuss the two cases one-by-one.

(a) *Right child edge not marked:* In this case, the anchor node is still part of the tree. We next check whether the key stored in the anchor node has changed. If the key has not changed, then the seek function returns the results of the (current) traversal, which consists of three addresses: (i) the address of the terminal node, (ii) the address of its parent, and (iii) the null address stored in the child field of the terminal node that caused the traversal to terminate. The last address is required to ensure that an insert operation works correctly (specifically to ascertain that the child field of the terminal node has not undergone any change since the completion of the traversal). We refer

to it as the *injection point* of the insert operation. On the other hand, if the key has changed, then the seek function restarts from the root of the tree.

(b) *Right child edge marked:* In this case, we compare the information gathered in the current traversal about the anchor node with that in the previous traversal, if one exists. Specifically, if the anchor node of the previous traversal is same as that of the current traversal and the keys found in the anchor node in the two traversals also match, then the seek function terminates, but returns the results of the previous traversal (instead of that of the current traversal). This is because the anchor node was definitely part of the tree during the previous traversal since it was reachable from the root of the tree at the beginning of the current traversal. Otherwise, the seek function restarts from the root of the tree.

The seek function also keeps track of the *second-to-last* edge in the access-path (whose endpoints are the parent and grandparent nodes of the terminal node), which is used for helping, if there is a conflict. For insert and delete operations, we refer to the terminal node as the *target node*.

### 5.2.2 The Execution Phase of an Insert Operation

In the execution phase, an insert operation creates a new node containing the target key. It then adds the new node to the tree at the injection point using a CAS instruction. If the CAS instruction succeeds, then (the new node becomes a part of the tree and) the operation terminates; otherwise, the operation determines if it failed because of a *conflicting* delete operation in progress. If there is no conflicting delete operation in progress, then the operation restarts from the seek phase; otherwise it performs helping and then restarts from the seek phase. Figure 5.5 shows a flow chart describing the sequence of steps of an insert operation.

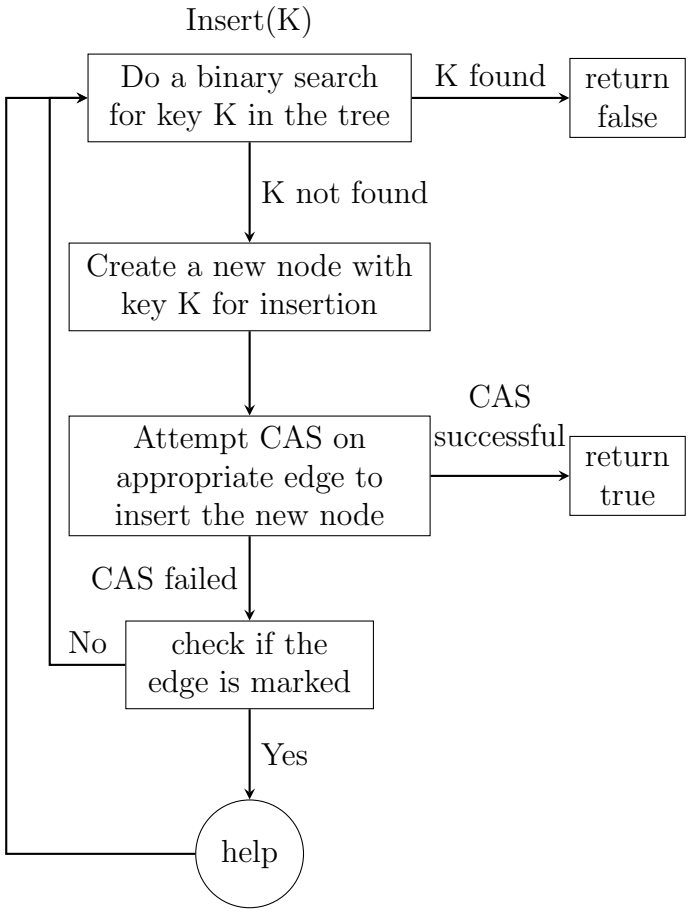



Figure 5.5. ELFtree - Sequence of steps in an insert operation

### 5.2.3 The Execution Phase of a Delete Operation

The execution of a delete operation starts in *injection mode*. Once the operation has been injected into the tree, it advances to either *discovery mode* or *cleanup mode* depending on the type of the delete operation.

**Injection Mode** In the injection mode, the delete operation marks the three edges involving the target node as follows: (i) It first sets the intent-flag on the edge from the parent of the target node to the target node using a CAS instruction. (ii) It then sets the delete-flag on the left edge of the target node using a CAS instruction. (iii) Finally, it sets the delete-flag on the right edge of the target node using a CAS instruction. If the CAS instruction fails at

any step, the delete operation performs helping, and either repeats the same step or restarts from the seek phase. Specifically, the delete operation repeats the same step when setting the delete-flag as long as the target node has not been claimed as the successor node by another delete operation. In all other cases, it restarts from the seek phase.

We maintain the invariant that an edge, once marked, cannot be unmarked. After marking both the edges of the target node, the operation checks whether the target node is a binary node or not. If it is a binary node, then the delete operation is classified as complex; otherwise it is classified as simple. Note that the type of the delete operation cannot change once all the three edges have been marked as described above. If the delete operation is complex, then it advances to the discovery mode after which it will advance to the cleanup mode. On the other hand, if it is simple, then it directly advances to the cleanup mode (and skips the discovery mode). Eventually, the target node is either removed from the tree (if simple delete) or replaced with a "new" node containing the next largest key (if complex delete).

For a tree node $X$, let $X.parent$ denote its parent node, and $X.left$ and $X.right$ denote its left and right child node, respectively. Also, hereafter in this section, let $T$ denote the target node of the delete operation under consideration.

**Discovery Mode** In the discovery mode, a complex delete operation performs the following steps:

1. **Find Successor Key:** The operation locates the next largest key in the tree, which is the smallest key in the subtree rooted at the right child of $T$. We refer to this key as the *successor key* and the node storing this key as the *successor node*. Hereafter in this section, let $S$ denote the successor node.
2. **Mark Child Edges of Successor Node:** The operation sets the promote-flag on both the child edges of $S$ using a CAS instruction. Note that the left child edge of $S$ will be

null. As part of marking the left child edge, we also store the address of $T$ (the target node) in the edge. This is done to enable helping in case the successor node is obstructing the progress of another operation. In case the CAS instruction fails while marking the left child edge, the operation repeats from step 1 after performing helping if needed. On the other hand, if the CAS instruction fails while marking the right child edge, then the marking step is repeated after performing helping if needed.

3. **Promote Successor Key:** The operation replaces the target node's original key with the successor key. At the same time, it also sets the mark bit in the key to indicate that the current key stored in the target node is the replacement key and not the original key.
4. **Remove Successor Node:** The operation removes $S$ (the successor node) by changing the child pointer at$S.parent$ that is pointing to $S$ to point to the right child of $S$ using a CAS instruction. If the CAS instruction succeeds, then the operation advances to the cleanup mode. Otherwise, it performs helping if needed. It then finds $S$ again by performing another traversal of the tree starting from the right child of $T$. If the traversal fails to find $S$ (recall that the left edge of $S$ is marked for promotion and contains the address of $T$), then $S$ has already been removed from the tree by another operation as part of helping, and the delete operation advances to the cleanup mode. On advancing to the cleanup mode, the operation sets a flag in $T$ indicating that $S$ has been removed from the tree (and $T$ can now be replaced with a new node) so that other operations trying to help it know not to look for $S$.

Figure 5.6 shows a flow chart describing the sequence of steps of a delete operation.

**Cleanup Mode** There are two cases depending on whether the delete operation is simple or complex.

(a) **Simple Delete:** In this case, either $T.left$ or $T.right$ is pointing to a null node. Note that both $T.left$ and $T.right$ may be pointing to null nodes (which in turn will imply

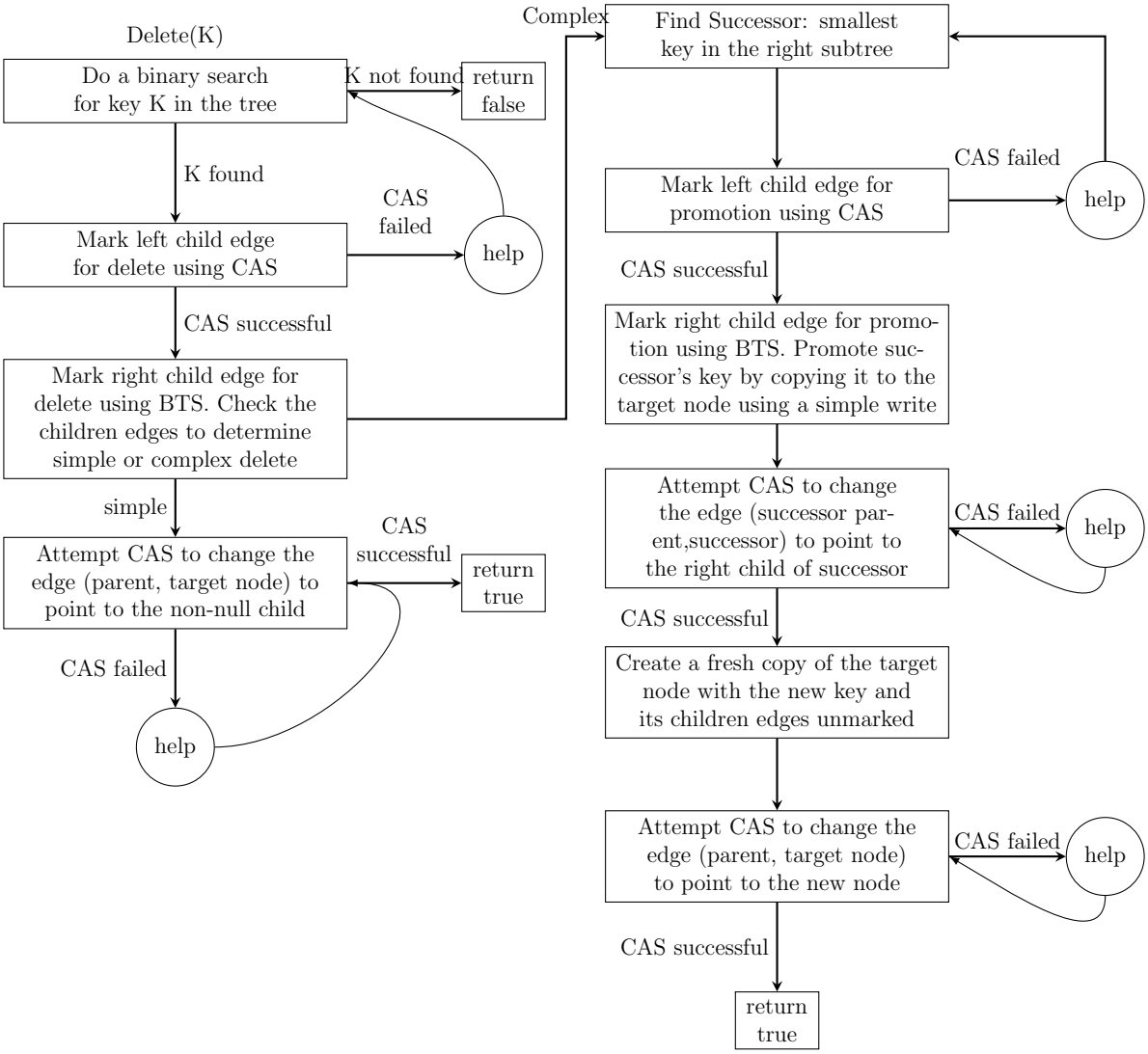



Figure 5.6. ELFTREE - Sequence of steps in a delete operation

**Algorithm 11:** Data Structures Used

```
173 struct Node {
174     {Boolean, Key} mKey;
175     {Boolean, Boolean, Boolean, Boolean, NodePtr} child[2];
176     Boolean readyToReplace;
177 };
178 struct Edge {
179     NodePtr parent, child;
180     enum which { LEFT, RIGHT };
181 };
182 struct SeekRecord {
183     Edge lastEdge, pLastEdge, injectionEdge;
184 };
185 struct AnchorRecord {
186     NodePtr node;
187     Key key;
188 };
189 struct StateRecord {
190     Edge targetEdge, pTargetEdge;
191     Key targetKey, currentKey;
192     enum mode { INJECTION, DISCOVERY, CLEANUP };
193     enum type { SIMPLE, COMPLEX } ;
        // the next field stores pointer to a seek record; it is used for finding the
           successor if the delete operation is complex
194     SeekRecordPtr successorRecord;
195 };
    // object to store information about the tree traversal when looking for a given key
       (used by the seek function)
196 SeekRecordPtr targetRecord := new seek record;
    // object to store information about process' own delete operation
197 StateRecordPtr myState := new state;
```

that $T$ is a leaf node). Without loss of generality, assume that $T.right$ is a null node. The removal of $T$ involves changing the child pointer at $T.parent$ that is pointing to $T$ to point to $T.left$ using a CAS instruction. If the CAS instruction succeeds, then the delete operation terminates; otherwise, it performs another seek on the tree. If the seek function either fails to find the target key or returns a terminal node different from $T$, then $T$ has been already removed from the tree (by another operation as part of helping) and the delete operation terminates; otherwise, it attempts to remove $T$ from

**Algorithm 12:** Seek Function

```
198 SEEK( key, seekRecord )
199 begin
200     pAnchorRecord := {𝕊, ∞1};
201     while true do
            // initialize all variables used in traversal
202         pLastEdge := {ℝ, 𝕊, RIGHT};      lastEdge := {𝕊, 𝕋, RIGHT};
203         curr := 𝕋;      anchorRecord := {𝕊, ∞1};
204         while true do
                // read the key stored in the current node
205             ⟨∗, cKey⟩ := curr→mKey;
                // find the next edge to follow
206             which := key < cKey ? LEFT: RIGHT;
207             ⟨n, ∗, d, p, next⟩ := curr→child[which];
                // check for the completion of the traversal
208             if key = cKey  or n then
                    // either key found or no next edge to follow; stop the traversal
209                 seekRecord→pLastEdge := pLastEdge;
210                 seekRecord→lastEdge := lastEdge;
211                 seekRecord→injectionEdge := {curr, next, which};
212                 if key = cKey then // keys match
213                     return;
214                 else break;
215             if which = RIGHT then
                    // next edge to be traversed is a right edge; keep track of the
                       current node and its key
216                 anchorRecord := ⟨curr, cKey⟩;
                // traverse the next edge
217             pLastEdge := lastEdge;      lastEdge := {curr, next, which};      curr := next;
            // key was not found; check if can stop
218         ⟨∗, ∗, d, p, ∗⟩ := anchorRecord.node→child[RIGHT];
219         if not (d)  and not (p) then
                // anchor node is still part of the tree; check if anchor node's key has
                   changed
220             ⟨∗, aKey⟩ := anchorRecord.node→mKey;
221             if anchorRecord.key = aKey then  return;
222         else
                // check if the anchor record (the node and its key) matches that of the
                   previous traversal
223             if pAnchorRecord = anchorRecord then
                    // return the results of the previous traversal
224                 seekRecord := pSeekRecord;
225                 return;
            // store the results of the traversal and restart
226         pSeekRecord := seekRecord;      pAnchorRecord := anchorRecord;
```

**Algorithm 13:** Search Operation

```
227 Boolean SEARCH( key )
228 begin
229     SEEK( key, mySeekRecord );
230     node := mySeekRecord→lastEdge.child;
231     ⟨∗, nKey⟩ := node→mKey;
232     if nKey = key then return true;
233     else return false;
```

**Algorithm 14:** Insert Operation

```
234 Boolean INSERT( key )
235 begin
236     while true do
237         SEEK( key, targetRecord );
238         targetEdge := targetRecord→lastEdge;
239         node := targetEdge.child;
240         ⟨∗, nKey⟩ := node→mKey;
241         if key = nKey then return false;
            // create a new node and initialize its fields
242         newNode := create a new node;
243         newNode→mKey := ⟨0_m, key⟩;
244         newNode→child[LEFT] := ⟨1_n, 0_i, 0_d, 0_p, null⟩;
245         newNode→child[RIGHT] := ⟨1_n, 0_i, 0_d, 0_p, null⟩;
246         newNode→readyToReplace := false;
247         which := targetRecord→injectionEdge.which;
248         address := targetRecord→injectionEdge.child;
249         result := CAS(node→child[which], ⟨1_n, 0_i, 0_d, 0_p, address⟩, ⟨0_n, 0_i, 0_d, 0_p, newNode⟩);
250         if result then return true;
            // help if needed
251         ⟨∗, ∗, d, p, ∗⟩ := node→child[which];
252         if d then HELPTARGETNODE( targetEdge ) ;
253         else if p then HELPSUCCESSORNODE( targetEdge ) ;
```

**Algorithm 15:** Delete Operation

```
254 Boolean DELETE( key )
255 begin
        // initialize the state record
256     myState→targetKey := key;        myState→currentKey := key;
257     myState→mode := INJECTION;
258     while true do
259         SEEK( myState→currentKey, targetRecord );
260         targetEdge := targetRecord→lastEdge;        pTargetEdge := targetRecord→pLastEdge;
261         ⟨∗, nKey⟩ := targetEdge.child→mKey;
262         if myState→currentKey ≠ nKey then
                // the key does not exist in the tree
263             if myState→mode = INJECTION then return false;
264             else return true;
            // perform appropriate action depending on the mode
265         if myState→mode = INJECTION then
                // store a reference to the target edge
266             myState→targetEdge := targetEdge;
267             myState→pTargetEdge := pTargetEdge;
                // attempt to inject the operation at the node
268             INJECT( myState );
            // mode would have changed if injection was successful
269         if myState→mode ≠ INJECTION then
                // check if the target node found by the seek function matches the one
                   stored in the state record
270             if myState→targetEdge.child ≠ targetEdge.child then return true;
                // update the target edge information using the most recent seek
271             myState→targetEdge := targetEdge;
272         if myState→mode = DISCOVERY then
                // complex delete operation; locate the successor node and mark its child
                   edges with promote flag
273             FINDANDMARKSUCCESSOR( myState );
274         if myState→mode = DISCOVERY then
                // complex delete operation; promote the successor node's key and remove
                   the successor node
275             REMOVESUCCESSOR( myState );
276         if myState→mode = CLEANUP then
                // either remove the target node (simple delete) or replace it with a new
                   node with all fields unmarked (complex delete)
277             result := CLEANUP( myState );
278             if result then return true;
279             else
280                 ⟨∗, nKey⟩ := targetEdge.child→mKey;
281                 myState→currentKey := nKey;
```

**Algorithm 16:** Injecting a Deletion Operation

```
282 INJECT( state )
283 begin
284     targetEdge := state→targetEdge;
        // try to set the intent flag on the target edge
        // retrieve attributes of the target edge
285     parent := targetEdge.parent;
286     node := targetEdge.child;
287     which := targetEdge.which;
288     result := CAS( parent→child[which],
                       ⟨0_n, 0_i, 0_d, 0_p, node⟩,
                       ⟨0_n, 1_i, 0_d, 0_p, node⟩ );
289     if not (result) then
            // unable to set the intent flag; help if needed
290         ⟨∗, i, d, p, address⟩ := parent→child[which];
291         if i then HELPTARGETNODE( targetEdge ) ;
292         else if d then
293             HELPTARGETNODE( state→pTargetEdge );
294         else if p then
295             HELPSUCCESSORNODE( state→pTargetEdge );
296         return;
        // mark the left edge for deletion
297     result := MARKCHILDEDGE( state, LEFT );
298     if not (result) then return;
        // mark the right edge for deletion; cannot fail
299     MARKCHILDEDGE( state, RIGHT );
        // initialize the type and mode of the operation
300     INITIALIZETYPEANDUPDATEMODE( state );
```

the tree again using possibly the new parent information returned by seek. This process may be repeated multiple times.

(b) **Complex Delete:** Note that, at this point, the key stored in the target node is the replacement key (the successor key of the target key). Further, the key as well as both the child edges of the target node are marked. The delete operation attempts to replace target node with a *new* node, which is basically a copy of target node except that all its fields are unmarked. This replacement of $T$ involves changing the child pointer at $T.parent$ that is pointing to $T$ to point to the new node. If the CAS instruction succeeds, then the delete operation terminates; otherwise, as in the case of simple delete,

**Algorithm 17:** Locating the Successor Node

```
301 FindAndMarkSuccessor( state )
302 begin
        // retrieve the addresses from the state record
303     node := state→targetEdge.child;
304     seekRecord := state→successorRecord;
305     while true do
            // read the mark flag of the key in the target node
306         ⟨m, ∗⟩ := node→mKey;
            // find the node with the smallest key in the right subtree
307         result := FindSmallest( state );
308         if m or not (result) then
                // successor node had already been selected before the traversal or the
                   right subtree is empty
309             break;
            // retrieve the information from the seek record
310         successorEdge := seekRecord→lastEdge;
311         left := seekRecord→injectionEdge.child;
            // read the mark flag of the key under deletion
312         ⟨m, ∗⟩ := node→mKey;
313         if m then // successor node has already been selected
314             continue;
            // try to set the promote flag on the left edge
315         result := CAS( successorEdge.child        →
                           child[LEFT],
                           ⟨1_n, 0_i, 0_d, 0_p, left⟩,
                           ⟨1_n, 0_i, 0_d, 1_p, node⟩ );
316         if result then break;
            // attempt to mark the edge failed; recover from the failure and retry if
               needed
317         ⟨n, ∗, d, ∗, ∗⟩ := successorEdge.child→child[LEFT];
318         if n and d then
                // the node found is undergoing deletion; need to help
319             HelpTargetNode( successorEdge );
        // update the operation mode
320     UpdateMode( state );
```

**Algorithm 18:** Removing the Successor Node

```
321 RemoveSuccessor( state )
322 begin
        // retrieve addresses from the state record
323     node := state→targetEdge.child;
324     seekRecord := state→successorRecord;
        // extract information about the successor node
325     successorEdge := seekRecord→lastEdge;
        // ascertain that seek record for successor node contains valid information
326     ⟨∗, ∗, ∗, p, address⟩ := successorEdge.child→child[LEFT];
327     if not (p) or (address ≠ node) then
328         node→readyToReplace := true;
329         UpdateMode( state );
330         return;
        // mark the right edge for promotion if unmarked
331     MarkChildEdge( state, RIGHT );
        // promote the key
332     node→mKey := ⟨1_m, successorEdge.child→mKey⟩;
333     while true do
            // check if the successor is the right child of the target node itself
334         if successorEdge.parent = node then
                // need to modify the right edge of target node whose delete flag is set
335             dFlag := 1;      which := RIGHT;
336         else
337             dFlag := 0;      which := LEFT;
338         ⟨∗, i, ∗, ∗, ∗⟩ := successorEdge.parent→child[which];
339         ⟨n, ∗, ∗, ∗, right⟩ := successorEdge.child→child[RIGHT];
340         oldValue := ⟨0_n, i, dFlag, 0_p, successorEdge.child⟩;
341         if n then                    // only set the null flag; do not change the address
342             newValue := ⟨1_n, 0_i, dFlag, 0_p, successorEdge.child⟩;
343         else                              // switch the pointer to point to the grand child
344             newValue := ⟨0_n, 0_i, dFlag, 0_p, right⟩ ;
345         result := CAS(successorEdge.parent→child[which],oldValue, newValue);
346         if result or dFlag then break;
347         ⟨∗, ∗, d, ∗, ∗⟩ := successorEdge.parent→child[which];
348         pLastEdge := seekRecord→pLastEdge;
349         if d and (pLastEdge.parent ≠ null) then
350             HelpTargetNode( pLastEdge );
351         result := FindSmallest( state );
352         lastEdge := seekRecord→lastEdge;
353         if not (result) or lastEdge.child ≠ successorEdge.child then
354             break;                        // the successor node has already been removed
355         else successorEdge := seekRecord→lastEdge ;
356     node→readyToReplace := true;
357     UpdateMode( state );
```

**Algorithm 19:** Cleaning Up the Tree

```
358 Boolean CLEANUP( state )
359 begin
360     ⟨parent, node, pWhich⟩ := state→targetEdge;
361     if state→type = COMPLEX then
            // replace the node with a new copy in which all fields are unmarked
362         ⟨∗, nKey⟩ := node→mKey;
363         newNode→mKey := ⟨0_m, nKey⟩;
            // initialize left and right child pointers
364         ⟨∗, ∗, ∗, ∗, left⟩ := node→child[LEFT];
365         newNode→child[LEFT] := ⟨0_n, 0_i, 0_d, 0_p, left⟩;
366         ⟨n, ∗, ∗, ∗, right⟩ := node→child[RIGHT];
367         if n then
368             newNode→child[RIGHT] := ⟨1_n, 0_i, 0_d, 0_p, null⟩;
369         else newNode→child[RIGHT] := ⟨0_n, 0_i, 0_d, 0_p, right⟩ ;
            // initialize the arguments of CAS instruction
370         oldValue := ⟨0_n, 1_i, 0_d, 0_p, node⟩;
371         newValue := ⟨0_n, 0_i, 0_d, 0_p, newNode⟩;
372     else // remove the node
            // determine to which grand child will the edge at the parent be switched
373         if node→child[LEFT] = ⟨1_n, ∗, ∗, ∗, ∗⟩ then
374             nWhich := RIGHT;
375         else nWhich := LEFT;
            // initialize the arguments of the CAS instruction
376         oldValue := ⟨0_n, 1_i, 0_d, 0_p, node⟩;
377         ⟨n, ∗, ∗, ∗, address⟩ := node→child[nWhich];
378         if n then // set the null flag only
379             newValue := ⟨1_n, 0_i, 0_d, 0_p, node⟩;
380         else // change the pointer to the grand child
381             newValue := ⟨0_n, 0_i, 0_d, 0_p, address⟩ ;
382     result := CAS( parent→child[pWhich],
                       oldValue, newValue );
383     return result;
```

**Algorithm 20:** Mark Child Edge

```
384 Boolean MarkChildEdge( state, which )
385 begin
386     if state→mode = INJECTION then
387         edge := state→targetEdge;
388         flag := DELETE_FLAG;
389     else
390         edge := (state→successorRecord)→lastEdge;
391         flag := PROMOTE_FLAG;
392     node := edge.child;
393     while true do
394         ⟨n, i, d, p, address⟩ := node→child[which];
395         if i then
396             helpeeEdge := {node, address, which};
397             HelpTargetNode( helpeeEdge );
398             continue;
399         else if d then
400             if flag = PROMOTE_FLAG then
401                 HelpTargetNode( edge );
402                 return false;
403             else return true;
404         else if p then
405             if flag = DELETE_FLAG then
406                 HelpSuccessorNode( edge );
407                 return false;
408             else return true;
409         oldValue := ⟨n, 0_i, 0_d, 0_p, address⟩;
410         newValue := oldValue | flag;
411         result := CAS( node → child[which],
                           oldValue,
                           newValue );
412         if result then break;
413     return true;
```

**Algorithm 21:** Find Smallest

```
414 Boolean FindSmallest( state )
415 begin
        // find the node with the smallest key in the subtree rooted at the right child
           of the target node
416     node := state→targetEdge.child;
417     seekRecord := state→seekRecord;
418     ⟨n, ∗, ∗, ∗, right⟩ := node→child[RIGHT];
419     if n then // the right subtree is empty
420         return false;

        // initialize the variables used in the traversal
421     lastEdge := ⟨node, right, RIGHT⟩;
422     pLastEdge := ⟨node, right, RIGHT⟩;
423     while true do
424         curr := lastEdge.child;
425         ⟨n, ∗, ∗, ∗, left⟩ := curr→child[LEFT];
426         if n then // reached the node with the smallest key
427             injectionEdge := ⟨curr, left, LEFT⟩;
428             break;

            // traverse the next edge
429         pLastEdge := lastEdge;
430         lastEdge := ⟨curr, left, LEFT⟩;

        // initialize seek record and return
431     seekRecord→lastEdge := lastEdge;
432     seekRecord→pLastEdge := pLastEdge;
433     seekRecord→injectionEdge := injectionEdge;
434     return true;
```

it performs another seek on the tree, this time looking for the successor key. If the seek function either fails to find the successor key or returns a terminal node different from $T$, then $T$ has been already replaced (by another operation as part of helping) and the delete operation terminates. Otherwise, it attempts to replace $T$ again using possibly the new parent information returned by seek. This process may be repeated multiple times.

**Discussion** It can be verified that, in the absence of conflict, a delete operation performs three atomic instructions in the injection mode, three in the discovery mode (if delete is complex), and one in the cleanup mode.

**Algorithm 22:** Helper Routines

```
435 InitializeTypeAndUpdateMode( state )
436 begin
        // retrieve the target node's address from the state record
437     node := state→targetEdge.child;
438     ⟨lN, *, *, *, *⟩ := node→child[LEFT];
439     ⟨rN, *, *, *, *⟩ := node→child[RIGHT];
440     if lN or rN then
            // one of the child pointers is null
441         ⟨m, *⟩ := node→mKey;
442         if m then state→type := COMPLEX;
443         else state→type := SIMPLE;
444     else // both child pointers are non-null
445         state→type := COMPLEX;
446     UpdateMode( state );
447 UpdateMode( state )
448 begin
        // update the operation mode
449     if state→type = SIMPLE then // simple delete
450         state→mode := CLEANUP;
451     else // complex delete
452         node := state→targetEdge.child;
453         if node→readyToReplace then
454             state→mode := CLEANUP;
455         else state→mode := DISCOVERY;
```

### 5.2.4 Helping

To enable helping, as mentioned earlier, whenever traversing the tree to locate either a target key or a successor key, we keep track of the *last two* edges encountered in the traversal. When a CAS instruction fails, depending on the reason for failure, helping is either performed along the last edge or the second-to-last edge.

## 5.3 Formal Description

A pseudo-code of our algorithm is given in Algorithms 11-23.

Algorithm 11 describes the data structures used in our algorithm. Besides Node, three important data types in our algorithm are: Edge, SeekRecord and StateRecord. The data

**Algorithm 23:** Helping Conflicting Delete Operations

```
456 HelpTargetNode( helpeeEdge )
457 begin
        // intent flag must be set on the edge
        // obtain new state record and initialize it
458     state→targetEdge := helpeeEdge;
459     state→mode := INJECTION;
        // mark the left and right edges if unmarked
460     result := MarkChildEdge( state, LEFT );
461     if not (result) then return;
462     MarkChildEdge( state, RIGHT );
463     InitializeTypeAndUpdateMode( state );
        // perform the remaining steps of a delete operation
464     if state→mode = DISCOVERY then
465         FindAndMarkSuccessor( state );
466     if state→mode = DISCOVERY then
467         RemoveSuccessor( state );
468     if state→mode = CLEANUP then Cleanup( state ) ;
469 HelpSuccessorNode( helpeeEdge )
470 begin
        // retrieve the address of the successor node
471     parent := helpeeEdge.parent;
472     node := helpeeEdge.child;
        // promote flat must be set on the successor node's left edge
        // retrieve the address of the target node
473     ⟨∗, ∗, ∗, ∗, left⟩ := node→child[LEFT];
        // obtain new state record and initialize it
474     state→targetEdge := {null, left, _};
475     state→mode := DISCOVERY;
476     seekRecord := state→successorRecord;
        // initialize the seek record in the state record
477     seekRecord→lastEdge := helpeeEdge;
478     seekRecord→pLastEdge := {null, parent, _};
        // promote the successor node's key and remove the successor node
479     RemoveSuccessor( state );
        // no need to perform the cleanup
```

type Edge is a structure consisting of three fields: the two endpoints and the direction (left or right). The data type SeekRecord is a structure used to store the results of a tree traversal. The data type StateRecord is a structure used to store information about a delete operation (*e.g.*, target edge, type, current mode, etc.). Note that only objects of type Node are shared between processes; objects of all other types (*e.g.*, SeekRecord, StateRecord) are *local* to a process and not shared with other processes.

The pseudo-code of the seek function is described in Algorithm 12, which is used by all the operations. The pseudo-codes of the search, insert and delete operations are given in Algorithm 13, Algorithm 14 and Algorithm 15, respectively. A delete operation executes function Inject in injection mode, functions FindAndMarkSuccessor and RemoveSuccessor in discovery mode and function Cleanup in cleanup mode. Their pseudo-codes are given in Algorithm 16, Algorithm 17, Algorithm 18 and Algorithm 19, respectively. The pseudo-codes for helper routines (used by multiple functions) are given in Algorithm 21, Algorithm 20 and Algorithm 22. Finally, the pseudo-codes of functions used to help other (conflicting) delete operations are given in Algorithm 23.

### 5.4 Correctness Proof

It can be shown that our algorithm satisfies linearizability and lock-freedom properties [18]. Broadly speaking, linearizability requires that an operation should appear to take effect instantaneously at some point during its execution. Lock-freedom requires that some process should be able to complete its operation in a finite number of its own steps. It is convenient to treat insert and delete operations that do not change the tree as search operations. We call a tree node *active* if it is reachable from the root of the tree. We call a tree node *passive* if it was active earlier but is not active any more. It can be verified that, if an insert operation completes successfully, then its target node was active when it performed the successful CAS instruction on the node's child edge. Likewise, if a delete operation completes successfully,

then its target node was active when it marked the node's left edge for deletion. Further, for a complex delete, the successor node was active when it marked the node's left edge for promotion.

**All Executions are Linearizable**

We show that an arbitrary execution of our algorithm is linearizable by specifying the *linearization point* of each operation. Note that the linearization point of an operation is the point during its execution at which the operation appeared to have taken effect. Our algorithm supports three types of operations: search, insert and delete. We now specify the linearization point of each operation.

1. *Insert operation:* The operation is linearized at the point at which it performed the successful CAS instruction that resulted in its target key becoming part of the tree.
2. *Delete operation:* There are two cases depending on whether the delete operation is simple or complex. If the operation is simple delete, then the operation is linearized at the point at which a successful CAS instruction was performed at the parent of the target node that resulted in the target node becoming passive. Otherwise, it is linearized at the point at which the original key of the target node was replaced with its successor key.
3. *Search operation:* There are two cases depending on whether the target node was active when the operation read the key stored in the node. If the target node was not active, then the operation is linearized at the point at which the target node became passive. Otherwise, it is linearized at the point at which the read action was performed.

It can be easily verified that, for any execution of the algorithm, the sequence of operations obtained by ordering operations based on their linearization points is legal, *i.e.*, all operations in the sequence satisfy their specification. This establishes that *our algorithm generates only linearizable executions.*

**All Executions are Lock-Free**

We say that the system is in a *quiescent state* if no modify operation completes hereafter. We say that the system is in a *potent state* if it has one or more pending modify operations. Note that a quiescence is a *stable* property; once the system is in a quiescent state, it stays in a quiescent state. We show that our algorithm is lock-free by proving that a potent state is necessarily non-quiescent provided assuming that some process with a pending modify operation continues to take steps.

Assume, by the way of contradiction, that there is an execution of the system in which the system eventually reaches a state that is potent as well as quiescent. Note that, once the system has reached a quiescent state, it will eventually reach a state after which the tree will not undergo any structural changes. This is because a modify operation makes at most two structural changes to the tree. So, if the tree is undergoing continuous structural changes, then it clearly implies that modify operations are continuously completing their responses, which contradicts the assumption that the system is in a quiescent state. Further, on reaching such a state, the system will reach a state after which no new edges in the tree are marked. Again, this is because a modify operation marks at most four edges and the set of edges in the tree does not change any more. We call such a system state after which neither the set of edges nor the set of *marked* edges in the tree change any more as a *strongly quiescent state*. Note that, like quiescence, strong quiescence is also a stable property.

From the above discussion, it follows that the system in a quiescent state will eventually reach a state that is strongly quiescent. Consider the search tree in such a strongly quiescent state. It can be verified that no more modify operations can now be injected into the tree, and, moreover, all modify operations already injected into the tree are delete operations currently "stuck" in either discovery or cleanup mode. Now, consider a process, say $p$, that continues to take steps to execute either its own operation or another operation blocking its progress (directly or indirectly) as part of helping. Consider the recursive chain of the *helpee*

operations that $p$ proceeds to help in order to complete its own operation. Let $\alpha_i$ denote the $i^{th}$ helpee operation in the chain. It can be shown that:

**Lemma 1.** *Let $\mathcal{C}_D$ denote the set of all complex delete operations already injected into the tree that are "stuck" in the discovery mode. Then,*

1. *$\alpha_1 \in \mathcal{C}_D$, and*
2. *Suppose $p$ is currently helping $\alpha_i$ for some $i \geq 1$ and assume that $\alpha_i \in \mathcal{C}_D$. Let $\alpha_{i+1}$ denote the next operation that $p$ selects to help. Then, (a) $\alpha_{i+1}$ exists, (b) $\alpha_{i+1} \in \mathcal{C}_D$, and (c) the target node of $\alpha_{i+1}$ is at strictly larger depth than the target node of $\alpha_i$.*

Using the above lemma, we can easily construct a chain of distinct helpee operations whose length exceeds the number of processes—a contradiction. This establishes that *our algorithm only generates lock-free executions.*

# PART II

# OPTIMIZATIONS

# CHAPTER 6

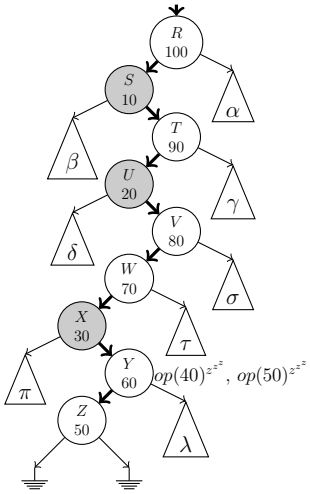
# LOCAL RECOVERY FOR CONCURRENT BINARY SEARCH TREES

In this chapter we describe our local recovery algorithm for concurrent internal binary search trees. We first present the main idea behind the algorithm and then provide its pseudo-code with more details.

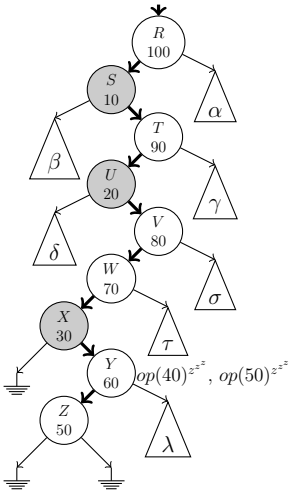
## 6.1 Overview of the Algorithm

As mentioned earlier, every operation on a BST involves first traversing the tree from top to down starting from the root node and following either the left or the right child pointer until either the target key is found or a null pointer is encountered (termination condition). Depending on the outcome of the traversal and the type of the operation, the tree may then need to be modified to actually realize the operation. We refer to the period during which the tree is being traversed as *seek phase*. Further, we refer to the period during which the tree is being modified as *execution phase*.

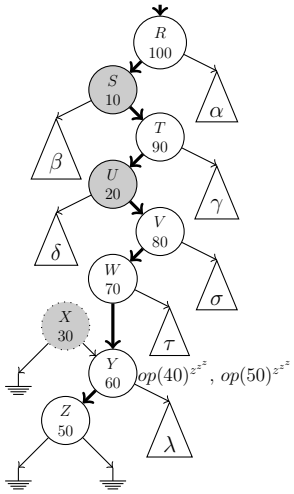
During the seek phase, the target key may move from its current location to a new location up the tree. As a result, the traversal may miss the key both at its old location as well as its new location. For an illustration, see Figure 6.1. In the illustration, key 50 has moved up by five nodes. In most concurrent BST algorithms, if it is suspected that the key may have moved up the tree, then the traversal is simply restarted from the root node. Different algorithms use different approaches to detect possible key movement. For example, in [20], the traversal is restarted if the *last right-turn* node is detected to have undergone some change. For example, in Figure 6.1a, the last right-turn node for the operation $op(50)$ is node $X$. On reaching the terminal node in Figure 6.1d, after resuming running, $op(50)$ needs to restart since $X$ has since been removed from the tree.



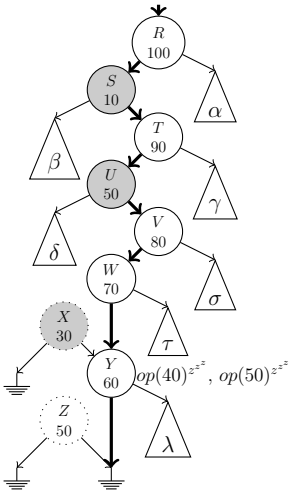

(a) Operation $op(50)$ is suspended at node $Y$ during its traversal.




(b) All keys in subtree $\pi$ are deleted one-by-one.




(c) Key 30 is deleted (simple delete); node $X$ is removed.




(d) Key 20 is deleted (complex delete); key 20 is replaced with key 50 in node $U$ and node $Z$ is removed.

Figure 6.1. An illustration of a key moving up the tree

A *re*-traversal of the tree may also be required if the operation encounters any failure during the execution phase. For example, in [34], which is a lock-based algorithm, execution phase is aborted if, after locking the relevant edges, the validation step fails. This happens if the portion of the tree that lies within the "operation's window", which typically consists of a small constant number of nodes, has undergone some change since it was last observed. In that case, the operation moves to the seek phase again.

In most concurrent BST algorithms, (a single instance of) the execution phase of an operation typically tends to have constant time complexity. The seek phase is where an operation may end up spending most of its time especially if the tree is large. Hence, it is desirable to make the seek phase of an operation more efficient by: (i) reducing the number of restarts due to "suspected" key movement, and (ii) restarting the traversal from a point "close" to the operation's window. This leads to two separate but related questions that any local recovery algorithm needs to address. First, "If a key is not found, then does the traversal need to restart?". Second, "If the traversal needs to be restarted, then from which node should the traversal restart?"

Consider an operation $\alpha$ currently traversing the tree; let $\Pi(\alpha)$ denote the path taken by $\alpha$ so far. For example, in Figure 6.1a, considering only the subtree shown in the figure, $\Pi(op(50)) = \langle R, S, T, U, V, W, X, Y \rangle$. At each node in the path (except the last node), $\alpha$ either followed the left or the right child pointer. We say that a node in the path is an *anchor* node if the operation followed its *right* child pointer; otherwise we say that is a *non-anchor* node. For example, in the path $\Pi(op(50))$, nodes $S$, $U$ and $X$ are anchor nodes, whereas nodes $R$, $T$, $V$, $W$ and $Y$ are non-anchor nodes. We assume that the first anchor node in a traversal path is always a *sentinel* node that is never marked (not shown in the figure). Of course, as an operation is traversing the tree, the tree may undergo changes as a result of which the path taken by the operation may no longer be correct. For example, in Figure 6.1d, the new access-path of $op(50)$ in the subtree, which is obtained from the subtree in Figure 6.1a after applying several delete operations, is now given by $\langle R, S, T, U \rangle$.

Note that, since in a complex delete operation we assume that the key being deleted is replaced with its successor key, the value of a key stored in a node can only increase. Therefore, the child pointer followed by an operation at a node, if it is still part of the access-path, may change (from right to left) for an anchor node but cannot change for a non-anchor node. For example, as shown in Figure 6.1a, the node $U$ is an anchor node for $op(40)$. But due to the changes made to the tree, the key at $U$ has now become 50, as shown in Figure 6.1d. Hence, the pointer that $op(40)$ now needs to follow at $U$ is left and not right. We say that an anchor node is *consistent* with respect to an operation if its key is still less than the operation's key; otherwise, we say that it is *inconsistent*. For example, in Figure 6.1d, anchor nodes $S$ and $X$ are still consistent with respect to $op(40)$ but node $U$ has become inconsistent with respect to $op(40)$.

Clearly, in case an operation needs to restart, *no node* in the path *after an inconsistent anchor node* in general can serve as a restart point since the path taken and the path that needs to be taken may now diverge. This implies that, to find a restart point, a local recovery algorithm should locate the *shallowest* inconsistent anchor node in the path (with the least depth) and discard the suffix of the path after such a node. Moreover, a restart point has to be a node that is still a part of the tree.

This leads to the following approach to find a restart point for an operation $\alpha$ when needed. Find a node $C$ in the path taken so far by $\alpha$ such that the following two conditions hold. First, $C$ is not marked. Second, every anchor node in the path preceding $C$ is consistent with respect to $\alpha$. To check for the second condition, it is not necessary to examine every anchor node in the path preceding $C$ as stated in the following lemma.

**Lemma 2.** *Consider an operation $\alpha$ and let $\Pi(\alpha)$ denote the path taken by $\alpha$ when traversing the tree. Let $A$ be an anchor node in $\Pi(\alpha)$ and let $\sigma = A_0, A_1, \ldots, A_k$, where $A_k = A$, denote the sequence of anchor nodes in $\Pi(\alpha)$ up to and including $A$. Then, if $A$ is unmarked and*

*consistent with respect to $\alpha$, then, for every $i$ with $0 \leq i \leq k$, $A_i$ is also consistent with respect to $\alpha$. Moreover, the access-path of $\alpha$ in the current tree includes $A$.*

We say that an anchor node is *critical* with respect to a node $C$ in the path if it is the *closest* preceding anchor node to $C$ that is also unmarked. For example, in Figure 6.1c, the critical anchor node with respect to node $Y$ is node $U$ since node $X$ is marked. Using the above lemma, the second condition can now be replaced with the following: the critical anchor node with respect to $C$ in the path, say $A$, as well as every anchor node in the path that lies between $A$ and $C$, which will be marked, is consistent with respect to $\alpha$.

The next lemma states a useful property about an inconsistent anchor node.

**Lemma 3.** *Consider an operation $\alpha$ with target key $k$ and let $\Pi(\alpha)$ denote the path taken by $\alpha$ when traversing the tree. Let $A$ be an anchor node in $\Pi(\alpha)$. Assume that $A$ is now inconsistent with respect to $\alpha$. Then, at the time $A$ became inconsistent, the tree did not contain $k$.*

To see why the above lemma holds, let $k_{old}$ ($k_{new}$) be the key stored in $A$ just before (after) $A$ became inconsistent. Clearly, $k_{old} < k < k_{new}$. Let $t$ denote the time just after which $k_{old}$ was replaced with $k_{new}$ in $A$. Note that $k_{new}$ must be the next smallest key in the right subtree of $A$ at time $t$. This implies that the right subtree of $A$ did not contain $k$ at time $t$. Further, from Lemma 2, $A$ is on the access-path of $\alpha$ at time $t$. Hence, we can conclude that the tree does not contain $k$ at time $t$. Note that, if a key is not present in the tree at some point while a search/delete operation is in progress, it is acceptable for the operation to say that key was not found. In this case, the operation will be linearized after the delete operation that removed the key from the tree.

When an operation fails to find the target key after traversing the tree from top to bottom, it examines the path it took to check whether or not the key has moved up the tree and/or a re-traversal is required. To that end, it examines the anchor nodes in the reverse

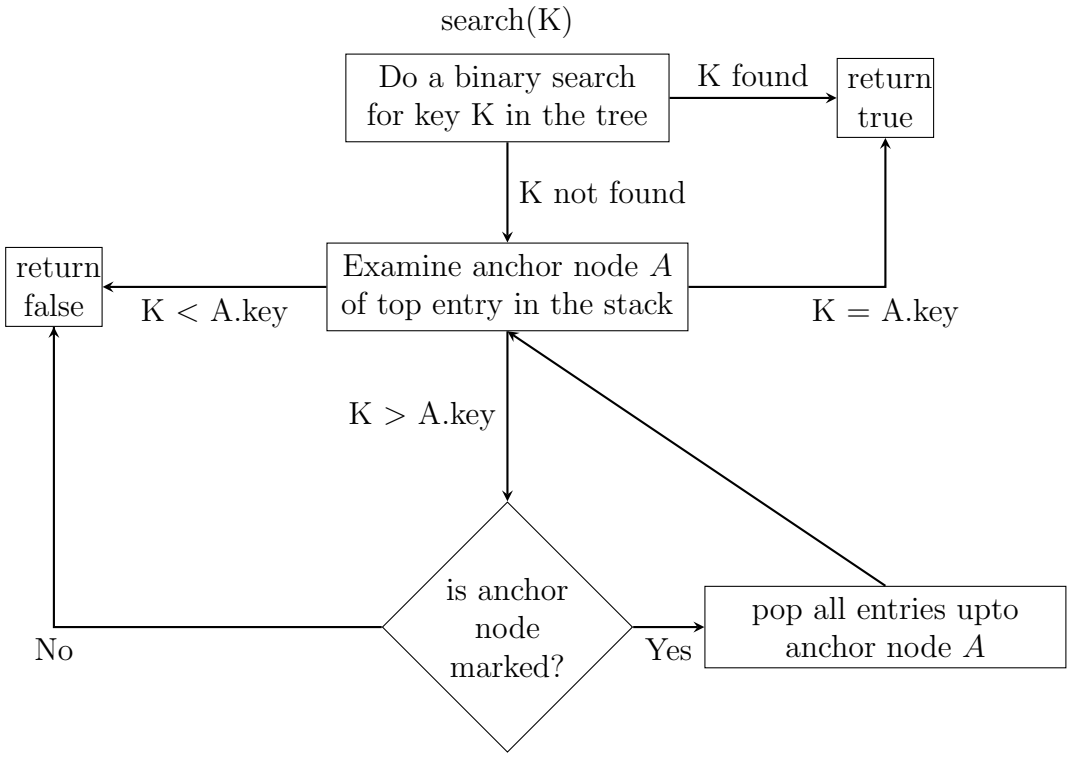
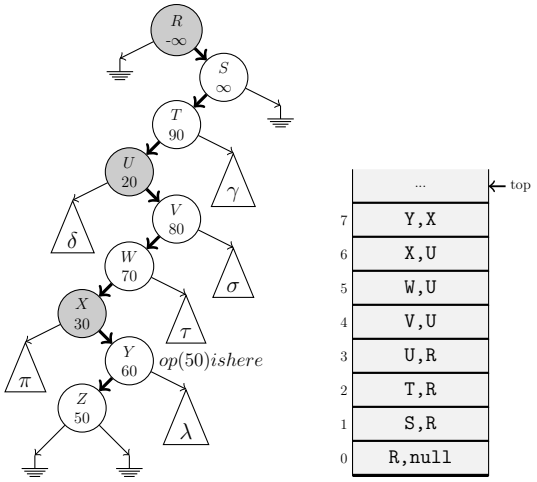


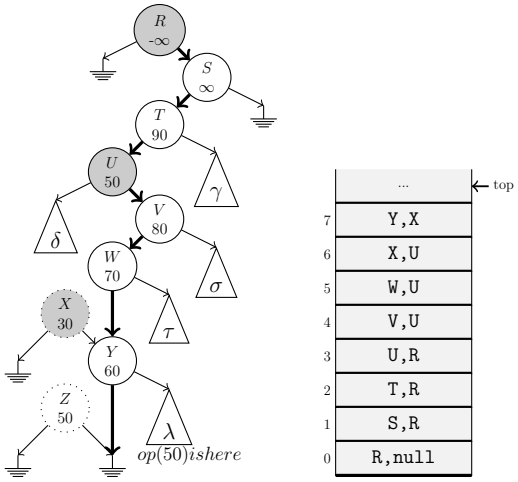

Figure 6.2. Sequence of steps in a search operation

order in which they were visited, starting from the one closest to the terminal node. We now discuss the behaviour of each operation one-by-one.

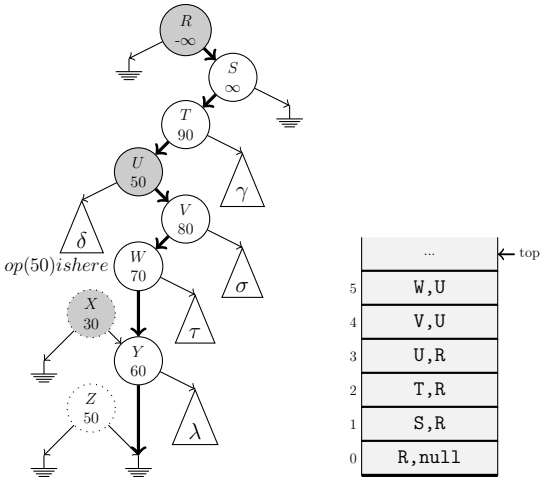
**Search Operation:** A search operation *does not* need to restart. When it examines an anchor node as mentioned above, there are three possibilities. First, if the anchor node's key matches the target key, then the key has been found and the operation terminates. Second, if the anchor node's key is greater than the target key (the anchor node has become inconsistent), then the operation concludes that the key is not present in the tree and terminates. Finally, if the anchor node's key is smaller than the target key (the anchor node is still consistent), then the operation terminates if the node is not marked; otherwise it moves to the preceding anchor node and repeats the comparison. Figure 6.2 gives an overview of the steps involved in a search operation. Figure 6.3 shows an example of how a stack is used to




(a) Operation $op(50)$ starting at R and suspended at Y along with the stack




(b) Key 30 is deleted;key 20 is deleted & replaced with key 50 in node $U$ and node $Z$ is removed




(c) Pop upto marked anchor node $X$. Top of stack is now $W$. Examine anchor node $U$

Figure 6.3. An illustration of local recovery for a search operation

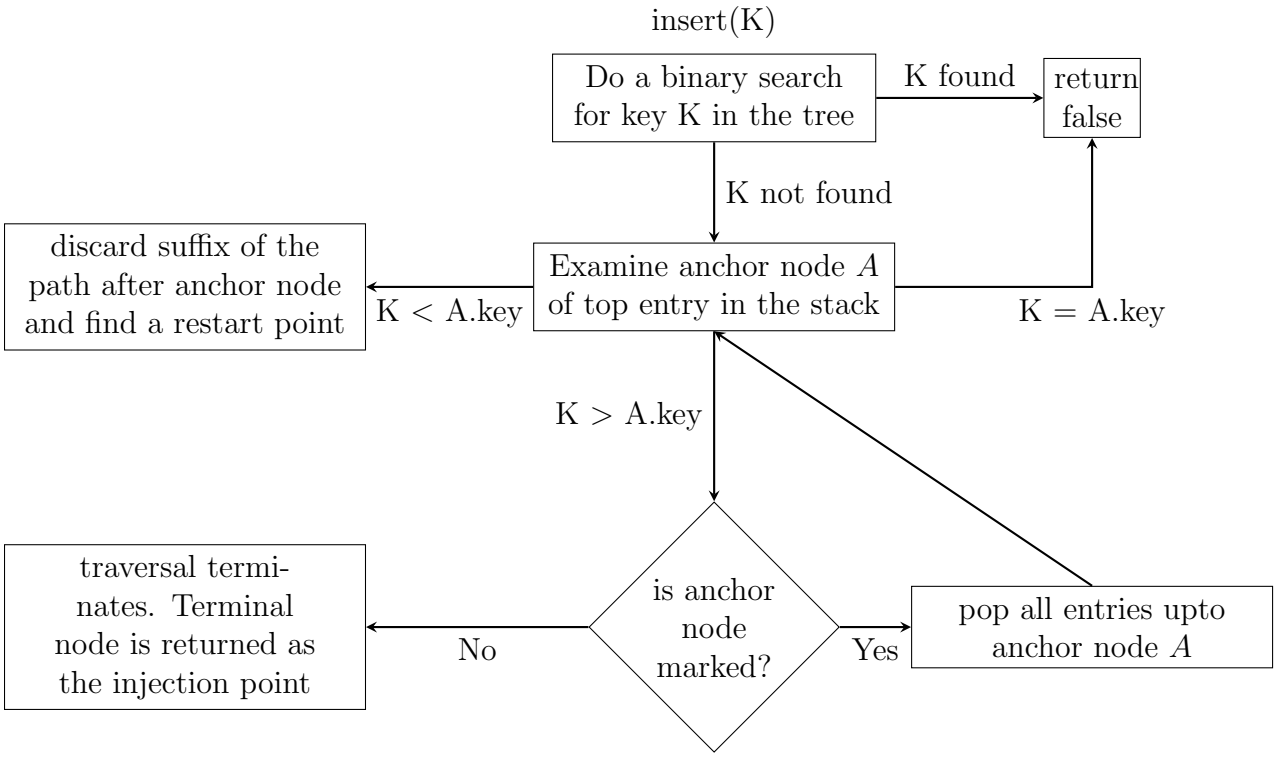



Figure 6.4. Sequence of steps in an insert operation

do local recovery. Each entry in the stack has the node encountered in the traversal along with its anchor node.

**Insert Operation:** An insert operation needs to restart only if one of the anchor nodes in the path has become inconsistent. When it examines an anchor node as mentioned above, there are three possibilities. First, if the anchor node's key matches the target key, then the key has been found and the operation terminates. Second, if the anchor node's key is greater than the target key (the anchor node has become inconsistent), then it discards the suffix of the path after the anchor node and restarts the traversal from a restart point. Finally, if the anchor node's key is smaller than the target key (the anchor node is still consistent), then the traversal terminates if the node is not marked (the terminal node is

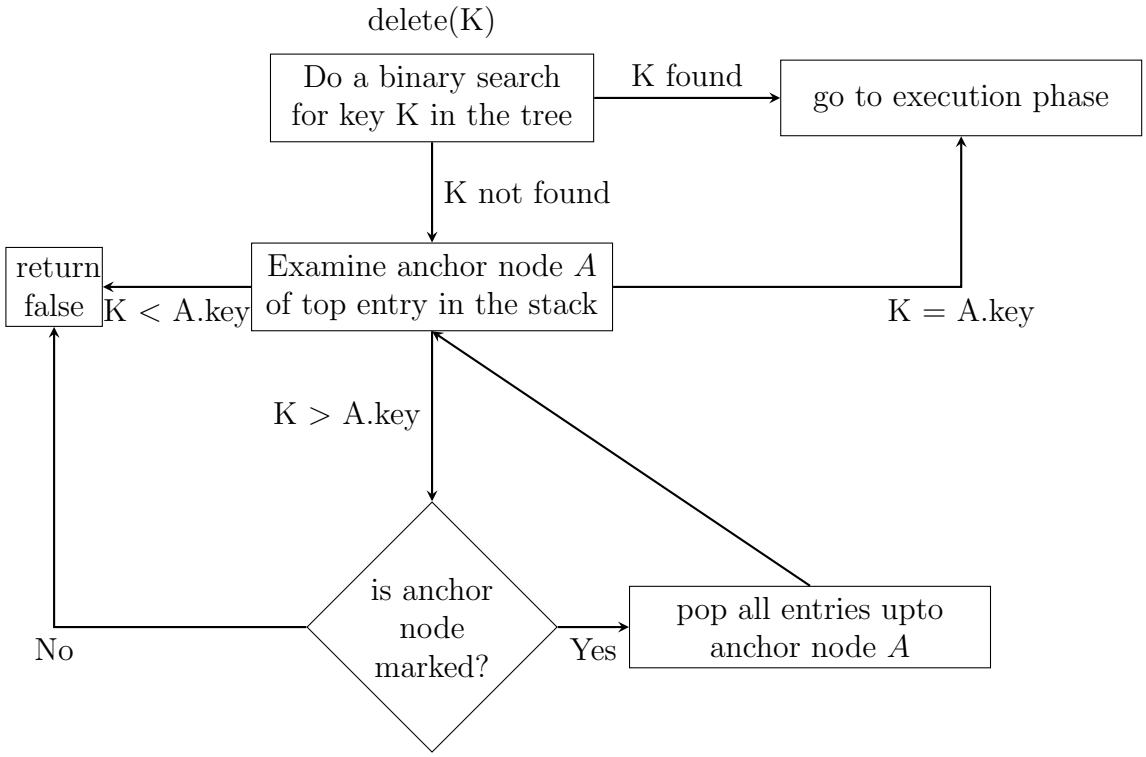



Figure 6.5. Sequence of steps in a delete operation

returned as the injection point); otherwise it moves to the preceding anchor node and repeats the comparison. Figure 6.4 gives an overview of the steps involved in an insert operation.

**Delete Operation:** A delete operation also *does not* need to restart except when there is a failure in the execution phase. When it examines an anchor node as mentioned above, there are three possibilities. First, if the anchor node's key matches the target key, then the key has been found and the operation moves to the execution phase. Second, if the anchor node's key is greater than the target key (the anchor node has become inconsistent), then the operation concludes that the key is not present in the tree and terminates. Finally, if the anchor node's key is smaller than the target key (the anchor node is still consistent), then the traversal terminates if the node is not marked; otherwise it moves to the preceding anchor node and repeats the comparison. Figure 6.5 gives an overview of the steps involved in a delete operation.

**Algorithm 24:** Data Structures Used

```
   // Used to store information about a node visited during tree traversal
480 struct StackEntry {
481     NodePtr node;
482     enum Direction which;
483     integer anchor;
484 };
   // Used to store the path from the root node to the current node in the tree
485 struct State {
486     StackEntry[ ] stack;
487     integer top;
488 };
   // Used to store information about the operation currently in progress
489 struct OpRecord {
490     enum Type type;
491     Key key;
492     State targetStack, successorStack;
493     NodePtr injectionPoint;
        // algorithm-specific fields
494 };
   // Used to store the outcome of a tree traversal
495 struct SeekRecord{
        // algorithm-specific fields (e.g., target node and its parent)
496 };
```

## 6.2 Details of the Algorithm

A pseudo-code of the local recovery algorithm is given in Pseudo-codes 24-31. The pseudo-code only shows the seek phase of an algorithm and not its execution phase since the execution phase is algorithm-specific. We have also moved the pseudo-code for local recovery when looking for a successor key to the appendix due to lack of space.

The local recovery algorithm assumes that the original algorithm supports the following functions: (a) GetKey( ), IsMarked( ) and GetChild( ) returns the various attributes of a tree node, (b) IsNull( ) returns true if a reference is null and false otherwise, (c) GetAddress( ) returns the node address stored in a reference, if non-null, (d) Move( ) enables the original algorithm to move along an edge, which may invoke helping and restarting of the traversal as in [20], (e) NeedCleanParentNode( ) returns true if the operation needs the

**Algorithm 25:** Functions for Manipulating Traversal Stack

```
    // Returns the number of elements in the stack
497 integer SIZE( state )
498 begin
499     return state→top + 1;

    // Returns the topmost node in the stack
500 NodePtr GETTOP( state )
501 begin
502     {stack, top} := state;
503     return stack[top]→node;

    // Returns the second topmost node in the stack
504 NodePtr GETSECONDTOTOP( state )
505 begin
506     {stack, top} := state;
507     return stack[top − 1]→node;

    // Adds the given node to the stack along with its anchor node
508 ADDTOTOP( state, node, which )
509 begin
510     {stack, top} := state;
        // find the anchor node
511     if which = RIGHT then anchor := top ;
512     else anchor := stack[top]→anchor ;
        // push the node into the stack
513     stack[top + 1] := {node, which, anchor};
514     state→top := top + 1;

    // Removes the topmost node from the stack
515 REMOVEFROMTOP( state )
516 begin
517     {stack, top} := state;
        // update the anchor node of the penultimate entry if needed
518     anchor := stack[top − 1]→anchor;
519     if stack[top]→anchor < stack[anchor]→anchor then
520         stack[anchor]→anchor := stack[top]→anchor;
        // pop the node from the stack
521     state→top := top − 1;

    // Pops the stack until a given entry
522 REMOVEUNTIL( state, index )
523 begin
524     state→top := index;
```

parent node to be clean and have no operation in progress (needed for a delete operation since it needs to modify a child pointer at the parent node), and (f) POPULATESEEKRECORD( )

**Algorithm 26:** Functions for Manipulating Traversal Stack (Continued)

```
    // Remember the critical node (to avoid locating it again)
525 RememberCritical( state, critical )
526 begin
527     {stack, top} := state;
528     anchor := stack[top]→anchor;
529     if critical < stack[anchor]→anchor then
530         stack[anchor]→anchor := critical;

    // Returns a given entry in the stack
531 { NodePtr, enum Direction, integer }
532                     GetFullEntry( state, index )
533 begin
534     {stack, top} := state;
535     if index = ⊤ then return stack[top] ;
536     else return stack[index] ;

    // initializes the traversal stack
537 InitializeTraversalState( state, type )
538 begin
539     if type = TARGET_STACK then
            // initialize the stack using sentinel nodes
            // sentinel nodes are never removed from the stack
            // a sentinel node is always a safe starting point for the traversal
540     else state→top := -1 ;
```

copies the relevant information from the traversal state required by the algorithm into a seek record.

Pseudo-code 24 shows the data structures used by the local recovery algorithm. Note that all the data structures shown in Pseudo-code 24 are *local* to a process not shared among processes. A process uses three main data structures, namely State, OpRecord and SeekRecord. A State (lines 485-488) is essentially a stack used to store the nodes visited during tree traversal when looking for a key (target or successor). Note that the traversal stack satisfies the last-in-first-out (LIFO) semantics but our algorithm sometimes uses it in a non-traditional way by accessing entries in the middle of the stack. One way to implement such an "augmented" stack is to use an auto-resizing vector provided as part of C++ STL library or Java package. Each entry in a traversal stack (lines 480-484) stores the address of the node, the location of its closest anchor node (within the stack's vector) and whether

the node is a left or right child of its parent. An OpRecord (lines 489-494) stores information about the operation such as type and key as well two stacks: one used when looking for the target key (all operations) and one used when looking for the successor key (only complex delete operations). Finally, a SeekRecord (lines 495-496) is used to return the outcome of a tree traversal to the original algorithm. Its fields are algorithm-specific. For example, for CASTLE, SeekRecord contains three fields: (a) two addresses, namely those of the target node and its parent, and (b) the contents of the injection point where an insert operation needs to attach the new node.

Pseudo-code 25 shows the functions used to manipulate a traversal stack. The function Size (lines 497-499) returns the number of entries in the stack. The functions GetTop (lines 500-503) and GetSecondToTop (lines 504-507) return the address of the node stored in the topmost entry and the entry below it, respectively. The function AddToTop (lines 508-514) adds an entry to the top of the stack while RemoveFromTop (lines 515-521) removes an entry from the top of the stack. The function RemoveUntil (lines 522-524) removes the entries from the top of the stack until a given point. The function RememberCritical (lines 525-530) updates the anchor field of the anchor node of the topmost entry in the stack. The function GetFullEntry (lines 532-536 returns all the three fields of a given entry in the stack (may not be the topmost entry). The function InitializeTraversalState (lines 537-540) initializes a traversal stack. The stack for target key

Pseudo-codes 27 & 28 shows the functions used to find the target key by a search operation. The function SeekForSearch (lines 572-578) first traverses the tree starting from the root node (line 574). If the traversal fails to locate the key, then the key may have moved up the tree. To address this possibility, the function examines the traversal stack to determine whether or not that is the case (line 576). The function Traverse (lines 541-554) first initializes the traversal stack (line 544) and then, starting from the topmost node in the stack (line 545), follows either the left or the right child pointer (line 548) until it either

finds the key (line 550) or encounters a null pointer (line 551). It also populates the traversal stack as it moves (line 554). The function ExamineStack(lines 555-571) examines the anchor nodes stored in the stack in the reverse order in which they were visited, starting from the anchor node closest to the topmost node in the traversal stack (line 559). If the anchor node's key matches the target key, then the function returns true (lines 564-566). If the anchor node is no longer consistent or is unmarked, then the function returns false (lines 567-568). Otherwise, the function backtracks and examines the preceding anchor node in the stack (lines 569-570).

Pseudo-codes 29-30 & 31 show the functions used to find the target key by a modify (insert or delete) operation. The function SeekForModify (lines 615-643) first backtracks to a safe node in the stack (line 620). Initially, the starting point is typically a sentinel node which is a safe node. The function then traverses the tree from top to down by following either the left or the right child pointer (line 625) until it either finds the key or encounters a null pointer (lines 627-636). In case the terminal node's key is greater than the target key, the function checks whether the path stored in the traversal stack is still valid (line 629). If not, the traversal is restarted. As the traversal moves down the tree, the function also populates the traversal stack (lines 637-641). The function ValidatePath (lines 579-600) checks whether or not the path stored in the stack is still valid. To that end, it examines the anchor nodes in the stack in the reverse order in which they were visited, starting from the anchor node closest to the topmost node in the traversal stack. There are three possible cases. First, the anchor node is still consistent (lines 586-591). In this case, the path is deemed to be valid if the anchor node is unmarked; otherwise, the function moves to the preceding anchor node. Second, the anchor node is no longer consistent (lines 592-597). In this case, the path is deemed to be invalid. However, if the operation is a delete operation, then it can be deduced that the key did not exist in the tree continuously and the function returns indicating that the key was not found (thereby causing the operation to terminate).

Finally, the anchor node's key matches the target key (lines 598-600). In this case, if the anchor node is marked and the operation is a delete operation, then the path is deemed to be invalid (and further backtracking is required). This is because the key may be in the process of moving up the tree. Otherwise, the function returns indicating that the key was found. The function FindASafeNode (lines 601-614) finds a safe node on the path stored in the stack from which the operation can restart its traversal. To that end, it backtracks to an unmarked node with a clean parent if required (lines 604-612). It then checks whether or not the remaining path in the stack is still valid (line 613). If not, it repeats the above-mentioned steps.

Pseudo-code 32 shows the function SeekForSuccessor used to locate the successor key by a complex delete operation (lines 644-679). The function first backtracks to an unmarked node with a clean parent if required (lines 648-657). It then checks whether or not the successor key is still needed by invoking NeedSuccessorKey function (line 658). The function NeedSuccessorKey returns a reference, which is null if the successor key is no longer needed and contains the address of the target node's right child otherwise. This address is used as a traversal point if the stack only contains a single entry (the node whose key needs to be replaced). If the successor key is still needed, then the function repeatedly follows the left child pointer until it encounters a null pointer (lines 667-676). While moving down the tree, the function also populates the traversal stack (line 673).

**Algorithm 27:** Seek Function for Target Key (Search Operation)

```
   // Traverses the tree starting from the root until either the key is found or a null
      pointer is encountered
541 boolean TRAVERSE( opRecord, seekRecord )
542 begin
543     state := opRecord→targetStack;
        // initialize the stack and the variables used in the traversal
544     INITIALIZETRAVERSALSTATE( state, TARGET_STACK );
545     current := GETTOP( state );
        // traverse the tree (starting from current)
546     while true do
547         key := GETKEY( current );
548         which := opRecord→key < key ? LEFT : RIGHT;
            // read the next address to de-reference
549         reference := GETCHILD( current, which );
550         if opRecord→key = key then return true ;
551         if ISNULL( reference ) then return false ;
            // traverse the next edge
552         address := GETADDRESS( reference );
553         current := address;
            // push the next node to be visited into the stack
554         ADDTOTOP( state, address, which );

   // Checks if the key being searched for has moved up in the path stored in the stack
555 boolean EXAMINESTACK( opRecord, seekRecord )
556 begin
557     result := false;
558     state := opRecord→targetStack;
        // start with the anchor closest to the topmost node in the stack
559     {∗, ∗, critical} := GETFULLENTRY( state, ⊤ );
560     while true do
            // retrieve the node and its closest anchor node from the stack
561         {node, ∗, anchor} := GETFULLENTRY( state, critical );
            // read the attributes of the node
562         marked := ISMARKED( node );
563         key := GETKEY( node );
564         if opRecord→key = key then
                // the key stored in the node matches the one being searched for
565             result := true;
566             break;
567         else if (opRecord→key < key) or not (marked) then
                // the target key did not exist continuously in the tree
568             break;
569         else // examine the preceding anchor node
570             critical := anchor;
571     return result;
```

**Algorithm 28:** Seek Function for Target Key (Search Operation) (continued)

```
   // Looks for a given key in the tree (invoked by a search operation)
572 boolean SeekForSearch( opRecord, seekRecord )
573 begin
        // traverse the tree from top to down
574     result := Traverse( opRecord, seekRecord );
575     if not (result) then
            // check if the key has moved up in the path stored in the stack
576         result := ExamineStack( opRecord, seekRecord );
        // return the outcome
577     PopulateSeekRecord( seekRecord, state );
578     return result;
```

**Algorithm 29:** validate path

```
   // Determines if the path stored in the stack is still valid
   // Returns one of the following four values:
   // { SAFE, NOT_SAFE, FOUND, NOT_FOUND}
   // May backtrack along the path under certain situations
579 enum Outcome ValidatePath( opRecord, state )
580 begin
       // check if any of the anchor nodes in the stack has become inconsistent
          starting with the one immediately preceding the topmost node in the stack
581    {∗, ∗, critical} := GetFullEntry( state, ⊤ );
582    while true do
           // retrieve the node and its anchor from the stack
583        {node, ∗, anchor} := GetFullEntry( state, critical );
           // read the attributes of the node
584        marked := IsMarked( node );
585        key := GetKey( node );
586        if opRecord→key > key then
               // the anchor node is still consistent
587            if not (marked) then
                   // the access-path is still valid
588                RememberCritical( state, critical );
589                return SAFE;
590            else // need to check the previous anchor node
591                critical := anchor;
592        else if opRecord→key < key then
               // the anchor node is no longer consistent
593            if opRecord→type = DELETE then
                   // the target key did not exist continuously in the tree
594                return NOT_FOUND;
595            else // the path is not valid
596                RemoveUntil( state, critical );
597                return NOT_SAFE;
598        else // the two keys match
599            RemoveUntil( state, critical );
               // stop the traversal
600            return FOUND;
```

**Algorithm 30:** find a safe node

```
   // Backtracks along the path stored in the stack until a suitable restart point is
      found
601 enum Outcome FindASafeNode( opRecord, state )
602 begin
603     while true do
            // backtrack until an unmarked node
604         current := GetTop( state );
605         while IsMarked( current ) do
606             RemoveFromTop( state );
607             current := GetTop( state );

            // check if the algorithm needs a clean parent node
608         if NeedCleanParentNode( opRecord, current ) then
609             parent := GetSecondToTop( state );
610             if not (IsClean( parent )) then
                    // need to backtrack even further
611                 RemoveFromTop( state );
612                 continue;

            // check if the topmost node in the stack is a suitable restart point
613         result := ValidatePath( opRecord, state );
614         if result ≠ NOT_SAFE then return result ;
```

**Algorithm 31:** Seek Function for Target Key (Modify Operation)

```
   // Looks for a given key in the tree (invoked by insert/delete operation)
615 boolean SeekForModify( opRecord, seekRecord )
616 begin
617     state := opRecord→targetStack;
618     result := NOT_SAFE;
619     while result = NOT_SAFE do
            // backtrack to a suitable restart point in the path stored in the stack
620         result := FindASafeNode( opRecord, state );
621         if result ∈ {FOUND, NOT_FOUND} then break;
            // traverse the tree starting from the topmost node in the stack
622         current := GetTop( state );
623         while true do
624             key := GetKey( current );
625             which := opRecord→key < key ? LEFT : RIGHT;
                // read the next address to de-reference
626             reference := GetChild( current, which );
627             if (opRecord→key = key)  or IsNull( reference ) then
                    // either stop or backtrack & restart
628                 if opRecord→key < key then
                        // check if the path traversed is still valid
629                     result := ValidatePath( opRecord, state );
630                 else // determine what value to return
631                     result := FOUND;
632                     if opRecord→key ≠ key then
                            // remember the address read
633                         opRecord→injectionPoint :=
634                                 GetAddress( reference );
635                         result := NOT_FOUND;
                    // terminate the current traversal
636                 break;
                // traverse the next edge
637             address := GetAddress( reference );
638             restart := Move( current, address, which );
639             if restart then
                    // the algorithm wants to restart the traversal
640                 break;
                // push the node visited into the stack
641             AddToTop( state, address, which );
        // return the outcome
642     PopulateSeekRecord( seekRecord, opRecord );
643     return (result = FOUND ?  true: false);
```

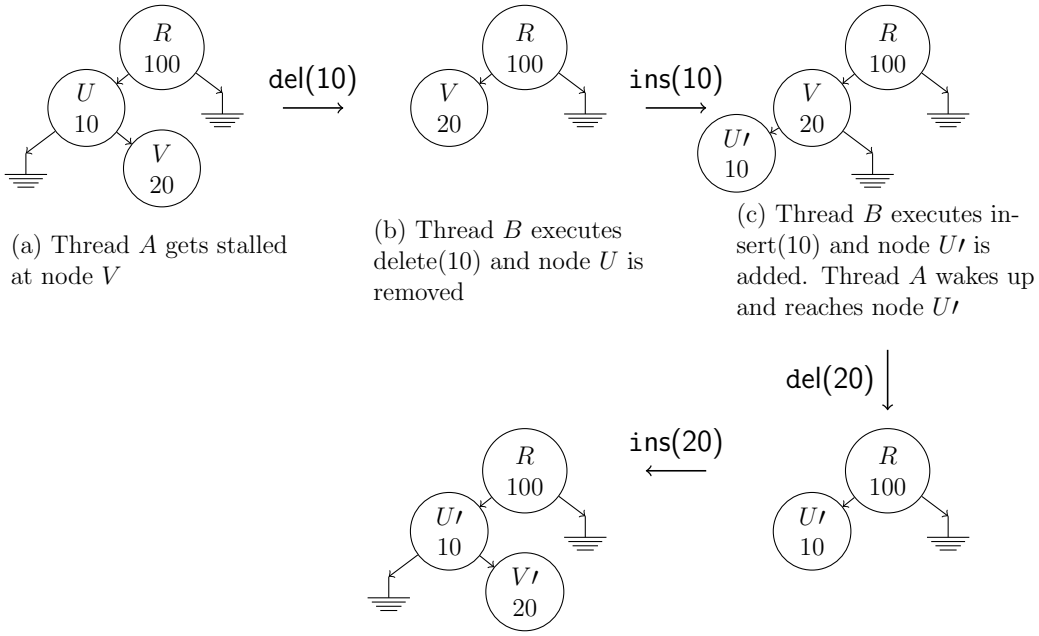
**Algorithm 32:** Seek Function for Successor Key

```
   // Looks for the next largest key with respect to a given key
644 boolean SeekForSuccessor( opRecord, seekRecord )
645 begin
      // the stack used in locating the successor key is initialized
646   state := opRecord→successorStack;
647   while true do
        // backtrack until either an unmarked node or the stack becomes empty
648     while (Size( state ) > 1) do
649       current := GetTop( state );
650       if not (IsMarked( current )) then break;
651       else RemoveFromTop( state ) ;

        // backtrack further if a clean parent is needed but the parent is not clean
652     if (Size( state ) ≥ 2) then
653       if NeedCleanParentNode( opRecord, current ) then
            // the parent node should be a clean node
654         parent := GetSecondToTop( state );
655         if not (IsClean( parent )) then
656           RemoveFromTop( state );
657           continue;

        // check if the successor key is still needed
658     reference := NeedSuccessorKey( opRecord );
659     if IsNull( reference ) then                    // successor key no longer required
660       return false;
661     current := GetTop( state );
662     if (Size( state ) = 1) then
          // visit the node pointed to by the reference returned by
             NeedSuccessorKey function
663       which := RIGHT;
664     else                // follow the left child node of the top node, if it exists
665       reference := GetChild( current, LEFT );
666       which := LEFT;
667     repeat                                                  // stop if reference is null
668       if IsNull( reference ) then break;
          // obtain the address of the node
669       address := GetAddress( reference );
          // traverse the edge
670       restart := Move( current, address, which );
671       if restart then              // the algorithm wants to restart the traversal
672         break;
          // push the node visited into the stack
673       AddToTop( state, address, which );
674       current := address;
          // determine the next node to be visited
675       reference := GetChild( current, LEFT );
676       which := LEFT;
677     until true;
      // return the outcome
678   PopulateSeekRecord( seekRecord, opRecord );
679   return true;
```

# CHAPTER 7

# WAIT FREE SEARCH

In this chapter we present two light-weight techniques to make search operations for concurrent binary search trees based on internal representation, *wait-free* with low additional overhead. Both of our techniques have the desirable feature that a search operation does not need to perform any write instructions on the share memory thereby minimizing the cache coherence traffic.




(a) Thread $A$ gets stalled at node $V$

(b) Thread $B$ executes delete(10) and node $U$ is removed

(c) Thread $B$ executes insert(10) and node $U\prime$ is added. Thread $A$ wakes up and reaches node $U\prime$

(d) Thread $B$ executes delete(20) and node $V$ is removed

(e) Thread $B$ executes insert(20) and a node $V\prime$ is added. Thread $A$ wakes up and reaches node $V\prime$

Figure 7.1. A scenario in which contains operation is not wait-free

The search operations in [2, 10, 20, 33, 34] are *not wait-free* even for a bounded key space. For example, in Figure 7.1 thread $A$ executes contains(15) and thread $B$ executes a series of

operations preventing the contains operation of thread $A$ to terminate. Note that the tree in (a) and (e) are same and this sequence of operations can occur ad infinitum. Hence the search operation is not wait-free.

In our first approach, we keep track of the count of modify operations. Here we do not make any new modifications to the tree node. In our second approach, we append a timestamp to the tree node but it has better complexity than the previous one.

### 7.1 No Modification to Tree Node

Due to the limited manner in which the tree can evolve in the concurrent algorithms described in [2, 10, 20, 33, 34], it is possible to design a light-weight wait-free algorithm for a search operation for all of the algorithms. The main property we use is that as long as a key is *continuously* present in the tree, its distance from the root of the tree is *monotonically non-increasing*. As a result, if a key is not found after visiting a "certain" number of nodes in the tree, then the traversal can stop and it is sufficient to examine the path traversed to check whether or not the key has moved up. In case the key is not continuously present in the tree, while a search operation is in progress, it is acceptable to return either of the outcomes—present or not present—to the application. In the first case, the search operation can be linearized after the insert operation that added the key to the tree. In the second case, it can be linearized after the delete operation that removed the key from the tree.

The main question is: "How do we *efficiently* determine the number of nodes to visit in the tree before stopping the downward traversal *without missing* the key that is continuously present in the tree?" To that end, we maintain two arrays with one entry for each process in the array, denoted by $IC$ and $DC$. Roughly speaking, entries $IC[i]$ and $DC[i]$ denote the number of insert and delete operations, respectively, process $P_i$ has performed so far. A process increments its insert counter before adding a key to the tree and its delete counter after removing a key from the tree. As a result, the insert (delete) counter at a process

is an upper (lower) bound on the number of keys that the process has added to (removed from) the tree. Before starting downward traversal for a search operation, a process first reads the delete counter values of all processes and then reads the insert counter values of all processes. It then computes an estimate for the number of nodes to visit as the sum of all the insert counter values minus the sum of all the delete counter values. We show that the estimate computed by a process is *safe* in the sense that a search operation will not miss a key continuously present in the tree while the operation is in progress.

To show that our algorithm works, we introduce some notation. Consider a time $t$ *after* an operation has read all the delete counter values but *before* its starts reading any of the insert counter values. Let $I_{t,i}$ ($D_{t,i}$) denote the *actual* number of keys added to (removed from) the tree by process $P_i$ at or before time $t$. Also, let $IC[i]$ ($DC[i]$) denote the value read for $IC[i]$ ($DC[i]$) by the operation. Note that, the way counters are maintained, $IC[i] \geq I_{t,i}$ and $DC[i] \leq D_{t,i}$. Also, let $S_t$ and $\Delta_t$ denote the *actual* size of the tree and the *actual* distance of the target key from the root of the tree, respectively, at time $t$. Clearly, we have:

$$S_t = \sum_{0 \leq i < p} I_{t,i} - \sum_{0 \leq i < p} D_{t,i} \quad \text{and} \quad \Delta_t \leq S_t$$

Thus we have:

$$\sum_{0 \leq i < p} IC[i] - \sum_{0 \leq i < p} DC[i] \;\geq\; \sum_{0 \leq i < p} I_{t,i} - \sum_{0 \leq i < p} D_{t,i} \;=\; S_t$$

In other words, the estimate computed by our algorithm is an upper bound on the actual distance of the key from the root of tree when the operation starts traversing the tree (which is monotonically non-increasing).

A pseudo-code of the algorithm is given in Pseudo-code 33. In the pseudo-code, $p$ denotes the number of processes. To amortize the overhead of reading $O(p)$ counters, an operation first visits $p$ nodes in the tree. If it does not find the key, then it reads the counter values and proceeds as described above. Thus $O(p)$ overhead is incurred only for "large" trees.

Some advantages of our approach are as follows. First, it works even if a key space is unbounded. Second, it does not require a search operation to perform any write instruction on shared memory. Third, it does not require a modify operation to perform any additional atomic instruction or helping (besides that performed by the original algorithm).

**Algorithm 33:** Seek Function for Target Key based on Estimating Tree Size

```
680 integer IC[p];
681 integer DC[p];
    // Traverses the tree starting from the root node but visits a limited number of
       nodes
682 boolean LimitedTraverse( opRecord, seekRecord, limit )
683 begin
        // similar to Traverse except that the while loop from lines 546-554 is executed
           at most limit times

    // A wait-free seek function for a search operation based on computing an
       upper-bound on tree size
684 boolean WFSeekForSearchBOSize( opRecord, seekRecord )
685 begin
686     result := LimitedTraverse( opRecord, seekRecord, p );
687     if not (result) then
688         D := DC[0] + DC[1] + ··· + DC[p − 1];
689         I := IC[0] + IC[1] + ··· + IC[p − 1];
690         S := I − D;
691         result := LimitedTraverse( opRecord, seekRecord, S );
692     if not (result) then
            // examine the stack
693         result := ExamineStack( opRecord, seekRecord );
        // return the outcome
694     PopulateSeekRecord( seekRecord, state );
695     return result;
```

## 7.2 With Modification to Tree Node

A disadvantage of the previous approach is that the time complexity of a search operation depends on the tree size. We now describe another approach to achieve wait-freedom for which the time complexity of a search operation depends on the tree height. This approach, however, requires modifying tree node to store a time-stamp of when the node was created.

**Algorithm 34:** Seek Function for Target Key based on Time-Stamps

```
696 integer labels[p];
    // Traverses the tree starting from the root node but stops if recently added key is
       found
697 boolean TimeStampTraverse( opRecord, seekRecord, labels )
698 begin
        // similar to Traverse except that the while loop from lines 546-554 is
           terminated as soon as a node with a ''recent'' time-stamp is encountered
        // specifically, the following lines are inserted between lines 546 & 547
699     ⟨pid, label⟩ := node→timeStamp;
700     if label > labels[pid] then break;
    // A wait-free seek function for a search operation based on estimating tree height
701 boolean WFSeekForSearchBOHeight( opRecord, seekRecord )
702 begin
703     result := LimitedTraverse( opRecord, seekRecord, p );
704     if not (result) then
705         copyOfLabels := labels;
706         result := TimeStampTraverse( opRecord,
                                         seekRecord,
                                         copyOfLabels );
707     if not (result) then
            // examine the stack
708         result := ExamineStack( opRecord, seekRecord );
        // return the outcome
709     PopulateSeekRecord( seekRecord, state );
710     return result;
```

It consists of the identifier of the process that created the node and the process-specific sequence number (which is incremented before the node is added to the tree). This time-stamp is copied if a node is *replaced* with a new node (in a complex delete operation) as in [33]. Before a search operation starts traversing the tree, it reads the current sequence number values of all processes. Let $labels[i]$ denote the value read for process $P_i$. The operation then stops the downward traversal of the tree once it encounters a node with time-stamp $\langle i, v\rangle$ such that $v > labels[i]$. Clearly, this node and its descendants were added to the tree after the operation read the sequence number value of process $P_i$. A pseudo-code of the algorithm is given in Pseudo-code 34.

# CHAPTER 8

# EXPERIMENTAL EVALUATION

We now describe the results of the comparative evaluation of different implementations of a concurrent BST using simulated workloads. This chapter is organized as follows. Performance evaluation of our lock-based algorithm is described in Section 8.2 followed by our lock-free algorithm described in Section 8.3. Performance evaluation of our local recovery technique is described in Section 8.4.

## 8.1 Experimental Setup

We conducted our experiments on a single large-memory node in stampede[1] cluster at TACC (Texas Advanced Computing Center). This node is a Dell PowerEdge R820 server with 4 Intel E5-4650 8-core processors (32 cores in total) and 1TB of DDR3 memory. Hyper-threading has been disabled on the node. It runs CentOS 6.3 operating system.

To better understand the scalability of our algorithms we also conducted experiments on a single Intel Xeon Phi SE10P Coprocessor[2] having 61 1.1 GHz cores with 4 hardware threads per core and 8GB of GDDR5 memory.

We used Intel C/C++ compiler (version 2013.2.146) with optimization flag set to O3. We used GNU Scientific Library to generate random numbers. We used Intel's *TBB Malloc* [35] as the dynamic memory allocator since it provided superior performance to C/C+ default allocator in a multi-threaded environment.

[1]https://www.tacc.utexas.edu/systems/stampede

[2]http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

To compare the performance of different implementations, we considered the following parameters:

1. **Relative Distribution of Operations:** We considered three different workload distributions: (a) *read-dominated:* 90% search, 9% insert and 1% delete, (b) *mixed:* 70% search, 20% insert and 10% delete, and (c) *write-dominated:* 0% search, 50% insert and 50% delete.
2. **Maximum Degree of Contention:** This depends on number of threads that can concurrently operate on the tree. On 32 core machine, we varied the number of threads from 1 to 32 in powers of two. On 61 core machine we varied the number of threads from 1 to 244 in multiples of 61.
3. **Maximum Tree Size:** This depends on the size of the key space. To get the peak throughput, we set the number of threads to be the value where the peak performance is achieved and we varied key space size from $2^{13}$ (8Ki) to $2^{24}$ (16Mi). To understand the scalability of the algorithms, we varied the number of threads and considered three different key ranges: 20,000 (20K), 200,000 (200K) and 2 million (2M) keys.

We compared the performance of different algorithms with respect to *system throughput*, given by the number of operations executed per unit time. In each run of the experiment, we ran each algorithm for 2 minutes, and calculated the total number of operations completed by the end of the run to determine the system throughput. The results were averaged over 5 runs. To capture only the steady state behaviour, we *pre-populated* the tree to 50% of its maximum size, prior to starting a simulation run. The beginning of each run consisted of a 1 second "warm-up" phase whose numbers were excluded in the computed statistics to avoid initial caching effects.

### 8.2 Lock based tree

In this section we evaluate CASTLE against three other implementations of a concurrent BST, namely those based on:

(i) the lock-free internal BST by Howley and Jones [20], denoted by LF-IBST,

(ii) the lock-free external BST by Natarajan and Mittal [31], denoted by LF-EBST, and

(iii) the RCU-based internal BST by Arbel and Attiya [2], denoted by CITRUS.

Note that CITRUS is a blocking implementation. The above three implementations were obtained from their respective authors. All implementations were written in C/C++. In our experiments, none of the implementations used garbage collection to reclaim memory. The experimental evaluation in [20, 31] showed that, in all cases, either LF-IBST or LF-EBST outperformed the concurrent BST implementation based on Ellen *et al.*'s lock-free algorithm in [11]. So we did not consider it in our experiments.

The results of our experiments are shown in Figure 8.1 and Figure 8.2. In Figure 8.1, the absolute value of the system throughput is plotted against the number of threads (varying from 1 to 32 in powers of 2). Here each row represents a specific workload (read-dominated, mixed or write-dominated) and each column represents a specific key space size (20K, 200K and 2M). In Figure 8.2, the value of the system throughput is plotted against the key space size. Here each column represents a range of key space sizes (small, medium and large) and each row represents a specific workload. As the peak performance for all the four algorithms (for all cases) occurred at 32 threads, we set the number of threads to 32 while varying the key space size from $2^{13}$ to $2^{24}$.

As both Figure 8.1 and Figure 8.2 show, for smaller key space sizes, LF-EBST achieves the best system throughput. This is not surprising since LF-EBST is optimized for high contention scenarios. For medium and large key space sizes, CASTLE achieves the best

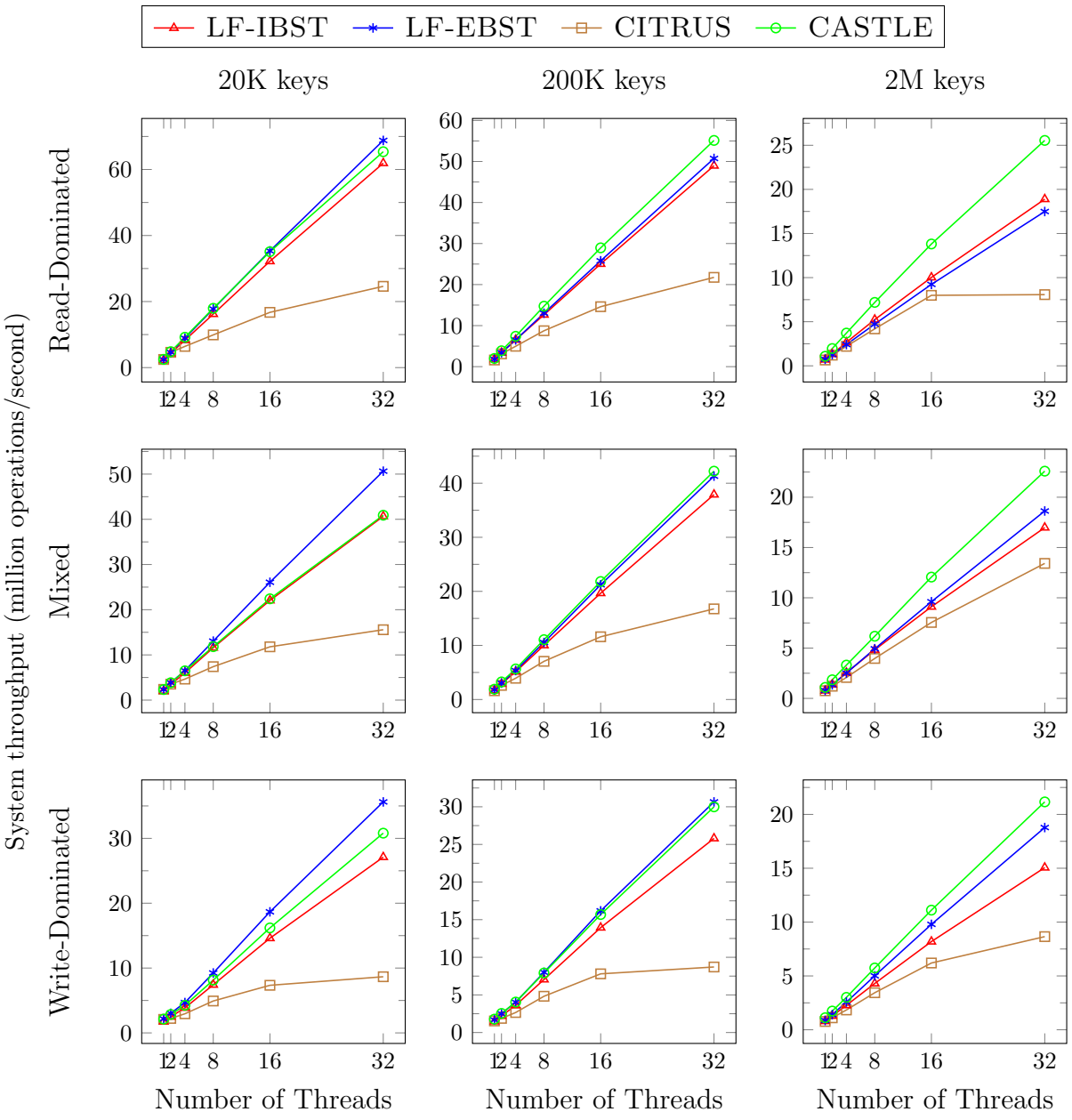



Figure 8.1. Comparison of system throughput of different algorithms. Each row represents a workload type and each column represents a key space size. Higher the throughput, better the performance of the algorithm.

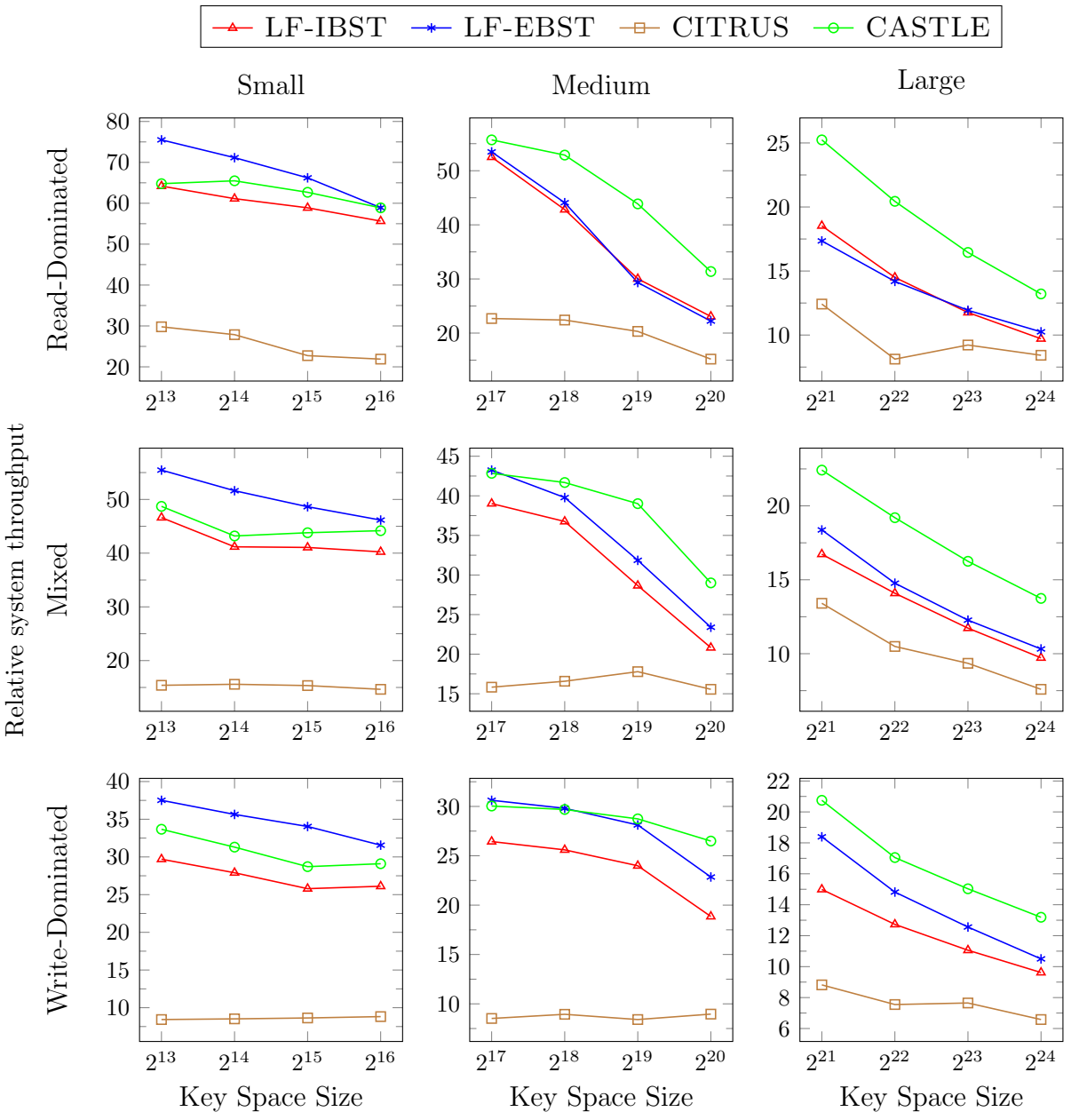



Figure 8.2. Comparison of system throughput of different algorithms at 32 threads. Each row represents a workload type. Each column represents a range of key space range. Higher the ratio, better the performance of the algorithm.

system throughput for all three workload types in almost all the cases (except when the workload is write-dominated and the key space size is in the lower half of the medium range). The maximum gap between CASTLE and the next best performer is around 46% which occurs at 500K key space size, read-dominated workload and 32 threads.

We believe that some of the reasons for the better performance of CASTLE over the other three concurrent algorithms, especially when the contention is relatively low, are as follows. First, as explained in [31], operating at edge-level rather than at node-level reduces the contention among operations. Second, using a CAS instruction for locking an edge also validates that the edge has not undergone any change since it was last observed. Third, locking edges as late as possible minimizes the blocking effect of the locks.

Table 8.1. Comparison of different concurrent algorithms in the absence of contention.

| **Algorithm** | **Number of Objects Allocated** | | **Number of Synchronization Primitives Executed** | |
|---|---|---|---|---|
| | **Insert** | **Delete** | **Insert** | **Delete** |
| LF-IBST | 2 | 1 | 3 | simple: 4<br>complex: 9 |
| LF-EBST | 2 | 0 | 1 | 3 |
| CASTLE | 1 | 0 | 1 | simple: 3<br>complex: 4 |

Table 8.1 shows a comparison of LF-IBST, LF-EBST and CASTLE with respect to the number of objects allocated dynamically and the number of synchronization primitives executed per modify operation in the absence of contention. We omitted CITRUS in this comparison since it based on a different framework. As Table 8.1 shows, our algorithm allocates fewer objects dynamically than the two lock-free algorithms (one for insert operations and none for a delete operations). Further, again as Table 8.1 shows, our algorithm executes much fewer synchronization primitives than LF-IBST. It executes the same number of synchronization primitives as LF-EBST for insert and simple delete operations and only one

more for complex delete operations. This is important since a synchronization primitive is usually much more expensive to execute than a simple read or write instruction. Finally, we observed in our experiments that CASTLE had a smaller memory footprint than all the three implementations (by a factor of two or more) since it uses internal representation of a search tree and allocates fewer objects dynamically. As a result, it was likely able to benefit from caching to a larger degree than the other algorithms.

In our experiments, we observed that, for key space sizes larger than 10K, the likelihood of an operation restarting was extremely low (less than 0.1%) even for a write-dominated workload. This implies that, in at least 99.9% of the cases, an operation was able to complete without encountering any conflicts. Thus, for key space sizes larger than 10K, we expect CASTLE to outperform the implementation based on the lock-free algorithm described in [12], which is basically derived from the one in [11].

### 8.3 Lock free tree

In this section we evaluate ELFtree against three other implementations of a concurrent BST, namely those based on:

(i) the lock-free internal BST by Howley and Jones [20], denoted by LF-IBST,

(ii) the lock-free external BST by Natarajan and Mittal [31], denoted by LF-EBST, and

(iii) the RCU-based internal BST by Arbel and Attiya [2], denoted by CITRUS.

The results of our experiments are shown in Figure 8.3 and Figure 8.4. In Figure 8.3, each row represents a specific workload (read-dominated, mixed or write-dominated) and each column represents a specific key space size; *small* (8Ki to 64Ki), *medium* (128Ki to 1Mi) and *large* (2Mi to 16Mi). Figure 8.4 shows the scaling with respect to the number of threads for key space size of $2^{19}$ (512Ki). We do not show the numbers for CITRUS in

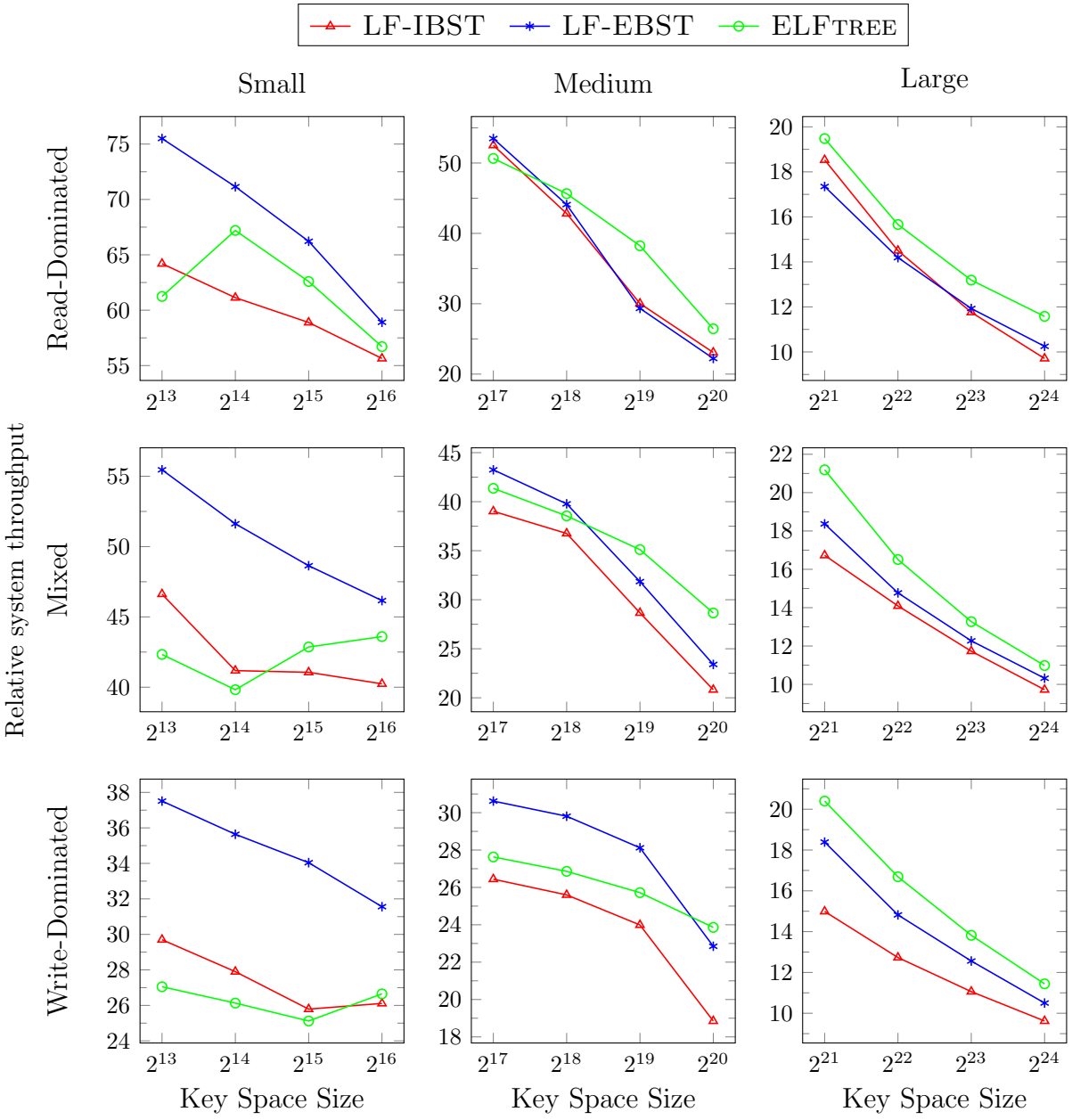



Figure 8.3. Comparison of system throughput of different algorithms at 32 threads. Each row represents a workload type. Each column represents a range of key space range. Higher the ratio, better the performance of the algorithm.

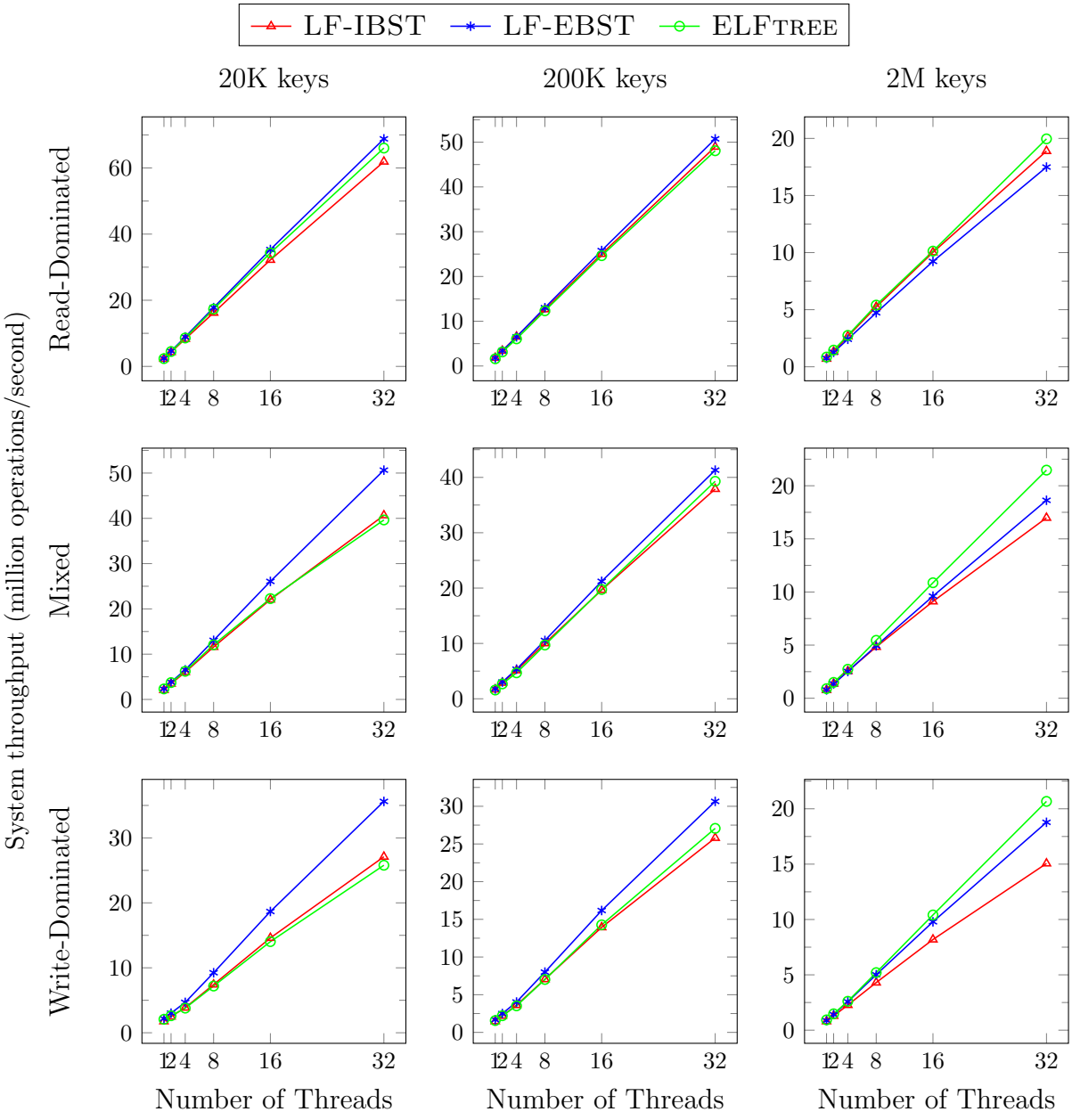



Figure 8.4. Comparison of system throughput of different algorithms. Each row represents a workload type and each column represents a key space size. Higher the throughput, better the performance of the algorithm.

the graphs as it had the worst performance among all implementations (slower by a factor of four in some cases). This is not surprising as CITRUS is optimized for read operations (*e.g.*, 98% reads & 2% updates) [2].

As the graphs show, ELFTREE achieved nearly same or higher throughput than the other two implementations for medium and large key space sizes (except for medium key space size with write-dominated workload). Specifically, at 32 threads and for a read-dominated workload, ELFTREE had 27% and 15% higher throughput than the next best performer for key space sizes of 512Ki and 1Mi, respectively. Also, at 32 threads and for a mixed workload, ELFTREE had 22% and 15% higher throughput than the next best performer for key space sizes of 1Mi and 2Mi, respectively. Overall, ELFTREE outperformed the next best implementation by as much as 27%; it outperformed LF-IBST by as much as 38% (mixed) and LF-EBST by as much as 30%(read-dominated). For large key space sizes, the overhead of traversing the tree appears to dominate the overhead of actually modifying the operation's window, and the gap between various implementations becomes smaller.

Table 8.2. Comparison of different lock-free algorithms in the absence of contention.

| **Algorithm** | **Number of Objects Allocated** | | **Number of Atomic Instructions Executed** | |
|---|---|---|---|---|
| | **Insert** | **Delete** | **Insert** | **Delete** |
| LF-IBST | 2 | simple: 1<br>complex: 1 | 3 | simple: 4<br>complex: 9 |
| LF-EBST | 2 | 0 | 1 | 3 |
| ELFTREE | 1 | simple: 0<br>complex: 1 | 1 | simple: 4<br>complex: 7 |

There are several reasons why ELFTREE outperformed the other two implementations in many cases. First, as Table 8.2 shows, our algorithm allocates fewer objects than the two other algorithms on average considering the fact that the fraction of insert operations will generally be larger than the fraction of delete operations in any realistic workload. Further,

we observed in our experiments that the number of simple delete operations outnumbered the number of complex delete operations by two to one, and our algorithm does not allocate any object for a simple delete operation. Second, again as Table 8.2 shows, our algorithm executes the same number of atomic instructions as in [31] for insert operations; and, in all the cases, executes same or fewer atomic instructions than in [20]. This is important since an atomic instruction is more expensive to execute than a simple read or write instruction. Third, we observed in our experiments that ELFtree had a smaller memory footprint than the other two implementations (by almost a factor of two) since it uses internal representation and allocates fewer objects. As a result, it was likely able to benefit from caching to a larger degree than LF-IBST and LF-EBST.

## 8.4 Impact of local recovery

In this section we evaluate our local recovery technique.

To show that our local recovery algorithm is sufficiently general, we implemented it for three different concurrent internal BSTs, namely those based on: (i) the lock-free BST by Howley and Jones [20], denoted by LF-IBST, (ii) the lock-based BST by Ramachandran and Mittal [34], denoted by CASTLE and (iii) the RCU (Read-Copy-Update) lock-based BST by Arbel and Attiya [2], denoted by CITRUS. These implementations were chosen so that we covered both lock-free and lock-based approaches. We choose two lock-based implementations as one is based on locking edges [34] and the other is based on locking nodes using RCU framework [2].

### Experimental Setup

As local recovery is useful for high contention cases, we conducted our experiments on a single Intel Xeon Phi SE10P Coprocessor [3] having 61 1.1 GHz cores with 4 hardware threads

[3]http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

Table 8.3. Effect of local recovery on system throughput. Positive number indicates a gain while a negative number indicates a drop.

| **Algorithm** | **Uniform (max%, min%)** | **Zipfian (max%, min%)** |
|---|---|---|
| **CASTLE** | (14, -15) | (31, -17) |
| **CITRUS** | (-1, -58) | (-2, -8) |
| **LF-IBST** | (-4, -29) | (70, -36) |

per core and 8GB of GDDR5 memory. With this machine we were able to experiment with up to 244 threads.

### Key distribution

Usually uniform key distribution (where all keys have same frequency of occurrence) have been used to evaluate concurrent BSTs. But in many of the real world workloads, keys have skewed distribution [7] where some keys are more popular than others. Zipfian distribution, a type of power-law distribution simulates this behaviour [5, 13, 16]. It is characterized by a parameter $\alpha$ which usually lies between 0.5 and 1 [1, 5]. In our experiments we used both uniform and Zipfian distributions to evaluate the local recovery algorithm.

Table 8.3 summarizes the performance gap (with respect to system throughput) between the base algorithm and its extension using local recovery for uniform and Zipfian distributions.

To better understand the effect of local recovery, we also measure several metrics defined in Table 8.4. Figures 8.5-8.6 show the impact of local recovery on throughput and various metrics for uniform distribution. We see that the performance gain is marginal and, in many cases, is actually slightly worse due to the overhead of stack maintenance. This is not surprising because, for small trees, even though contention is higher, seek time is small to begin with and any benefit of local recovery is nullified by additional overhead of stack maintenance. For larger trees, even though seek time is larger, contention is low as key accesses are spread evenly.

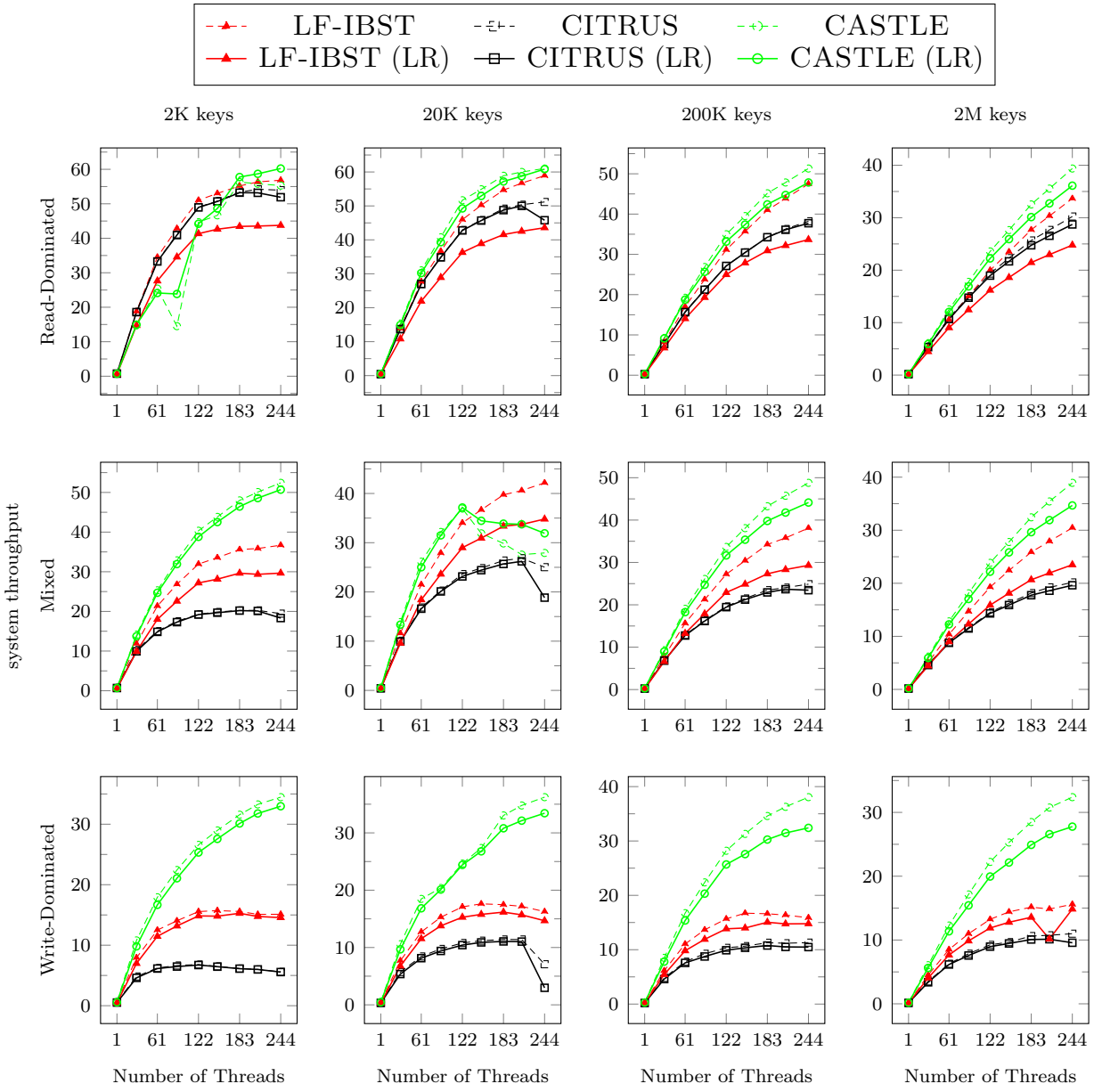



Figure 8.5. Comparison of throughput of different concurrent BST implementations with (solid lines) and without (dotted lines) local recovery for uniform distribution. Each row represents a workload type. Each column represents a key space range. Higher is better.

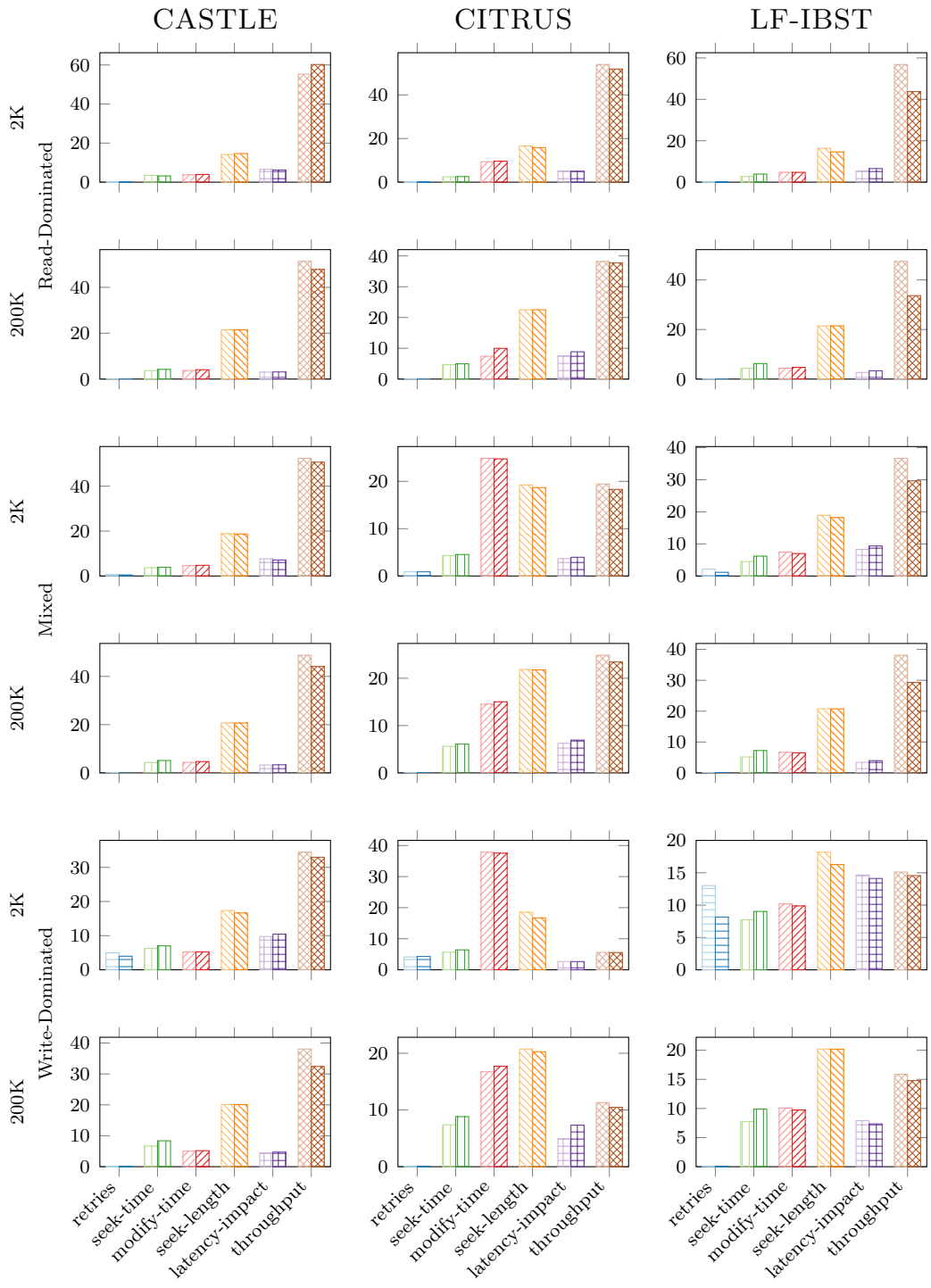



Figure 8.6. Impact of local recovery on various metrics for two key space ranges (2K and 200K) with uniform key distribution.

Table 8.4. Performance metrics used to evaluate the effect of local recovery.

| **Metric** | **Description** |
|---|---|
| *retries* | fraction of restarts for an operation |
| *seek-time* | total time an operation spends on tree traversal including all restarts as well as stack processing time (in $\mu$s) |
| *modify-time* | total time an operation spends on modify phase (in $\mu$s) |
| *seek-length* | number of nodes traversed in the tree by an operation |
| *latency-impact* | a rough approximation of the amount of clock cycles devoted to each L1 cache miss (a measure of cache performance) [22] |

Figures 8.7-8.8 show the impact of local recovery on throughput and various metrics for Zipfian distribution. In general, Zipfian distribution causes more contention than uniform distribution. So even for small trees for which seek times are smaller, we still see performance gains for mixed and write-dominated workloads. As the graphs show, we see a consistent increase in throughput (up to 70%) using local recovery as we move from low contention scenarios (read-dominated workload and large tree size) to high contention scenarios (write-dominated workload and small tree size). Specifically, we see up to 70% and 31% improvements for LF-IBST and CASTLE respectively. Except CITRUS, the other two implementations scale fairly well up to 244 threads.

As shown in Figure 8.8, for LF-IBST, all the metrics except *modify-time* show improvement. A modify operation in this algorithm helps any pending operation along its traversal path leading to higher fraction of restarts shown by the metric *retries*. Retries and hence *seek-length* and *seek-time* are significantly reduced due to local recovery.

For CASTLE, for smaller trees, we see that all the metrics improved and the throughput increases up to 31%. For CITRUS, there was no improvement due to local recovery across all three workloads as this algorithm is optimized for low contention scenarios and does not scale well for high contention scenarios. Most of its modify operation's time is spent on modify phase than in seek phase. The main reason being a complex delete operation has to wait till all previous readers have completed their traversals.

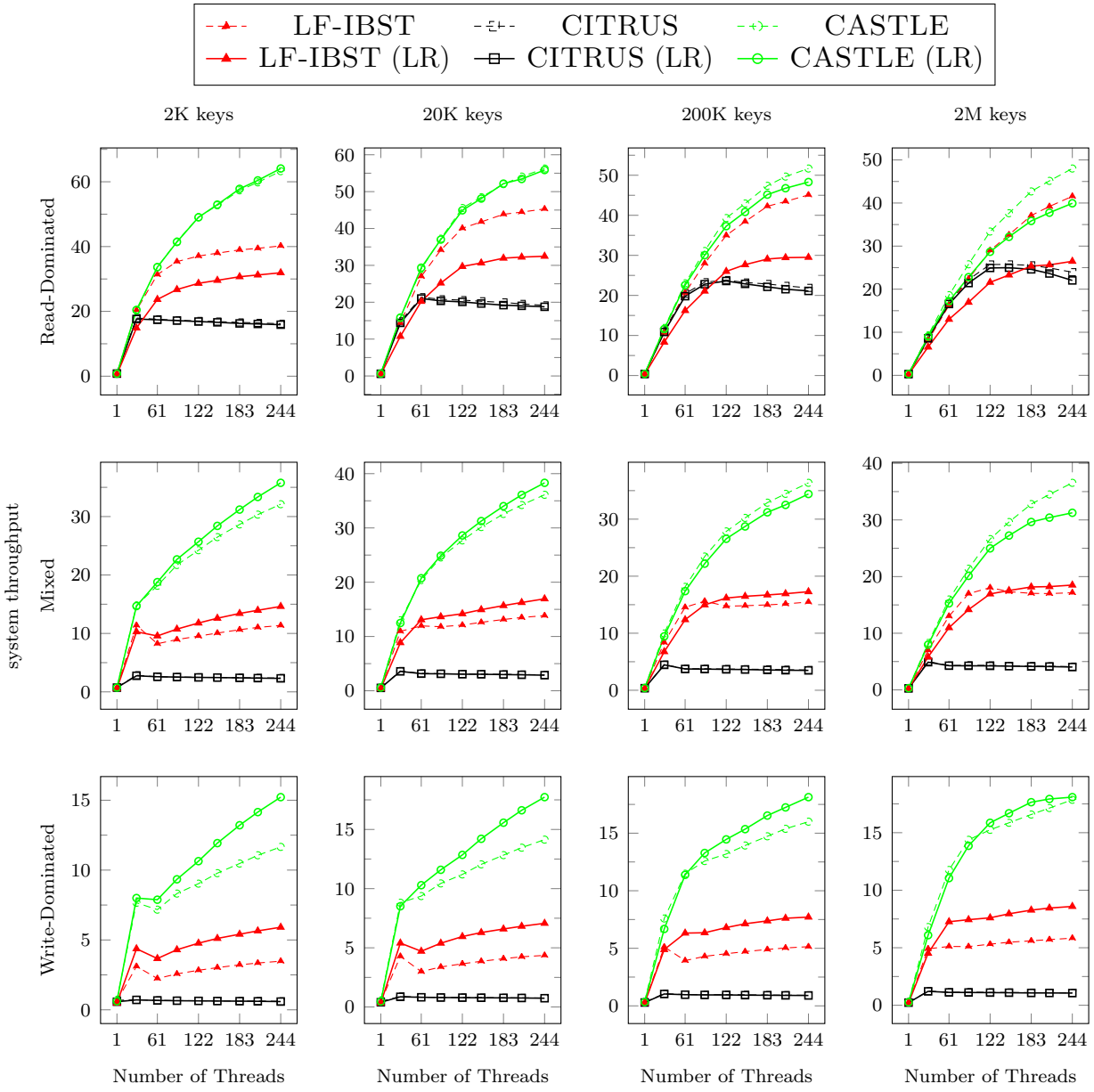



Figure 8.7. Comparison of throughput of different concurrent BST implementations with (solid lines) and without (dotted lines) local recovery for zipfian distribution with $\alpha$=1. Each row represents a workload type. Each column represents a key space range. Higher is better.

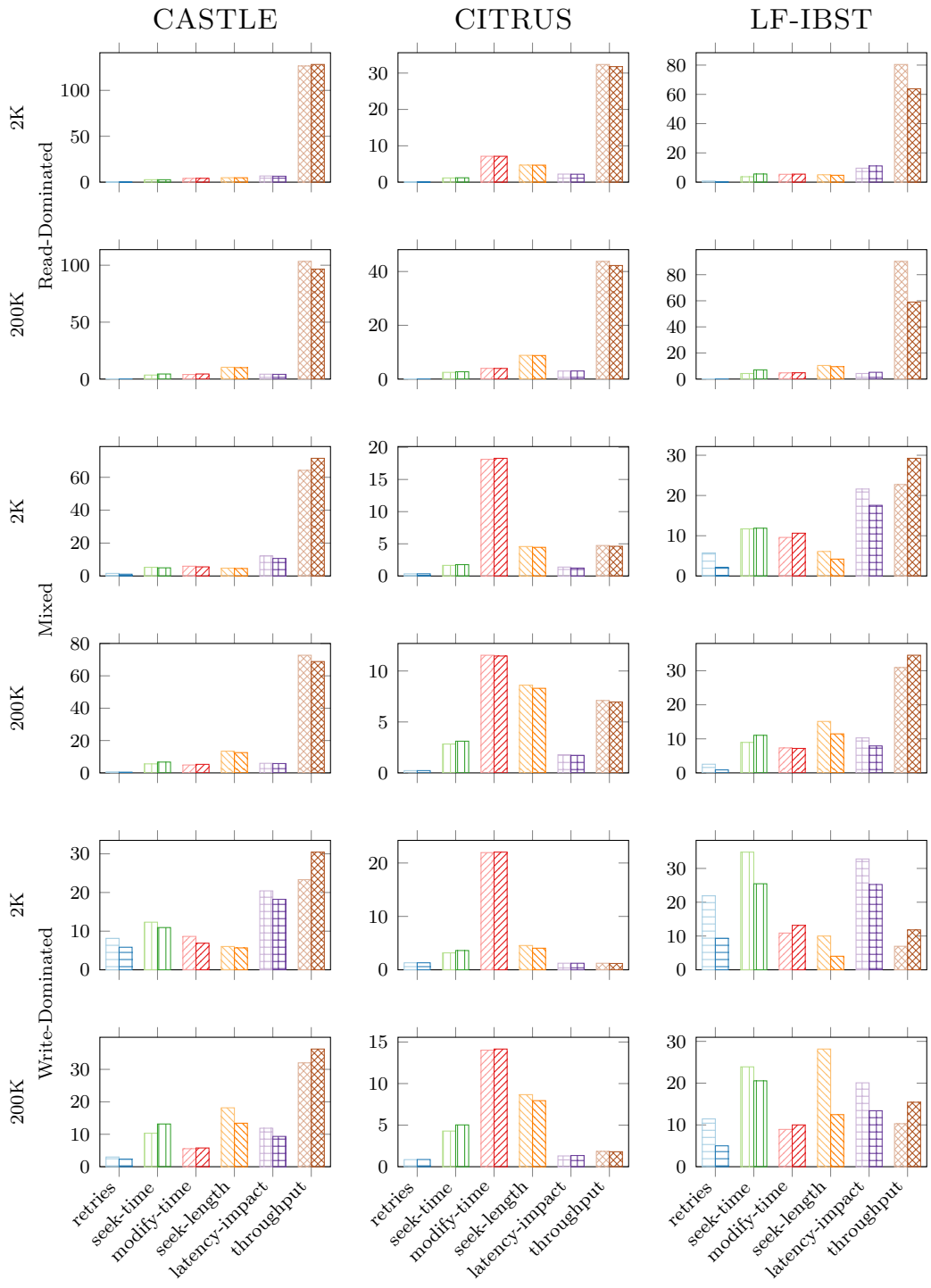



Figure 8.8. Impact of local recovery on various metrics for two key space ranges (2K and 200K) with zipf key distribution. Each column represents a concurrent BST implementation. To show all the metrics on the same plot, some of the metrics were scaled as follows: *retries* by $\frac{1}{8}$, *seek-length* by $\frac{1}{3}$ and *throughput* by 2. For CITRUS, *modify-time* was further scaled-down by a factor 20 to make other metrics visible.

# CHAPTER 9

# CONCLUSION

In this dissertation we presented a blocking and a non-blocking algorithm for concurrent manipulation of a binary search tree in an asynchronous shared memory system that supports search, insert and delete operations.

Our *lock-based* algorithm is very simple and looks almost identical to a sequential algorithm. In contrast to other lock-based algorithms, it locks edges rather than nodes. This minimizes the contention window of an operation and improves the system throughput. Since the locks are based on edges and as we steal bits from the children edges, the tree node structure is identical to a sequential tree node. This keeps the memory foot print low and reduces the impact of *memory-allocation*. A desirable feature of this algorithm is that its search and insert operations are lock-free; they do not obtain any locks. As indicated by our experiments, our algorithm has the best performance—compared to other concurrent algorithms for a binary search—when the contention is relatively low. Specifically, it achieved the best performance for medium-sized and larger trees with mixed workloads and read-dominated workloads.

Our *lock-free* algorithm combined ideas from two existing lock-free algorithms and is especially *optimized for the conflict-free scenario*. Specifically, when compared to modify operations exiting internal binary search trees, its modify operations (a) have a smaller contention window, (b) allocate fewer objects, (c) execute fewer atomic instructions, and (d) have a smaller memory footprint. Our experiments indicated that our new lock-free algorithm outperforms other lock-free algorithms in most cases.

We also presented a new approach to recover from such failures more efficiently in a concurrent binary search tree based on internal representation using *local recovery* by restarting

the traversal from the "middle" of the tree in order to locate an operation's window. Our approach is sufficiently general in the sense that we were able to apply it to a variety of concurrent binary search trees based on both blocking and non-blocking approaches.

We also presented a framework to allow a concurrent algorithm for maintaining an internal BST to recover locally when traversing the tree to locate a key. Our framework is sufficiently general that we were able to apply it to a variety of concurrent binary search trees based on both blocking and non-blocking approaches. we showed by experiments that our local recovery framework improved the performance of concurrent BST algorithms under non-uniform key distribution (*e.g.*, Zipfian) for many different workloads.

As a future work, we would like to analyze our local recovery algorithm (and possibly refine it if needed) so that, when applied to a *non-blocking* BST, it yields a concurrent BST whose operations have provably low amortized time complexity. Also, we would like to analyze the effect of local recovery for other standard non-uniform distributions like normal and Poisson on real workloads.

We also would like to extend the ideas used in our algorithms and our local recovery technique to other data structures. A simple extension would be to apply them to $k$-ary search trees and then extend it further to $B$-trees. We also plan to explore other data structures like Bloom-filters which are commonly used in big-data applications.

## REFERENCES


[1] Adamic, L. A. and B. A. Huberman (2002). Zipfs Law and the Internet. *Glottometrics 3*(1), 143–150.

[2] Arbel, M. and H. Attiya (2014, July). Concurrent Updates with RCU: Search Tree as an Example. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 196–205.

[3] Bender, M. A., J. T. Fineman, S. Gilbert, and B. C. Kuszmaul (2005, July). Concurrent Cache-Oblivious B-Trees. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 228–237.

[4] Braginsky, A. and E. Petrank (2012). A Lock-Free B+tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 58–67.

[5] Breslau, L., P. Cao, L. Fan, G. Phillips, and S. Shenker (1999, March). Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of the 18th IEEE Conference on Computer Communications (INFOCOM)*, pp. 126–134.

[6] Chatterjee, B., N. N. Dang, and P. Tsigas (2014). Efficient Lock-Free Binary Search Trees. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 321–331.

[7] Clauset, A., C. R. Shalizi, and M. E. J. Newman (2009). Power-Law Distributions in Empirical Data. *SIAM Review*, 661–703.

[8] Comer, D. (1979, June). Ubiquitous b-tree. *ACM Comput. Surv. 11*(2), 121–137.

[9] Cormen, T. H., C. E. Leiserson, and R. L. Rivest (1991). *Introduction to Algorithms*. The MIT Press.

[10] Drachsler, D., M. Vechev, and E. Yahav (2014, February). Practical Concurrent Binary Search Trees via Logical Ordering. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 343–356.

[11] Ellen, F., P. Fataourou, E. Ruppert, and F. van Breugel (2010, July). Non-Blocking Binary Search Trees. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 131–140.

[12] Ellen, F., P. Fatourou, J. Helga, and E. Ruppert (2014). The Amortized Complexity of Non-Blocking Binary Search Trees. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 332–340.

[13] Faloutsos, C. and H. V. Jagadish (1992, August). On B-tree Indices for Skewed Distributions. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB)*, pp. 364–373.

[14] Finkel, R. A. and J. L. Bentley (1974). Quad trees a data structure for retrieval on composite keys. *Acta Informatica 4*(1), 1–9.

[15] Fomitchev, M. and E. Ruppert (2004, July). Lock-Free Linked Lists and Skiplists. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 50–59.

[16] Gray, J., P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger (1994, May). Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the 23rd ACM SIGMOD International Conference on Managament of Data*, pp. 243–252.

[17] Heller, S., M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit (2005). A Lazy Concurrent List-Based Set Algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS)*, Pisa, Italy, pp. 3–16.

[18] Herlihy, M. and N. Shavit (2012). *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann.

[19] Herlihy, M. and J. M. Wing (1990, July). Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*(3), 463–492.

[20] Howley, S. V. and J. Jones (2012, June). A Non-Blocking Internal Binary Search Tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 161–171.

[21] Jackins, C. L. and S. L. Tanimoto (1980). Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing 14*(3), 249 – 270.

[22] Jeffers, J. and J. Reinders (2013). *Intel Xeon Phi Coprocessor High-Performance Programming*. Morgan Kaufmann.

[23] Lamport, L. (1979, Sept). How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on C-28*(9), 690–691.

[24] Lea, D. (2003). Java Community Process, JSR 166, Concurrent Utilities. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`.

[25] Lev, Y., M. Herlihy, V. Luchangco, and N. Shavit (2007, June). A Simple Optimistic Skiplist Algorithm. In *Proceedings of the 14th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, Castiglioncello, Italy, pp. 124–138.

[26] Michael, M. M. (2002). High Performance Dynamic Lock-Free Hash Tables and List-based Sets. In *Proceedings of the 14th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 73–82.

[27] Michael, M. M. (2004). Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 15*(6), 491–504.

[28] Michael, M. M. and M. L. Scott (1996). Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 267–275.

[29] Moir, M. and N. Shavit (2007). Concurrent data structures. *Handbook of Data Structures and Applications*, 47–14.

[30] Natarajan, A. and N. Mittal (2013, October). Brief Announcement: A Concurrent Lock-Free Red-Black Tree. In *Proceedings of the 27th Symposium on Distributed Computing (DISC)*, Jerusalem, Israel.

[31] Natarajan, A. and N. Mittal (2014, February). Fast Concurrent Lock-Free Binary Search Trees. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 317–328.

[32] Natarajan, A., L. H. Savoie, and N. Mittal (2013, November). Concurrent Wait-Free Red-Black Trees. In *Proceedings of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Osaka, Japan, pp. 45–60.

[33] Ramachandran, A. and N. Mittal (2015a, January). A Fast Lock-Free Internal Binary Search Tree. In *Proceedings of the 16th International Conference on Distributed Computing and Networking (ICDCN)*.

[34] Ramachandran, A. and N. Mittal (2015b, February). CASTLE: Fast Concurrent Internal Binary Search Tree using Edge-Based Locking. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 281–282.

[35] Reinders, J. (2007). *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, Inc.

# VITA

Arunmoezhi Ramachandran was born in Madurai, India and was brought up in Rajapalayam, India. He completed his schooling in 2003. He pursued his Bachelor's degree in Electronics and Communication at College of Engineering Guindy, Chennai. Then he worked at Infosys Technologies Ltd, Chennai as a software engineer from 2007 to 2010. Meanwhile he also pursued his Masters in Software Engineering at Birla Institute of Technology and Science, Pilani. Upon graduating with a Masters degree in 2011, he joined the PhD program in Computer Science at The University of Texas, Dallas, USA.