12-1-2017

Lock-free Self-adjusting Binary Search Tree

Sachiko Sueki
University of Nevada, Las Vegas, sachiko.s@gmail.com

LOCK-FREE SELF-ADJUSTING BINARY SEARCH TREE

By

Sachiko Sueki

Bachelor of Science in Engineering – Civil and Environmental Engineering
Nihon University
1998

A thesis submitted in partial fulfillment
Of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2017

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

November 14, 2017

This thesis prepared by

Sachiko Sueki

entitled

Lock-Free Self-Adjusting Binary Search Tree

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Ajoy Datta, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

John Minor, Ph.D.
Examination Committee Member

Wolfgang Bein, Ph.D.
Examination Committee Member

Kumud Acharya, Ph.D.
Examination Committee Member

Emma Regentova, Ph.D.
Graduate College Faculty Representative

Abstract

A binary search tree (BST) is a fundamental data structure for maintaining data in a way to allow fast search. As multi-core processors are widely used nowadays, such data structures require the ability to handle concurrent accesses to exploit the concurrent hardware. There are several algorithms on lock-based as well as lock-free BST with and without self-balancing. On the other hand, only a few reports can be found on lock-based self-adjusting BST. To the best of our knowledge, there are no algorithms for lock-free self-adjusting BST. Lock-free guarantees overall progress even if some processes fail, whereas lock-based algorithms fail to progress in the case of a faulty process. In this study, a lock-free self-adjusting binary search tree is proposed using compare-and-swap (CAS) operations. The algorithm is based on lazy splaying which moves frequently accessed items near the root in a contention friendly manner. The algorithm will include search, delete, insert, and restructuring operations.

Acknowledgements

I would like to express my sincere appreciation to my adviser, Dr. Ajoy Datta for his guidance and support throughout my study. I would also like to thank committee members, Dr. John Minor, Dr. Wolfgang Bein, and Dr. Emma Regentova for their support and time. Special thanks to Dr. Kumud Acharya at Desert Research Institute for his encouragement and financial support.

I would like to acknowledge the department chair, Dr. Laxmi Gewali for assisting me with the admission to the computer science program. Thanks to the department staff, Mr. Mario Martin for his kind and prompt help every time I reached out to him.

I would also like to thank Rashmi Niyolia for her friendship and valuable discussions.

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Computers are used for many different purposes such as simulation, data monitoring, image processing, entertainment, record maintenance, controlling transactions etc. Furthermore, computers can be found everywhere such as cell phones, ATM machines, game consoles, cars and all other modern devices. A significant improvement of computational power made computer ubiquitous and is largely due to the increased density of transistors. Since the invention of an integrated circuits in 1958, the number of transistors on a single chip was doubling every year (Stalling, 2015) improving a clock speed of a computer. However, the pace of shrinking transistors is slowly decreasing, making manufactures consider a new technique to enhance computer performance such as a multicore processor instead of a single-core processor. In 2005, Intel introduced the first dual-core processors soon followed by AMD's dual-core processors (Mueller, 2006). These days, a basic computer uses dual-core processors, and in some case processors with more than 20 cores are also available in the market. However, unlike a single-core processor, a multicore processor causes challenges on developing software. Multicore system exploits parallelism. Multicores concurrently execute several threads in different cores and cooperatively solve a computational problem by communicating and synchronizing via a shared memory. However, modern computer systems can unexpectedly halt or delay their activities by interrupts, preemption, cache misses, failures and other events with different time scale (Herlihy and Shavit, 2008). In a single-core processor, the processor runs multiple processes concurrently by assigning a time-slice to each thread of processes resulting in

sequential execution of each process. On the other hand, in a multi-core processor, multiple threads are run simultaneously in each core. This results in interleaving steps of concurrently running threads, which allows running parallel processes in any order. Therefore, programming for multicore processors requires handling an asynchronous concurrent environment.

The fundamental building block of programming and algorithms is data structures. In a multicore system, concurrently running threads access the data structure in a shared memory in any order, which can result in different and possibly unexpected outcomes. Therefore, data structures require the ability to handle concurrent access. Furthermore, small improvements in concurrent data structures can provide a large improvement in overall performance because of several threads accessing at the same time. Therefore, increased efficiency and scalability of concurrent data structures are crucial for better performance of computation.

A simple way to build concurrent data structures for multicore processors from sequential data structures is to use a mutual exclusion lock. By locking a data structure whenever a thread is accessing the data structure, the other threads must wait until the thread accessing the data structure releases the lock. However, use of locks introduces high contentions and a sequential bottleneck affecting performance of computation. This problem can be eased by fine-grained locking which uses multiple locks to protect different parts of the data structure instead of blocking the entire data structure. However, lock-based implementation in asynchronous environments poses other issues. For instance, if a thread holding a lock fails, the entire program fails. Another example of problems is that the rest of the threads must wait unnecessarily longer if the thread holding a lock is halted.

Unlike lock-based implementation, a non-blocking implementation provides a better progress condition. Even the weakest progress condition of a non-blocking implementation

guarantees progress of any threads that eventually execute in isolation. Therefore, non-blocking is desirable even though it is harder to implement correctly.

There are several lock-based concurrent data structures such as queues, stacks, skiplists and binary search trees (BST) (Herlihy and Shavit, 2008). These data structures also have non-blocking implementations as reported in Herlihy and Shavit (2008), Ellen et al. (2010) and several other papers. Among these data structures, BST is one of the fundamental data structures, which allows faster lookup, addition and deletion of items by maintaining an ordered map. There are popular variations of BST such as AVL tree, red-black tree and splay tree. AVL tree and red-black tree are self-balancing BST which maintain balanced tree height. Splay tree is a self-adjusting BST which moves frequently accessed items toward the root of a tree. A self-balancing concurrent BST has been implemented by both a lock-based algorithm (Bronson et al., 2010) and a lock-free algorithm (Brown et al., 2014). On the other hand, only lock-based algorithms can be found for self-balancing concurrent BST such as reported by Afek et al. (n.d.) and Afek et al. (2014).

1.2 Objective

Our objective in this study is to present a lock-free self-adjusting BST algorithm including basic BST operations, search, insert and remove. In order to reduce the sequential bottleneck, a lazy splay technique is selected as a self-adjusting BST. Furthermore, a contention friendly algorithm which runs restructuring operations in the background to avoid contention during traversal of a tree is considered to modify the algorithm of LST. Lock-free is designed using an atomic operation, the single-word compare-and-swap (CAS), which is compatible with

a wide range of systems, and flagging in a pointer, which helps to complete an ongoing operation by other threads. Correctness of the algorithm is also discussed in this study.

1.3 Outline

Basic background concepts and terminology required for this study are presented in Chapter 2 including characteristics of concurrent data structures and their implementations. Correctness conditions of concurrent data structures are also given in Chapter 2.

Related work is reviewed in Chapter 3. Mainly, algorithms of BST and splay trees which are a variation of BST are discussed in this chapter.

In Chapter 4, an algorithm for lock-free self-adjusting BST is presented. An overview and detailed description, including pseudo-code of the algorithm, are found in this chapter.

Correctness of our algorithm is discussed in terms of non-blocking and linearization in Chapter 5.

Finally, the work is summarized and possible future work to extend this study is presented in Chapter 6.

Chapter 2

Basic Background Concepts

This chapter explains some of the basic background concepts related to the study. Most of the information written in this chapter are summarized from books dealing with multiprocessors (Moir and Shavit, 2004; Herlihy and Shavit, 2008; Pacheco, 2011; Stallings, 2015).

2.1 Multicore Processors

A multicore processor is an integrated circuit which has multiple processors on a single chip. A multicore processor may provide increased performance without increasing clock speed. However, improvement in performance depends on the effective use of the multicore by software. If parts of a program do not require any coordination, it can be easily executed in parallel and utilize multicore. Whereas the parts that require coordination affect performance and must be effectively programmed to run on a multicore processor.

2.2 Shared Memory

In a shared-memory system, cores share access to a memory. Multicore processors use a shared memory to coordinate to solve computational problems. Multicores concurrently execute several threads in different cores. Each thread is a sequential program and threads run at different speeds. Therefore, threads must communicate and synchronize to solve a problem. In order for threads to communicate, they read from and write to a data structure in a shared memory.

2.3 Concurrent Data Structures

A data structure is a way to store and organize data so that data can be accessed and modified efficiently. Since multicores concurrently execute several threads in different cores at different speeds, concurrent data structures must handle multiple access and modifications of data at the same time. This can be achieved by synchronizing access to a shared data structure so that multiple access and modifications of data that cause bad interleaving operations are sequentially executed. There are two ways to implement synchronization, which are blocking and non-blocking.

2.3.1 Blocking Scheme

A simple implementation to synchronize access to a data structure is the use of locks. By locking a data structure, only one thread is allowed to access while other threads are blocked from accessing the data structure. Since access to a data structure is restricted to only one thread at a time, use of locks introduces high contentions and sequential bottlenecks affecting computation performance. This problem can be eased by fine-grained locking which uses multiple locks to protect different parts of the data structure instead of blocking the entire data structure. In order to efficiently implement blocking, concurrent data structures must provide enough blocking to maintain proper sequential execution while allowing concurrent operations to proceed in parallel.

Even though fine-grained locking can deal with some issues, lock-based implementation in an asynchronous environment poses other challenges such as priority inversion, deadlock, and convoying. Priority inversion is when a lower-priority thread holding a lock can prevent a high-priority thread to proceed. Deadlock occurs if a thread holding a lock fails. Convoying is when a

thread holding a lock is preempted causing the rest of the threads to wait unnecessarily long. A lock-based implementation is called blocking since the delay of any one thread can delay other threads.

2.3.2 Non-blocking Scheme

A non-blocking implementation can overcome problems associated with a blocking implementation. Unlike blocking implementations, a non-blocking implementation provides better progress conditions, namely wait-freedom, lock-freedom, and obstruction-freedom. Wait-freedom guarantees every thread to progress regardless of other threads. Lock-freedom guarantees system-wide progress even if individual threads may not progress. Obstruction-freedom guarantees any threads that eventually execute in isolation to progress. Wait-freedom is the strongest progress condition and obstruction-freedom is the weakest progress condition among these three conditions. All three conditions assure that failure or indefinite delay of a thread does not prevent other threads to progress, which blocking implementation does not assure.

Non-blocking can be implemented using hardware synchronization primitives such as CAS.

2.4 Compare-and-Swap (CAS)

Modern multicore processors provide hardware to support synchronizations. These are called hardware synchronization primitives which assure that read and modify instructions are executed atomically. In other words, read and modify memory locations are executed as one step without any interruptions. The CAS is one type of hardware synchronization primitive and is commonly used for non-blocking implementations since it is compatible with a wide range of

systems. The CAS takes three arguments: address to a memory location, an expected value and a new value. The CAS operation loads a value from the address, compares the loaded value with the expected value and stores the new value if the loaded and expected values are matched. The CAS operation returns true if a new value is stored otherwise it returns false. Since the CAS operation is executed atomically, it can be used instead of a lock.

2.5 Correctness of Concurrent Data Structures

Operations on a sequential data structure are executed one by one in order. Therefore, it is easy to see correctness of a data structure by following preconditions and post conditions of operations. On the other hand, operations on a concurrent data structure are not totally ordered and a state of a concurrent data structure can be at the middle of a method call when an instruction in another concurrent method call is executed. An execution of a method call takes some time that starts with an invocation and ends with a response. Therefore, method calls by concurrent threads can overlap. Because of this reason, correctness of a concurrent data structure is harder to show. One way of verifying correctness of concurrent execution is to reorder to a sequential execution. There are three correctness conditions that we can use to reorder a concurrent execution to sequential execution: quiescent consistency, sequential consistency, and linearizability. Quiescent consistency indicates non-overlapping method calls to take effect in a one-at-a-time sequential order, but overlapping method calls may be in any order. Sequential consistency indicates that, in any concurrent executions, method calls can be sequentially ordered that follows program order. Linearizability indicates that a data structure is sequentially consistent and method calls are ordered by the real-time order of the linearization points. A linearization point is a time-point that can be considered to take effect of a method call between invocation and response of each method call.

Chapter 3

Related Work

As multi-core processors became ubiquitous, efficient and scalable data structures are becoming the focus of studies. Since data structures such as queues, stacks, skiplists and BST are widely used in the sequential environment, these data structures have also been studied for a concurrent environment. In this chapter, some of the past studies related to this are summarized.

3.1 Binary Search Tree (BST)

The essential operations of BST are insert, remove and search. The insert operation places a new node associated to a given key in a tree. The remove operation removes the node associated with a given key, or makes no change in a tree if such node does not exist. The search operation searches the tree to find a node associated with the given key and returns true if a node is found, or returns false otherwise. All three operations must traverse a tree, which increase contentions in concurrent BST. Furthermore, a concurrent BST must be able to handle rotations to restructure a tree while some threads are traversing the tree. One way to handle contentions is the use of fine-grained locking. For instance, Bronson et al. (2010) used hand-over-hand locking, which uses a chain of locks with fixed size, to continue the traverse during a concurrent rotation. Crain et al. (2013) suggested decoupling search, logical delete and insert operations from structural modifications to limit contention. Drachsler et al. (2014) suggested a partially non-blocking internal BST, which allows searching a tree without locks using logical ordering and lock-based insert and delete operations.

In BST, the delete operation from an internal tree is more complicated than the insert operation. The delete operation of a node with two children in an internal tree requires the replacement of the deleted node with its successor. The successor may be located as far away as one less than the tree height. While replacing the deleted node with its successor, every node along the path from the deleted node to its successor must be locked in a lock-based implementation of concurrent BST, which negatively affects performance and scalability. This problem can be simplified by the use of an external tree, which only stores key values in leaf nodes and the other nodes are only used for routing purposes. Deleting a leaf from an external tree is done by replacing the parent routing node with the sibling of the deleted leaf node. However, an external tree increases storage overhead and the average search path since an external tree with n data requires n-1 routing nodes. Therefore, Bronson et al. (2010) used a partially external tree which keeps a deleted node with two children as a routing node and removes them when routing nodes have fewer than two children. Similarly, in the algorithm reported in Crain et al. (2013), the delete operation was achieved by two steps if a node has two children. A node is first flagged as "deleted" and physically removed from a tree later when the node has only one child.

Recent research on concurrent BST is more towards non-blocking implementations since non-blocking is fault tolerant unlike a lock-based implementation. The first complete, non-blocking, linearizable BST implementation was reported by Ellen et al. (2010). It is an unbalanced external BST using CAS operations. To achieve a non-blocking implementation, their algorithm also uses a helping mechanism which allows a blocked process to progress by helping a blocking process. Natarajan and Mittal (2014) suggested a non-blocking external BST by flagging edges instead of nodes to reduce CAS operations. The insert operation in their

algorithm requires only a single CAS and consequently it does not need to help each other. A non-blocking internal binary search tree was introduced by Howley and Jones (2012) using CAS. They used a cooperative update by storing details about an ongoing update into a structure and a pointer to the structure into the node. Brown et al. (2014) introduced a simple template to obtain an implementation of non-blocking trees using load-link extended (LLX), store-conditional extended (SCX) and validate-extended (VLX) primitives. LLX, SCX and VLX primitives are multiword generalizations of load-link (LL), store-conditional (SC) and validate (VL) which can be implemented using CAS (Brown et al., 2013). LL reads the contents of a memory location. SC updates the contents of a memory location with a new value, and VL returns true if the location has not been modified since LL was last performed on the memory location. Brown et al. (2014) provided the implementation of a chromatic tree which is a relaxed-balanced red-black tree using their template. In most of non-blocking BST, when an insert or delete operation fails, the operation restarts from the root of a tree. Ellen et al. (2014) improved their previous non-blocking binary search tree (Ellen et al., 2010) to not restart from the root but by backtracking the tree.

More recent work is to improve the efficiency of existing BST algorithms. For instance, Ramachandra and Mittal (2015) presented non-blocking internal BST, which is efficient especially under a conflict-free condition using ideas from Howley and Jones (2012) and Natarajan and Mittal (2014). Chatterjee et al. (2016) reevaluated non-blocking algorithms to optimize the number of CAS steps considering efficiency and memory footprint.

3.2 Splay Tree

One type of self-adjusting BST is a splay tree in which the frequently accessed items are moved to the root of a tree by performing a sequence of rotations along the path from the node to the root (Sleator and Tarjan, 1985). As a splay tree is a variation of BST, items are arranged in symmetric order: If a node *x* contains an item *i*, its left subtree contains only nodes with items less than *i*, and its right subtree contains only nodes with items greater than *i*. Once an item is accessed, the node with the item is moved to the root by using combinations of three rotations, Zig, Zig-Zig, and Zig-Zag. Zig is used when the parent of node *x* is the tree root. Zig rotates the edge joining *x* with its parent. Zig-Zig is used when the parent of *x* is not the tree root, and both *x* and its parent are either right children or left children. Zig-Zig first rotates the edge joining the parent of *x* with its grandparent. Then, the edge joining *x* with its parent is rotated. Zig-Zag is used when the parent of *x* is not the tree root, and *x* is a left child and the parent of *x* is a right child, or vice-versa. Zig-Zag first rotates the edge joining *x* with its parent. Then, the edge joining *x* with the new parent of *x* is rotated. The combination of the rotations to move a node to the root is called splaying. Splaying not only moves a node to the root but also changes the depth of every node along the access path to roughly half. Therefore, a splay tree becomes more efficient as access to the tree continues. However, self-adjusting trees are not suitable for a concurrent setting because of constant rotations, especially around the root of a tree.

Afek et al. (n.d.) suggested a self-adjusting concurrent binary search tree by reducing to one local adjustment to the tree on each access, which is called lazy splaying. They implemented a lazy splay tree (LST) replacing Bronson et al. (2010)'s re-balancing code. Unlike a sequential splay tree, LST has two splaying steps, Zig and Zig-Zag. Zig and Zig-Zag rotations are the same as the splay tree reported in Sleator and Tarjan (1985). In LST, each node keeps track of the total

number of operations performed on the item (self-count), its left subtrees (left-count) and its right subtrees (right-count). The self-counter of the node containing *n*, and the left/right-counters of all the nodes along the path from the root to the node with *n*'s parent are increased by one when an insert(*n*) or search(*n*) is successfully executed. Zig is performed if a node is a left child of its parent and the total number of its self-count plus left-count is larger than the total number of parent's self-count plus parent's right count. Zig-Zag is performed if a node is a left child of its parent and its right-count is larger than the total number of parent's self-count plus right-count. If a node is a right child, Zig and Zig-Zag operations are performed as reflection symmetry.

Chapter 4

Algorithm

A lock-free self-adjusting BST is designed by modifying Afek et al. (n.d.)'s LST. In their algorithm, tree rotations and modifications (insert and delete) were implemented with the use of locking. In our lock-free algorithm, Afek et al. (n.d.)'s LST is modified based on Crain et al. (2013)'s contention friendly BST and Howley and Jones (2012)'s non-blocking internal BST.

4.1 Overview

This algorithm implements LST in a lock-free manner. LST is a self-adjusting BST and loosely follows a splay tree condition. In LST, frequently accessed items are moved to the root of a tree by performing one local rotation, which is called lazy-splay. Whether to rotate or not is decided based on the number of operations performed on the item (selfCount), its left subtree (leftCount) and its right subtrees (rightCount) as shown in Figure 4.1. The counters are not protected and it is possible to have incorrect counter values which may affect performance but not the safety properties of operations. Since Afek et al. (n.d.) reported that the effect of inaccurate counters on performances are negligible while elimination of the locking significantly improves the overall performances, the counters are used without locks in our algorithm.

Because of unprotected counters, it is possible to have an unbalanced tree (all nodes are right (left) parents of their children). Therefore, rotations are performed from a node with depth larger than $2log\ n$, where n is the number of nodes in the tree, to the root. This sequence of rotations is called semi-splay (Sleator and Tarjan, 1985). Since the purpose of semi-splay is to reduce the depth of the tree, rotations are performed slightly different from lazy-splay as shown

in Figure 4.2. Note that zig-zig rotation moves the current node's parent, y, to the top instead of the current node, x. Rotations for lazy-splay and semi-splay are performed by rotating the copy of the nodes and replacing the old nodes in the tree to the new rotated nodes.

Operations of insert, remove and search are included in the algorithm. These operations are executed separately from restructuring operations, semiSplay and lazySplay. The idea of separate execution is from a contention-friendly BST by Crain et al. (2013). A contention-friendly BST avoids contentions during traversal of a tree by eagerly updating keys and lazily restricting a tree. By separating restructuring operations, it is possible to use processor cores that might otherwise be idle to perform the structural adaptation (Crain et al., 2013). Furthermore, the remove operation physically removes a node selectively to reduce contention. If the node has less than two children, the node is physically removed. If the node has two children, the node is logically removed by flagging the field, removed, and postponing physical removal until the node has less than two children. The idea of two-step removal is commonly used (Bronson et al., 2010, Crain et al., 2013).

Lock-free is achieved using CAS and flagging a pointer that points to an ongoing operation. Storing the address of a memory location does not use all the allocated memory for a pointer. Therefore, the least significant bits of memory for a pointer can be used to store auxiliary data (Howley and Jones, 2012). These extra bits are used for flagging. If a node needs to be modified (inserting, removing or rotating a node), the pointer associated with the node is flagged and points to the data structure which stores necessary information to complete the ongoing operation.

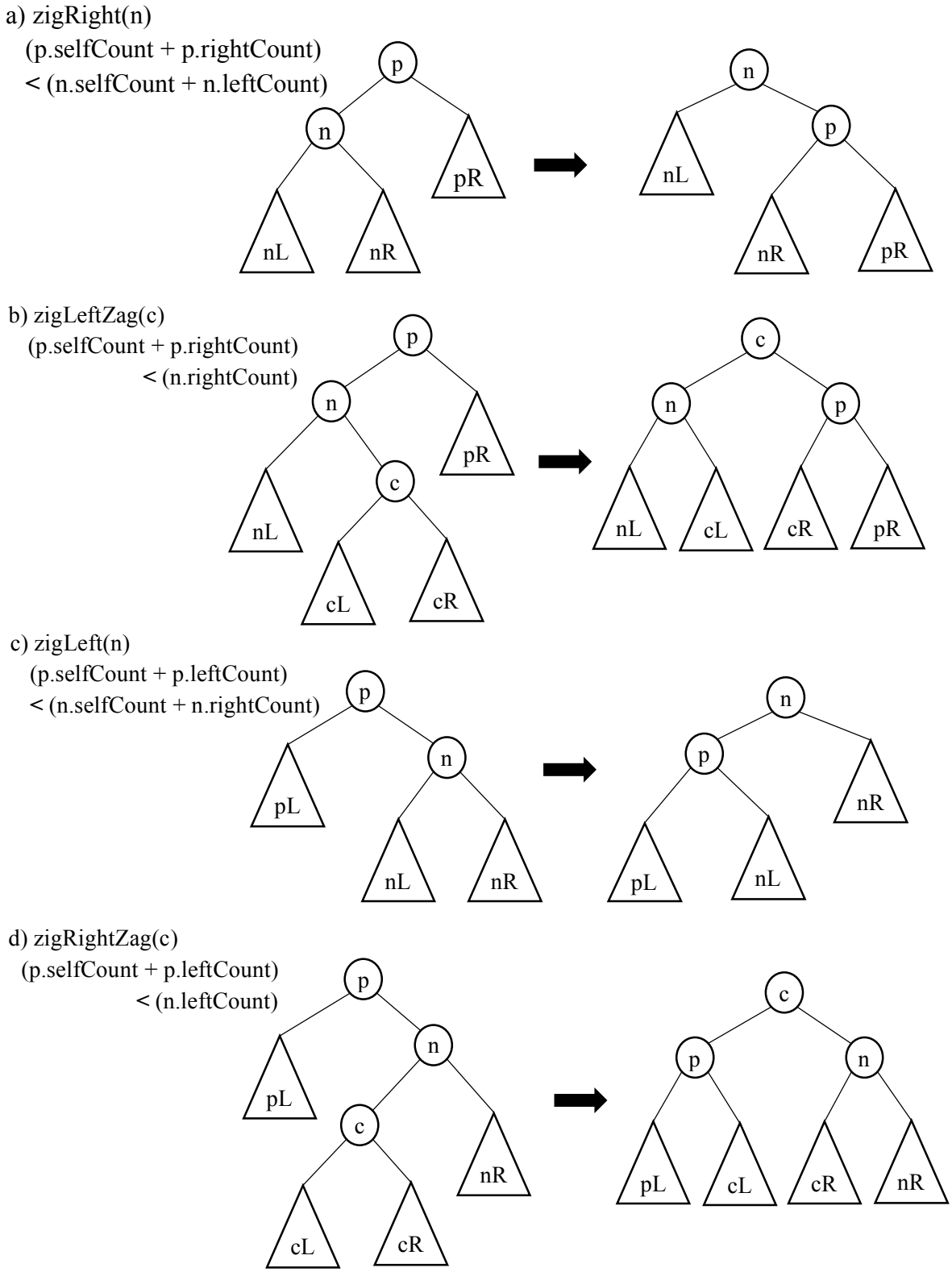

Figure 4.1 Lazy-splay conditions and rotations included in the lazySplay operation. Circle represents a node and triangle represent subtree. The selfCount, leftCount and rightCount are the number of operation performed on the item, its left subtree and its right subtree, respectively.

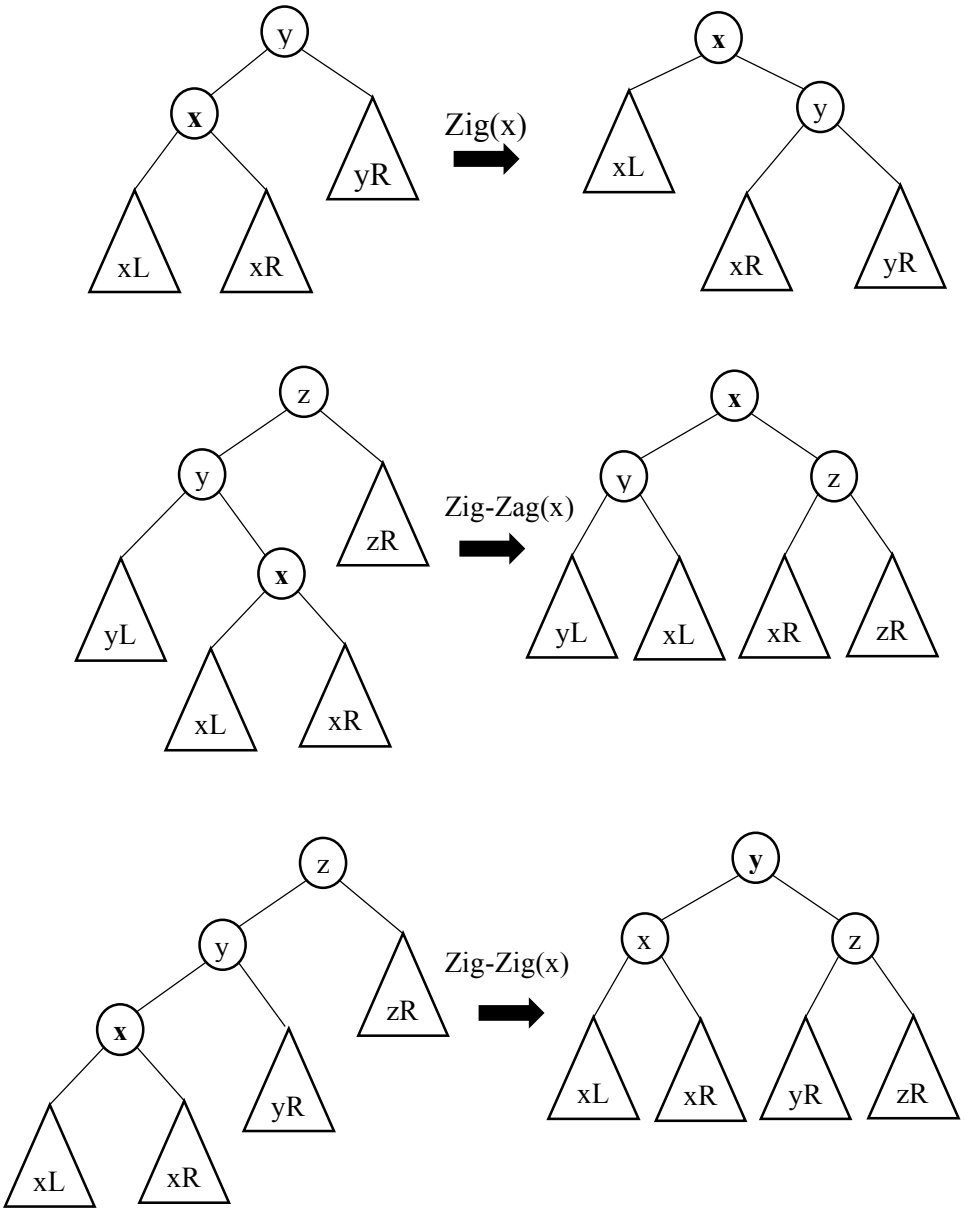

Figure 4.2 Semi-splay conditions and rotations included in the semiSplay operation. Circle represents a node and triangle represent subtree. Bold character is the current node.

4.2 Detailed Implementation

4.2.1 Data Structures

There are four data structures, node, childCASOp, helpRotateOp and findResult, that are used in the algorithm.

4.2.1.1 node

A node data structure in a tree has the following fields: key, op, removed, leftChild, rightChild, selfCount, leftCount and rightCount. The op is a pointer which points to one of two operation data structures which are childCASOp and helpRotateOp. The operation data structures store necessary information to change a child pointer of a node or to rotate nodes. The removed is a Boolean value to indicate if a node is logically removed. The leftChild and rightChild point to left and right children of a node, respectively. The selfCount, leftCount and rightCount are counters that keep track of the number of operations performed on the node itself, its left subtree and its right subtree, respectively. Counted operations are insert and search.

The op pointer is also used for flagging. Flagging status include NONE, CHILDCAS, REMOVE and ROTATE. NONE means no ongoing operation on a node. CHILDCAS means one of the child pointers of a node is being modified. REMOVE means the node is going to be physically removed, and ROTATE means the node is currently affected by rotations. Flagging/unflagging is aided by two macros defined as follow: FLAG(pointer, status) which sets the pointer to the status, and GETFLAG(pointer) which returns a status of flag in the pointer.

4.2.1.2 childCASOp

A childCASOp data structure is used to save information necessary to change a child pointer using CAS. A childCASOp has three fields: isLeft, expected and update. The isLeft is

true if modification is to the left child of a node and false otherwise. The expected and update are pointers to nodes that are to be there before and after modification, respectively.

4.2.1.3 helpRotateOp

A helpRotateOp data structure is used to save information necessary to rotate nodes. A helpRotateOp has the two following fields: whichRotation and findResult. The field whichRotation stores the choice of rotation, zigRight, zigLeft, zigLeftZag or zigRightZag. The field findResult stores a sequence of nodes that is affected by a rotation. See the following section for detail for findResult.

4.2.1.4 findResult

A findResult data structure keeps the sequence of nodes and their op pointers at the time they are found while a node is searched. A findResult has the following fields: ggp, gp, p, n, ggpOp, gpOp, pOp and nOp. The ggp, gp, and p are great-grandparent, grandparent, parent of the node, n, respectively. The ggpOp, gpOp, pOp and nOp are op pointers of great-grandparent, grandparent, parent and node, respectively.

4.2.2 search(key)

The pseudo-code for the search operation can be found in Algorithm 4.1. The search operation takes a key to be searched and returns a Boolean value. The search operation calls the find operation. The find operation (Algorithm 4.2) starts from the root and looks for a location of the given key or a place to insert the given key in a tree by recursively calling the find operation. The find operation takes a key, four nodes, two Boolean values and thread id as input and returns a pointer to the data structure, findResult. Necessity of tracking four nodes is because rotation affects at the utmost four nodes from the current nodes to its great-grandparent. Since nodes used in the algorithm do not have a parent pointer, a sequence of nodes cannot be backtracked.

Therefore, nodes must be tracked in the find operation, which is also called by the operations of rotations. A Boolean value, isLeft, is used for keeping track of the path to find the given key. If the search of the given key takes a left child of a node, isLeft is set to true. Otherwise isLeft is false. A Boolean value, isCounting, triggers counting since counters are incremented only for the insert and search operations whereas the find operation is called by other operations. The return value of a pointer may indicate retry if the find operation encounters a concurrent operation. The other possibilities of return values are a data set with the last visited leaf node if the node with the given key is not found, and a data set with the node with the given key. For the case of retry, the find operation helps completing the ongoing concurrent operation first then returns retry to the caller indicating the find operation needs to be restarted because of the existence of a concurrent operation. The search operation keeps calling the find operation until it receives a return value of a data set. The search operation then returns false if the find operation did not find the node with the key and the pointer to the childCASOp hasn't been changed from the time the node is found in the find operation. If the find operation locates the node with the key, the search operation checks if the node is logically removed along with an existence of concurrent operation. If the node is logically removed and there is no concurrent operation, the search operation returns false. If the node is not logically removed and there is no concurrent operation, the search operation returns true. Otherwise, the search operation restarts from the beginning.

Algorithm 4.1 search(key)

```
 1  bool search(key) {
 2      while (true)
 3          // initiate find operation until result is not retry
 4          while ((result = find(key, null, null, null, root, false, true, id)) == retry)
 5              continue
 6
 7          // key is not in the tree and no concurrent operation
 8          if (result.n.key != key && result.nOp == n.op)
 9              return false
10
11          // key is in the tree, no concurrent operation and not logically removed
12          if (result.n.key == key &&
13                  result.nOp == n.op && result.n.removed == false)
14              return true
15
16          // key is in the tree, no concurrent operation and logically removed
17          if (result.n.key == key &&
18                  result.nOp == n.op && result.n.removed == true)
19              return false
20  }
```

Algorithm 4.2 find(key, node, node, node, node, bool, bool, id)

```
 1    findResult find(key, node ggp, node gp, node p, node n,
 2                    bool isLeft, bool isCounting, id) {
 3        If (key == n.key)   // found
 4            If (isCounting && n.removed != true)
 5                n.selfCount++
 6            return result = findResult(ggp, gp, p, n, ggp.op, gp.op, p.op, n.op)
 7
 8        // find next node to search
 9        if (isLeft  = (key < n.key))
10            next = n.leftChild
11        else
12            isLeft = false
13            next = n.rightChild
14
15        // check if node is valid
16        if (GETFLAG(next.op) == CHILDCAS)
17            helpChildCAS(next.op, next)
18            return restart
19        else if (GETFLAG(next.op) == REMOVE)
20            helpRemove(n, n.op, next, id)
21            return restart
22        else if (GETFLAG(next.op) == ROTATE)
23            helpRotate(next.op)
24            return restart
25
26        if (next == null)     // not found
27            return result = findResult(ggp, gp, p, n, ggp.op, gp.op, p.op, n.op)
28
29        else            // continue searching
30            result = find(key, gp,  p, n, next, isLeft, isCounting, isInsert, id)
31
32            // counter update
33            if ((result != null || result != restart) && isCounting)
34                if (isLeft)
35                    n.leftCount++
36                else
37                    n.rightCount++
38            return result
39    }
```

4.2.3 insert(key, id)

The insert operation takes two values, key and id, and returns a Boolean value as shown in Algorithm 4.3. The key is the value to be inserted in a tree, and the id is a thread id which is used in calculating a threshold for executing semiSplay. A return value is true if a new node with the given key is inserted. A return value is false if a node with the given key already exists in the tree.

The insert operation calls the find operation until the find operation returns a result other than retry. If the find operation finds a node with the given key, the insert operation checks if the node is logically removed along with the existence of a concurrent operation. In the case of a logically removed node and there is no concurrent operation, the removed field of the node is set back to false to indicate the node is inserted back in the tree. If the node with the given key is not in the tree, a new node is created and inserted in the tree using CAS. First, CAS is used to store necessary information to insert the created new node in the current node's childCASOp data set while flagging the pointer as CHILDCAS. The success of the CAS execution means that a new node is logically inserted. Since the necessary information to insert a new node is in the childCASOp field, any concurrent thread traversing the node can complete inserting the new node. Physically inserting a new node is completed by calling helpChildCAS. The helpChildCAS uses two CASs. The first CAS inserts a new node and the second CAS flags back the operation pointer to NONE indicating the new node is physically inserted (Algorithm 4.4). Once a new node is logically inserted in the tree, localCounterInsert (Algorithm 4.5) is called to recalculate a threshold for semiSplay. Return values of true or false from the insert operation indicate successful insertion, or existence of a node with the given key in the tree, respectively.

Algorithm 4.3 insert(key, id)

```
 1  bool insert(key, id) {
 2      while (true)
 3          while ((result = find(key, null, null, null, root, false, true, id)) == retry)
 4              continue
 5
 6          if (result.n.key == key)          // node with the given key is found
 7              // logically removed case
 8              if (result.n.removed == true && result.nOp == n.op)
 9                  CAS(&result.n.removed, true, false)
10                  return true
11              return false
12          else                              // node with then given key is not found
13              newNode = new Node(key)
14              bool isLeft = (result.n.key > key)
15              if (isLeft)
16                  node old = result.n.leftChild
17              else
18                  node old = result.n.rightChild
19              casOp = new childCASOp(isLeft, old, newNode)
20              if (CAS(&result.n.op, FLAG(result.nOp, NONE),
21                      FLAG(casOp, CHILDCAS))       // considered logically inserted
22                  localCounterInsert(id)
23                  helpChildCAS(casOp, result.n)
24                  return true
25  }
```

Algorithm 4.4 helpChildCAS(childCASOp, node)

```
1   bool hlepChildCAS(childCASOp op, node n){
2       // update nodes
3       if (op.isLeft)
4           CAS(&n.left, op.expect, op.update)
5       else
6           CAS(&n.right, op.expect, op.update)
7
8       // set flag in op pointer to NONE
9       if (CAS(&n.op, FLAG(op, CHILDCAS), FLAG(op, NONE)))
10          return true
11      else return false
12  }
```

Algorithm 4.5 localCounterInsert(id)

```
1   localCounterInsert(id) {
2       localCounter[id]++          // local counter per thread
3       if (localCounter[id] > max[id])
4           max[id] = localCounter[id]
5       if (localCounter[id] > 2*min[id])
6           max[id]=min[id]=localCounter[id]
7           succeeded = false
8           while(!succeeded)
9               prevLogSize = logSize;
10              totalSize = 0
11              for (i=0; I < NUM_OF_THREADS; i++)
12                  totalSize += localCounter[i]
13              if (log(totalSize) != logSize)
14                  succeeded = CAS(&logSize, prevLogSize, log(totalSize))
15  }
```

4.2.4 remove(key, id)

The remove operation takes two values, key and id and returns a Boolean value as shown in Algorithm 4.6. The key is the value to be removed from the tree, and the id is a thread id which is used to calculate a threshold for semiSplay. A return value is true if the node with the given key is removed. If the node with the given key is not found in the tree, false is returned.

The remove operation calls the find operation until the find operation returns a result other than retry. If the find operation returns the node with a different key, which means the node with the given key is not in the tree, the remove operation checks the existence of a concurrent operation. If there is no concurrent operation, the remove operation exits returning false. If the find operation finds the node with the given key, the remove operation checks the number of children for the node. If the node has less than two children, CAS is executed to store necessary information for removing the node from the tree in the node's childCASOp, and to flag the pointer that points to childCASOp to REMOVE. If the CAS failed, then the remove operation starts from the beginning. Otherwise, the node is considered as logically removed. Therefore, the removed flag is changed to true. Then, helpRemove (Algorithm 4.7) is called to physically remove the node. The helpRemove operation replaces the child pointer of the node's parent with the node's existing child pointer, or null if the node does not have a child, by calling helpChildCAS. Once the childCASOp field is set for CHILDCAS, physical removal of the node cannot fail. At this point, localCounterRemove (Algorithm 4.8) is called to update a threshold of semiSplay and to end the execution. If the node has two children and there is no concurrent operation, the removed field is changed to true using CAS and the remove operation returns true. Otherwise, the remove operation restarts from the beginning.

Algorithm 4.6 remove(key, id)

```
 1   bool remove(key, id) {
 2       while (true)
 3           while (result = (find(key, null, null, null, root, false, true, id) == retry))
 4               continue
 5
 6           // key is not in the tree
 7           If (result.n.key != key && result.nOp == n.op) return false
 8
 9           // key found
10           if (result.n.key == key)
11               // node with < 2 children
12               if (result.n.leftChild == null || result.n.rightChild == null)
13                   if (CAS(&n.op, FLAG(result.nOp, NONE), FLAG(n.op, REMOVE))
14                       CAS(&result.n.removed, false, true)
15                       If (helpRemove(result.p, result.p.op, result.n, id))
16                           return true
17                       else
18                           CAS(&n.op, FLAG(n.op, REMOVE), FLAG(n.op, NONE))
19
20               // node with 2 children
21               else if (result.nOp == n.op)
22                   CAS(&result.n.removed, false, true)
23                   return true
24   }
```

Algorithm 4.7 helpRemove(node, childCASOp, node, id)

```
1   bool helpRemove(node p, childCASOp pOp, node n, id) {
2       // check which child to replace a deleted node
3       if (n.leftChild == null)
4           if (n.rightCHild == null)
5               node newRef = null
6           else
7               node newRef = n.rightChild
8       else
9           node newRef = n.leftChild
10
11      // physically remove
12      casOp = new childCASOp(p.childLeft == n, n, newRef)
13      if (CAS(&p.op, pOp, FLAG(casOp, CHILDCAS))
14          helpChildCAS(casOp, p)
15          localCounterRemove(id)
16          return true
17      else
18          return false
19  }
```

Algorithm 4.8 localCounterRemove(id)

```
1   localCounterRemove(id) {
2       localCounter[id]--          // local counter per thread
3       if (localCounter[id] < min[id]
4           min[id] = localCounter[id]
5       if (localCounter[id] < 2*max[id]
6           max[id]=min[id]=localCounter[id]
7           succeeded = false
8           while(!succeeded)
9               prevLogSize = logSize;
10              totalSize = 0
11              for (i=0; I < NUM_OF_THREADS; i++)
12                  totalSize += localCounter[i]
13              if (log(totalSize) != logSize)
14                  succeeded = CAS(&logSize, prevLogSize, log(totalSize))
15  }
```

4.2.5 lazySplay(node, id)

The pseudo-code for lazySplay is given in Algorithm 4.9. The lazySplay operation traverses through a tree using a depth-first search and calls the find operation to locate the sequence of nodes that are required to rotate the given node. If the node is logically removed, the lazySplay operation calls the remove operation to try to physically remove the node. If the node is physically or logically removed, the node is skipped for lazy-splaying. Otherwise, the node is checked for the lazy-splay conditions shown in Figure 4.1. If the node matches one of the lazy-splay conditions, an appropriate operation to rotate the node is called. There are four operations to rotate a node, zigRight, zigLeft, zigLeftZag and zigRightZag. In any outcomes of no rotation required, successful rotation, and unsuccessful rotation, the lazySplay operation moves to the next node.

To make the algorithm lock-free, the rotational operations are implemented using CAS. The major issue of using CAS to execute a rotation is that a rotation requires several pointers to be changed at a time whereas CAS allows only one change at a time. In an asynchronous system, this may cause a halt in the middle of a rotation, which may lead to concurrently traversing threads to a wrong outcome. Therefore, instead of modifying an existing nodes' child pointers, new nodes are created by copying the old nodes and linking new nodes after the rotated condition. At the end, the top most child pointer is changed to point to the new nodes using CAS to remove the old nodes and insert the rotated new nodes. Note that the old nodes' child pointers are not modified. Therefore, threads traversing old nodes are still able to reach any nodes that were reachable before the rotation. The two particular rotations, zigRight and zigLeftZag, are discussed below.

Algorithm 4.9 lazySplay(node, id)

```
 1    void lazySplay(node n, id) {
 2        if (n == null) return
 3        lazySplay(n.leftChild, id)
 4        lazySplay(n.rightChild, id)
 5
 6        while ((result = find(key, null, null, null, root, false, true, id)) == retry)
 7            continue
 8
 9        if (result.n != n) return
10
11        if (result.n.removed == true)        // logically removed node
12            remove(n, id)                    // try physically removing
13            return                           // if the node is removed, don't splay
14
15        // node n is left child
16        if (result.n.rightCount >= result.p.selfCount + result.p.rightCount)
17            if (result.gp.leftChild == result.p || result.gp.rightChild == result.p)
18                if (result.p.leftChild == result.n)
19                    if (result.n.rightChild != null)
20                        zigLeftZag(result.n.rightChild)
21        else if (result.n.selfCount + result.n.leftCount >
22                    result.p.selfCount + result.p.rightCount)
23            if (result.gp.left == result.p || result.gp.right == result.p)
24                if (result.p.left == result.n)
25                    zigRight(result.n)
26
27        // node n is right child
28        if (result.n.leftCount >= parentPlusLeftCount)
29            if (gp.leftChild == p || gp.rightChild == p)
30                if (p.rightChild == n)
31                    if (n.leftChild != null)
32                        zigRightZag(n.leftChild)
33        else if (result.n.selfCount + result.n.rightCount >
34                    result.p.selfCount + result.p.leftCount)
35            if (result.gp.left == result.p || result.gp.right == result.p)
36                if (result.p.right == result.n)
37                    zigLeft(result.n)
38
39        return
40    }
```

The zigRight operation (Algorithm 4.10) calls the find operation to find the sequence of nodes that will be affected by rotations. Once the nodes that are affected by a rotation are found, the nodes' op pointers are flagged as ROTATE. If all the pointers cannot be flagged, the pointers are un-flagged and the zigRight operation returns without rotations. If all the pointers are flagged, the helpRotate operation is called. The helpRotate operation checks which rotation is to be made and calls an appropriate operation among four choices, helpRotateZigRight, helpRotateZigLeft, helpRotateZigLZag or helpRotateZigRZag (Algorithm 4.11). For the zigRight operation, helpRotateZigRight is called (Algorithm 4.12). In the helpRotateZigRight operation, if all the nodes affected by a rotation are flagged as ROTATE, new nodes, n_n and p_n, are created copying the old nodes, n and p, respectively (Figure 4.3). Then, the status of flag and pointers are modified for n_n and p_n. The new nodes' op field is flagged back to NONE. The n_n's right child pointer is linked to p_n, and the p_n's left child pointer is linked to the n's right child. The counters that are tracking the number of operations applied to the nodes, n_n and p_n, are adjusted at this point. Finally, the grandparent's child pointer is changed to point to n_n replacing the old nodes, n and p, by the new nodes, n_n and p_n by using CAS. The zigLeft operation is the mirror operation of zigRight.

The zigLeftZag operation follows the same flow as zigRight as shown in Algorithm 4.13. The difference is the number of nodes affected by a rotation. There are four nodes that are affected by the zigLeftZag operations and all nodes must be flagged as ROTATE. Failure of flagging means that the rotation is not executed. Once flagging is successful, helpRotateZigLZag is called (Algorithm 4.1). If all the nodes affected by a rotation are flagged as ROTATE, new nodes, n_n, p_n and gp_n are created copying the old nodes, n, p and gp, respectively (Figure 4.4). Then, the new nodes are flagged as NONE, and the child pointers of the new nodes are modified.

The node, n_n's left and right child's pointers are linked to p_n and gp_n, respectively. The p_n's right child pointer is linked to the n's left child, and the gp_n's left child pointer is linked to the n's right child. The CAS is applied to the great-grandparent's child pointer to point to n_n replacing the old nodes, n, p and gp by the new nodes, n_n, p_n and gp_n. The zigRightZag operation is the mirror operation of zigLeftZag.

Algorithm 4.10 zigRight(node)

```
 1    void zigRight(node n) {          // node n is the one to go up
 2        result = find(n.key, null, null, null, root, false, false, id)
 3
 4        // concurrent operation modified the tree
 5        if (result == retry || result.n != n) return
 6
 7        // flag nodes
 8        bool flagSuccess = false
 9        result.gp.op = result.p.op = result.n.op =  new helpRotateOp(zigRight, result)
10        gpOp = result.gpOp
11        pOp = result.pOp
12        nOp = result.nOp
13        if (CAS(&result.gp.op, FLAG(result.gpOp, NONE),
14                FLAG(result.gp.op, ROTATE)
15            if (CAS(&result.p.op, FLAG(result.pOp, NONE),
16                    FLAG(result.p.op, ROTATE)))
17                if (CAS(&result.n.op, FLAG(result.nOp, NONE),
18                        FLAG(result.n.op, ROTATE)))
19                    flagSuccess = true
20        if (!flagSuccess)
21            CAS(&result.n.op, FLAG(result.n.op, ROTATE), nOp)
22            CAS(&result.p.op, FLAG(result.p.op, ROTATE), pOp))
23            CAS(&result.gp.op, FLAG(result.gp.op, ROTATE), gpOp)
24            return
25
26        helpRotate(result.n)
27
28        return
29    }
```

Algorithm 4.11 helpRotate(node)

```
1    helpRotate(node n) {
2        if (n.op.whichRotation == zigRIght)
3            helpRotateZigRIght(n.op.findResult)
4        else if (n.op.whichRotation == zigLeft)
5            helpRotateZigLeft(n.op.findResult)
6        else if (n.op.whichRotation == zigLeftZag)
7            helpRotateZigLZag(n.op.findResult)
8        else
9            helpRotateZigRZag(n.op.findResult)
10   }
```

Algorithm 4.12 helpRotateZigRight(findResult)

```
1   helpRotateZigRight(findResult result) {
2       if (GETFLAG(result.gpOp) != ROTATE) return
3       bool flagSuccess = false
4       result.p.op = result.n.op =  result.gp.op
5       pOp = result.pOp
6       nOp = result.nOp
7       result.nOpN = FLAG(result.nOp, NONE)
8       if (GETFLAG(result.pOp) == ROTATE || CAS(&result.p.op,
9               FLAG(result.pOp, NONE), FLAG(result.p.op, ROTATE)
10          If (GETFLAG(result.nOp) == ROTATE || CAS(&result.n.op,
11                  FLAG(result.nOp, NONE), FLAG(result.n.op, ROTATE)
12              flagSuccess = true
13      if (!flagSuccess)
14          CAS(&result.n.op, FLAG(result.n.op, ROTATE), nOp)
15          CAS(&result.p.op, FLAG(result.n.op, ROTATE), pOp)
16          CAS(&result.gp.op, FLAG(result.n.op, ROTATE), gpOp)
17          Return
18
19      // create new node pnew and nnew
20      pnew = new Node(result.p)
21      nnew = new Node(result.n)
22
23      // flag pnew and nnew's op to NONE
24      FLAG(pnew.op, NONE)
25      FLAG(nnew.op, NONE)
26
27      // set pnew and nnew's child pointers
28      pnew,lefChild = result.n.rightChild
29      nnew.rightChild = result.p
30
31      // update counter
32      pnew.leftCount = result.n.rightCount
33      nnew.rightCount = result.p.selfCount + result.p.rightCount + result.n.rightCount
34
35      // replace p and n with pnew and nnew
36      bool isLeft = (result.gp.leftChild == p)
37      casOp = new childCASOp(isLeft, result.p, nnew)
38      if (CAS(&result.gp.op,
39              FLAG(result.gp.op, ROTATE), FLAG(casOp, CHILDCAS)))
40          helpChildCAS(casOp, result.gp)
41
42      return
43  }
```

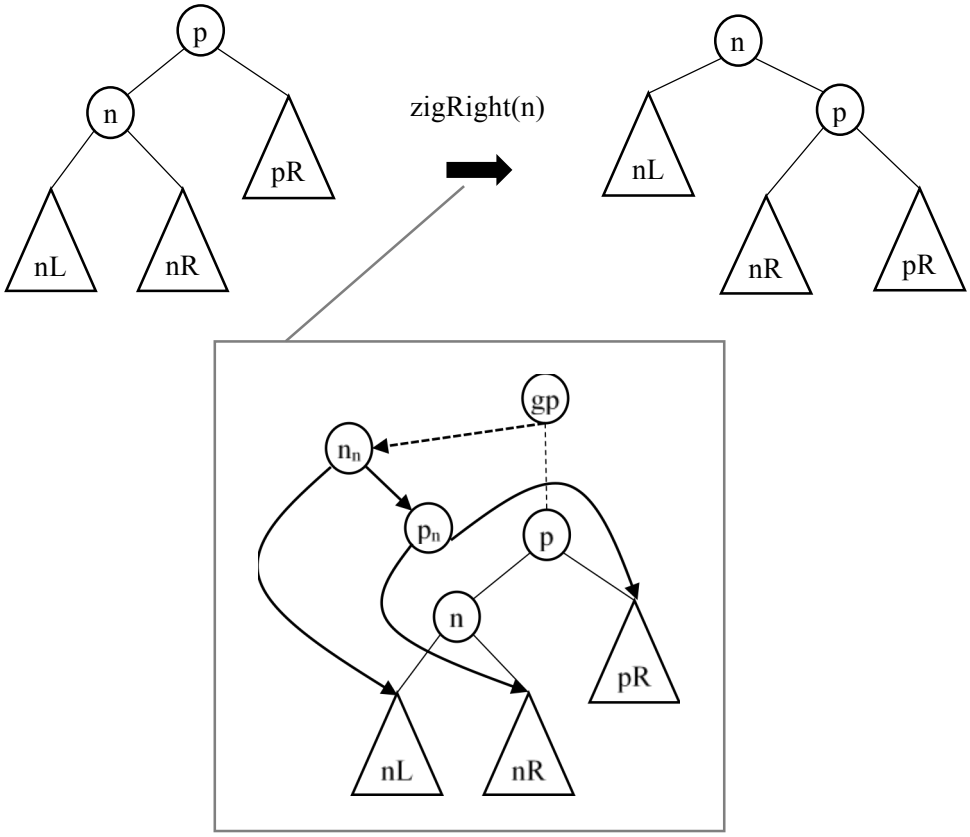

Figure 4.3 Schematic diagram of zigRight(n) implementation. Lines with allow are newly created link and dotted lines are replaced using CAS. The lines without allow are existing link before the rotation.

Algorithm 4.13 zigLeftZag(node)

```
 1    void zigLeftZag(node n) {
 2        result = find(n.key, null, null, null, root, false, false, id)
 3
 4        // concurrent operation modified the tree
 5        if (result == retry || result.n != n) return
 6
 7        // flag nodes
 8        result.ggp.op = result.gp.op = result.p.op = result.n.op =
 9                                        new helpRotateOp(zigRight, result)
10        ggpOp = result.ggpOp
11        gpOp = result.gpOp
12        pOp = result.pOp
13        nOp = result.nOp
14        if (CAS(&result.ggp.op, FLAG(result.ggpOp, NONE),
15                FLAG(result.ggp.op, ROTATE)))
16            if (CAS(&result.gp.op, FLAG(result.gpOp, NONE),
17                    FLAG(result.gp.op, ROTATE)))
18                if (CAS(&result.p.op, FLAG(result.pOp, NONE,
19                        FLAG(result.p.op, ROTATE)))
20                    if (CAS(&result.n.op, FLAG(result.nOp, NONE),
21                            FLAG(result.n.op, ROTATE))
22                        flagSuccess = true
23        if (!flagSuccess)
24            CAS(&result.n.op, FLAG(result.n.op, ROTATE), nOp)
25            CAS(&result.p.op, FLAG(result.p.op, ROTATE), pOp)
26            CAS(&result.gp.op, FLAG(result.gp.op, ROTATE), gpOp)
27            CAS(&result.ggp.op, FLAG(result.ggp.op, ROTATE), ggpOp)
28            return
29
30        helpRotate(result.n)
31
32        return
33    }
```

Algorithm 4.14 helpRotateZigLZag(findResult)

```
1   helpRotateZigLZag(findResult result) {
2       if (GETFLAG(result.ggp.op) != ROTATE) return
3       bool flagSuccess = false
4       result.gp.op = result.p.op = result.n.op = result.ggp.op
5       gpOpN = FLAG(result.gpOp, NONE)
6       pOpN = FLAG(result.pOp, NONE)
7       nOpN = FLAG(result.nOp, NONE)
8       if (GETFLAG(result.gp.op) == ROTATE || CAS(&result.gp.op,
9               FLAG(result.gpOp, NONE), FLAG(result.gp.op, ROTATE)
10          If (GETFLAG(result.p.op) != ROTATE || CAS(&result.p.op,
11                  FLAG(result.pOp, NONE), FLAG(result.p.op, ROTATE)
12          If (GETFLAG(result.n.op) == ROTATE || CAS(&result.n.op,
13                      FLAG(result.nOp, NONE), FLAG(result.n.op, ROTATE)
14                  flagSuccess = true
15      if (!flagSuccess)
16          CAS(&result.n.op, FLAG(result.n.op, ROTATE), nOp)
17          CAS(&result.p.op, FLAG(result.p.op, ROTATE), pOp)
18          CAS(&result.gp.op, FLAG(result.gp.op, ROTATE), gpOp)
19          CAS(&result.ggp.op, FLAG(result.ggp.op, ROTATE), ggpOp)
20      // create new node gpnew, pnew and nnew
21      gpnew = new Node(result.gp)
22      pnew = new Node(result.p)
23      nnew = new Node(result.n)
24      // set gpnew, pnew and nnew's child pointers and flag op pointers to NONE
25      gpnew.leftChild = result.n.rightChild
26      pnew,rightChild = result.n.leftChild
27      nnew.leftChild = result.p
28      nnew.rightChild = result.gp
29      FLAG(gpnew.op, NONE)
30      FLAG(pnew.op, NONE)
31      FLAG(nnew.op, NONE)
32      // update counter
33      gpnew.leftCount = result.n.rightCount
34      pnew.rightCount = result.n.leftCount
35      nnew.rightCount = result.gp.selfCount + result.gp.rightCount + result.n.rightCount
36      nnew.leftCount = result.p.selfCount + result.p.leftCount + result.n.leftCount
37      // replace p and n with pnew and nnew
38      bool isLeft = (result.ggp.leftChild == gp)
39      casOp = new childCASOp(isLeft, result.gp, nnew)
40      if (CAS(&result.ggp.op,
41              FLAG(result.ggp.op, ROTATE), FLAG(casOp, CHILDCAS))
42          helpChildCAS(casOp, result.ggp)
43      return
44  }
```

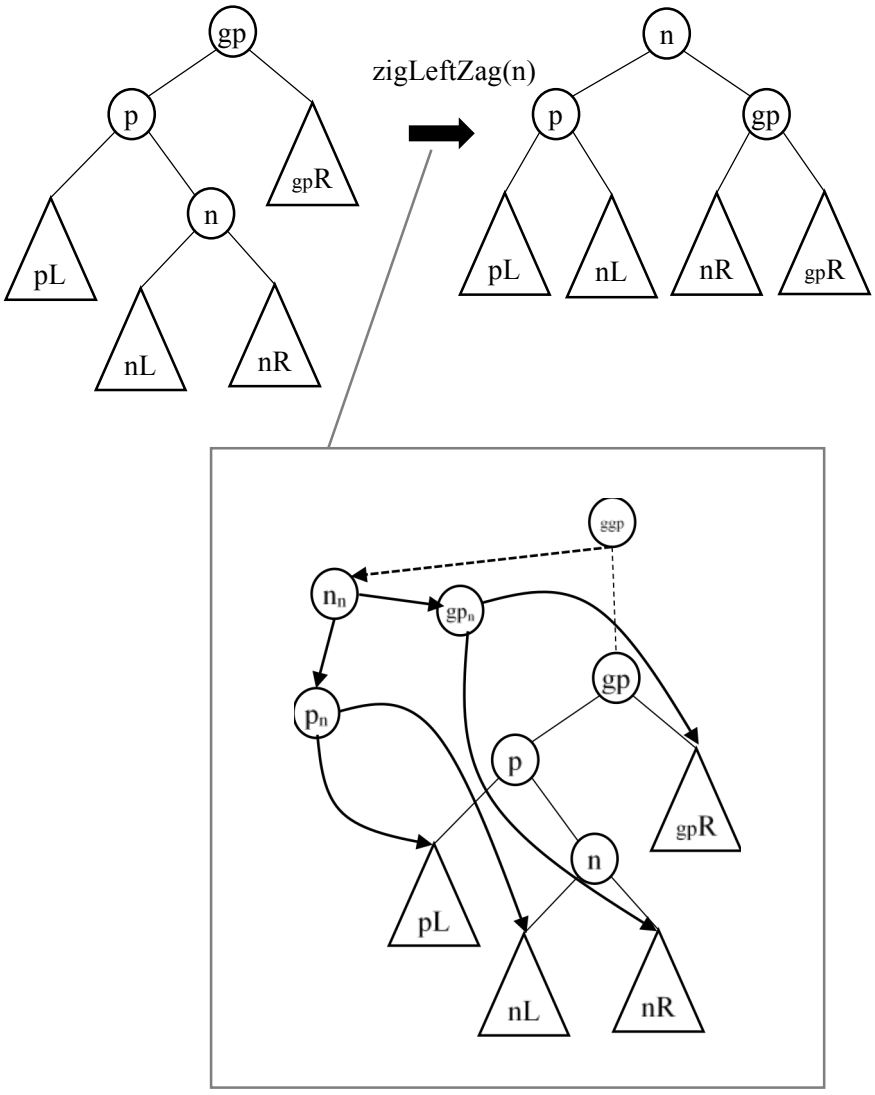

Figure 4.4 Schematic diagram of zigLeftZag(n) implementation. Lines with allow are newly created link and dotted lines are replaced using CAS. The lines without allow are existing link before the rotation.

4.2.6 semiSplay(node, height)

A pseudo-code for semiSplay is given in Algorithm 4.15. The semiSplay operation traverses through a tree using a depth-first search. When the height of the tree becomes more than 2*log n*, where *n* is the total number of nodes in the tree, nodes are rotated to reduce the height of the tree. Rotations executed in the semiSplay operation are shown in Figure 4.2. The semiSplay operation calls the find operation to locate a sequence of nodes that will be affected by rotation. If the current node is logically removed, the semiSplay operation calls the remove operation to try to physically remove the node. If the node is physically or logically removed, the node is skipped for semi-splaying. Otherwise, depending on the arrangement of nodes as shown in Figure 4.2, an appropriate rotation, zigRight, zigLeft, zigRightZag or zigLeftZag, is called.

The restructuring operations, lazySplay and semiSplay, are executed in the background and they can be called through the backgroundRestructure operation (Algorithm 4.16).

Algorithm 4.15 semiSplay(node, height)

```
1   void semiSplay(node n, int height) {
2       if (n == null) return
3       semiSplay(n.leftChild, height+1)
4       semiSplay(n.rightChild, height+1)
5
6       // semi splay when height is more than 2 x log(N)
7       if (height >= 2*logSize)
8           while ((result = find(n.key, null, null, null, root, false, false, id)) == retry)
9               continue
10
11          if (result.n != n) return
12
13          if (result.n.removed == true)  // logically removed node
14              remove(n, id)                       // try physically removing
15              return                              // if the node is removed, don't splay
16
17          if (result.n == n && result.p != null)
18              If (result.gp == null)        // Zig: node n does not have a grand parent
19                  If (result.n == result.p.leftChild)
20                      zigRight(result.n)
21                  else
22                      zigLeft(result.n)
23              else          // ZigZig or ZigZag: node n has a grand parent
24                  if (result.n == result.p.leftChild)
25                      if (result.p == result.gp.leftChild)
26                          zigRight(result.p)
27                      else
28                          zigRightZag(result.n)
29                  else
30                      if (result.p == result.gp.rightChild)
31                          zigLeft(result.p)
32                      else
33                          zigLeftZag(result.n)
34
35      return
36  }
```

Algorithm 4.16 backgroundRestructure(id)

```
1    backgroundRestructure(id){
2        while true  // restructure is continuously running
3            semiSplay(root, 0)
4            lazySplay(root, id)
5    }
```

Chapter 5

Correctness

Correctness of our lock-free self-adjusting BST algorithm is discussed in terms of the non-blocking condition and linearizability.

5.1 Non-blocking

In our algorithm, search, insert, remove and restructuring BST are implemented using CAS instead of using locks. To show the algorithm is non-blocking, interactions between reading and writing are discussed below.

A thread traversing a tree will eventually locate the key for which it is searching or stop at a leaf node. However, the thread may need to restart traversing the tree in a few occasions. If there is a concurrent operation which already flagged the node's operation pointer, the traversing thread tries to help complete the ongoing operation, and traversing will be restarted. Flagging an operation pointer always happens when the flag of the operation pointer is NONE, meaning that no operation is executed at this node. Therefore, successful flagging indicates that no other operation can be applied until the current operation finishes. However, successful flagging also means that the data structure which stores necessary information to complete the ongoing operation is successfully updated for the current operation. Using the stored data, any other thread which encounters the ongoing operation can complete the current operation and restart traversing the newly updated BST. This implies system-wide progress.

Other restarts of traversing happen when an operation pointer has been changed after the time the key was found. The changes to operation pointers are caused by flagging as

CHILDCAS, REMOVE or ROTATE. To be able to change the flag means that another thread is making advances, indicating that system-wide progress is happening.

When a rotation happens, several nodes are affected. However, rotation is done by creating new nodes with the rotated condition and replacing the new nodes with the old nodes by changing one child pointer in the tree. The child pointers of the old nodes are not modified. Therefore, any threads in the old nodes will still be able to traverse after the rotation. Furthermore, if a thread traversing the tree encounters the node with the rotating nodes, the thread can help complete the rotation by getting information from the stored data. Therefore, a rotation will not stop any thread to continue, even if a thread may require restarting a tree traversal.

5.2 Linearisability

The linearisability of our algorithm is discussed by defining the linearization points of the search, insert, remove and rotation operations.

5.2.1 Search Operation

Two possible outcomes can be returned from the search operation: the key is found or not found in a tree. The linearization point of finding the key is the point when the matching key is found in a tree (line 3 in Algorithm 4.2). However, it is possible that the node has been logically deleted or another traversing thread modifies the found node while the search operation waits for the find operation to return its result. Therefore, the linearization point of finding a key is when the search operation verifies that no changes are made in the operation pointer and the node is not logically deleted at line 12 in Algorithm 4.1.

There are two cases for the search operation to return false as not finding a key. The first case is when a key is not found in a tree. The linearization point of not finding a key is the point when the next node is read as null (line 26 in Algorithm 4.2). However, as the same with the found case, this result needs to be verified in the search operation. Therefore, the linearization point of not finding a key is when the search operation verifies no changes in the operation pointer at line 8 in Algorithm 4.1. For the second case, a key is found in a tree but the node was logically removed. The linearization point of this case is when the search operation verifies that there is no concurrent operation and the node is logically removed at line 17 in Algorithm 4.1.

5.2.2 Insert Operation

If a key is already in a tree, an insert operation fails. Therefore, similar to the search operation of finding the key, the linearization point of failing insertion is when the find operation finds the key and the insert operation verifies no change in an operation pointer (line 8 in Algorithm 4.3).

A successful insertion occurs at two places. When the key is found but it was logically removed, an insertion occurs when the removed flag is verified to be false using CAS. One of the linearization points of a successful insertion happens at line 9 in Algorithm 4.3. The other successful insertion occurs when the childCASOp operation is inserted into the inserting node using CAS (line 20 in Algorithm 4.3). Once childCASOp is set, any other thread encounter of the CHILDCAS flag must help complete the insert operation before moving forward. Therefore, when CAS succeeds, the key is considered logically inserted.

5.2.3 Remove Operation

A failure of remove is determined when the key is found in the tree and verified that no changes have been made in the operation pointer after the key is found. Therefore, the linearization point of a failure of the remove operation is line 7 in Algorithm 4.6.

A successful removal of the key occurs in two ways. When the node has two children, the key is considered as removed when the node's removed field is marked true. One of the linearization points of successful removal is at line 22 in Algorithm 4.6. If the node has less than two children, the successful removal of the key is not guaranteed when the node is flagged as REMOVE. It occurs when the operation pointer of the current node's parent is flagged as CHILDCAS. Once the parent's operation pointer is flagged, any other thread encounter CHILDCAS flag must help complete the remove operation before moving forward. The linearization point for this is at line 13 in helpRemove (Algorithm 4.7).

5.2.4 Rotational Operation

Unlike insert and remove operations, the rotational operation affects several nodes. By flagging the operation pointer of one of the nodes does not guarantee that rotation will succeed. Even though any thread which encounters the node with the operation pointer flagged as ROTATE can help complete the operation, a successful rotation only occurs when the new rotated nodes, which are copied and modified from the old nodes, are replaced with the old nodes. This is achieved by changing the top most node of the child pointer from the old node to a new node. For the case of zigRight rotation, the linearization point occurs at line 38 in helpRotateZigRight (Algorithm 4.12). For the case of zigLeftZag, this occurs at line 40 in helpRotateZigLZag (Algorithm 4.14).

Chapter 6

Conclusion and Future Work

In this study, a lock-free self-adjusting BST is proposed using CAS and flagging techniques. Specifically, an algorithm for LST is considered based on the contention friendly BST. LST executes one local rotation at a time to bring the frequently accessed items to the root of a tree. Whether to rotate or not is decided based on non-protected self, left and right counters, that track the number of operations performed on a node. Since the counters are not protected, semi-splay is also included in the algorithm in order to reduce the height of a tree if the height becomes more than 2*log* n, where n is the total number of nodes. The rotation operations are executed in the background following a contention friendly BST algorithm. Furthermore, the rotational operation is executed by making new nodes with the rotated condition and changing the top most node's child pointer to replace the old nodes with new rotated nodes. This overcomes the issue that rotation requires multiple pointers to be changed while CAS can change only one pointer at a time. Use of flagging of the operation pointer that points to a data structure which stores necessary information to complete the ongoing operation, allows other threads to help. This enables system-wide progress. In this study, pseudo-code of a lock-free self-adjusting BST is presented, and linearization points of insert, remove, search and rotation operations are established.

As a future work, considering the memory usage of the algorithm might be helpful since the remove operation was executed two ways: physically removing if there are less than two children, and logical removing if there are two children. Furthermore, the rotational operation starts with making several duplicate nodes. Therefore, memory efficiency of the algorithm may

become one of the drawbacks of this algorithm. Additionally, it will be worthwhile to see the performance evaluation when implementing and experimenting the algorithm.

Bibliography

Afek, Yehuda, Korenfeld, Boris, and Morrison, Adam. n.d. Concurrent search tree by lazy splaying. Retrieved from http://www.cs.tau.ac.il/~afek/LazySplaying.pdf.

Afek, Yehuda, Kaplan, Haim, Korenfeld, Boris, Morrison, Adam, Tarjan, Robert E. 2014. The CB tree: a practical concurrent self-adjusting search tree. *Distributed Computing*, 27: 393-417.

Bronson, Nathan G., Casper, Jared, Chafi, Hassan, and Olukotun, Kunle. 2010. A practical concurrent binary search tree. PPoPP ’10, January 9-14, 2010. Bangalore, India.

Brown, Trevor, Ellen, Faith, and Ruppert, Eric. 2013. Pragmatic primitives for non-blocking data structures. PODC ’13, July 22-24, 2013, Montreal, Quebec, Canada. 13-22.

Brown, Trevor, Ellen, Faith, and Ruppert, Eric. 2014. A general technique for non-blocking trees. PPoPP ’14, February 15-19, 2014, Orlando, Florida, USA. 329-341.

Chatterjee, Bapi, Walulya, Ivan, and Tsigas, Philippas. 2016. Help-optimal and language-portable lock-free concurrent data structures. 2016 45th International Conference on Parallel Processing, August 16-19, 2016, Philadelphia, Pennsylvania, USA. 360- 369.

Crain, Tyler, Gramoli, Vincent, and Raynal, Michel. 2013. A contention-friendly binary search tree. In Euro-Par 2013: Euro-Par 2013 Parallel Processing, Lecture Notes in Computer Science, F. Wolf, B. Mohr, and D. an Mey (Eds). Springer, Berlin, Heidelberg. Vol. 8097, 229-240.

Drachsler, Dana, Vechev, Martin, and Yahav, Eran. 2014. Practical concurrent binary search trees via logical ordering. PPoPP ’14, February 15-19, 2014, Orlando, Florida, USA. 343-356.

Ellen, Faith, Fatourou, Panagiota, Helga, Joanna, and Ruppert, Eric. 2014. The amortized complexity of non-blocking binary search trees. PODC ’14, July 15-18, 2014, Paris, France. 332-340.

Ellen, Faith, Fatourou, Panagiota, Ruppert, Eric, and van Breugel, Franck. 2010. Non-blocking binary search trees. PODC ’10, July 25-28, 2010, Zurich, Switzerland.

Herlihy, Maurice and Shavit, Nir. 2008. The art of multiprocessor programming. Burlington, Massachusetts: Morgan Kaufmann Publishers.

Howley, Ahane V., and Jones, Jeremy. 2012. A non-blocking internal binary search tree. SPAA ’12, June 25-27, 2012, Pittsburgh, Pennsylvania, USA. 161-171.

Moir, Mark and Shavit. Nir. 2004. Concurrent Data Structures. In Handbook of Data Structures and Applications, D. Metha and S.Sahni (Eds). Chapman and Hall/CRC Press, Chapter 47, 1-30.

Mueller, Scott. 2006. Upgrading and repairing PCs (17th ed.). Indianapolis, Indiana. Que Publishing.

Natarajan, Aravind, and Mittal, Neeraj. 2014. Fast concurrent lock-free binary search trees. PPoPP ’14, February 15-19, 2014, Orlando, Florida, USA. 317-328.

Pacheco, Peter S. 2011. An introduction to parallel programming. Burlington, Massachusetts, Morgan Kaufmann Publishers.

Ramachandran, Arunmoezhi, and Mittal, Neeraj. 2015. A fast lock-free internal binary search tree. ICDCN ’15, January 04-07, 2015, Goa, India. Article No. 37.

Sleator, Daniel D. and Tarjan, Robert E. 1985. Self-adjusting binary search trees. *Journal of the Association for Computing Machinery*, 32(3): 652-686.

Stallings, William. 2015. Computer organization and architecture: designing for performance (10th ed.). Hoboken, New Jersey: Pearson Education, Inc.

Curriculum Vitae

Sachiko Sueki

Department of Computer Science
University of Nevada, Las Vegas
Email: suekis@unlv.nevada.edu

Education

University of Nevada, Las Vegas
M.S. in Computer Science, December 2017

Thesis Title: Lock-Free Self-Adjusting Binary Search Tree

Thesis Examination Committee:

Chairperson, Dr. Ajoy K. Datta
Committee Member, Dr. John Minor
Committee Member, Dr. Wolfgang Bein
Committee Member, Dr. Kumud Acharya
Graduate Faculty Representative, Dr. Emma E. Regentova