5-1-2014

Self Adjusting Contention Friendly Concurrent Binary Search Tree by Lazy Splaying

Mahesh Raj Regmee
University of Nevada, Las Vegas, regmeem@unlv.nevada.edu

SELF ADJUSTING CONTENTION FRIENDLY CONCURRENT BINARY SEARCH TREE BY LAZY SPLAYING

by

Mahesh R Regmee

Bachelor of Engineering (Computer)
Tribhuwan University,IOE, Pulchowk Campus, Nepal
2008

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
May 2014

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Mahesh R. Regmee

entitled

Self Adjusting Contention Friendly Concurrent Binary Search Tree by Lazy Splaying

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

Master of Science in Computer Science
Department of Computer Science

Ajoy K. Datta, Ph.D., Committee Chair

Lawrence Larmore, Ph.D., Committee Member

Yoohwan Kim, Ph.D., Committee Member

Emma Regentova, Ph.D., Graduate College Representative

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

May 2014

Abstract

We present a partial blocking implementation of concurrent binary search tree data structure that is contention friendly, fast, and scales well. It uses a technique, called lazy splaying to move frequently accessed items close to the root without making the root of the tree a sequential bottleneck. Most of the self adjusting binary search trees are constrained to guarantee the height of a tree even in the presence of concurrency. But, this methodology roughly guarantees the height of a tree only in the absence of contention and limits the contention during concurrent accesses.

The main idea is to divide the update operation into two operations: *an eager abstract modification with lazy splaying* that completes quickly and makes at most one local rotation of the tree on each access as a function of historical access frequencies; and *a lazy structural adaptation with long/semi splaying* which implements top down recursive splaying of the tree that may be postponed to diminish contention and re-balance the tree during less contention. This way, the frequently accessed items perform full splaying but after a few accesses only and will always appear near the root of the tree. Whereas, the infrequently accessed items will not get enough pushes up the tree and stay in the bottom part of the tree.

As in sequential counting based splay tree, the amortized time bound of each operation is *O(log N)*, where N is the number of items in the tree.

Acknowledgements

I would like to express my sincere gratitude to my thesis supervisor Dr. Ajoy K. Datta for motivating and mentoring me to work in such an important topic in the field of multi-core computing –the cutting edge technology– of this time. I would also like to thank him for showing me the various research areas in the field of distributed, multi-core computing. His guidance and support from the very beginning helped me in staying on the right track and complete the work on time.

I would also like to thank my committee members Dr. Emma E. Regentova, Dr. Lawrence Larmore and Dr. Yoohwan Kim for their support and I am grateful to have them in my thesis committee.

I would like to remember my friends and family for their moral support.

Mahesh R Regmee

University of Nevada, Las Vegas
May 2014

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

With the advancement of multi-core computers and disappearance of unicore computers, multi threaded programs are not just an option but the necessity of this period. Multicore computers make computing more effective by exploiting parallelism and utilizing all the cores to work on a single task. To support multi threaded programs, we are in need of data structure that supports multicore computers similar to the conventional data structures. The data structures that supports multithreading and synchronization among the threads is now called the concurrent data structures.

It is extermely difficult to design concurrent data structures compared to sequential ones because threads executing concurrently may interleave their steps in many ways resulting different and potentially unexpected outcome. The concurrency makes the algorithm of data structures more complex and less efficient. Multicore processors are widely being used in every processor dependent devices these days, from high end devices to low end devices like smart phones, tablets etc. That directly leads to the necessity of desigining an efficient algorithms of concurrent data structures. Also, there is a challenge in designing scalable concurrent data structures that should continue supporting as the number of concurrent threads keeps on increasing more and more.

One of the widely used search data structure in sequential context is binary-search-tree (BST) and it provides logarithmic time complexity for all the operations provided the tree is balanced. In the sequential context and on skewed accessed sequences, it has been accepeted that the self-adjusting binary search trees (BST) like Splay tree are theorertically better than balanced BST like AVL and red-black tree. But in practice, red-black tree and AVL tree outperform even on skewed sequences [1]. This is because of the overhead associated with self-adjustment. To maintain the tree balanced, upon tree update that breaks the structural invariant of the BST, the rebalancing is immediately done to restructure the tree to make it balance. In the concurrent context, this immediate rebalancing

produces lots of contention though slightly weakened balance requirement have been suggested [2] that still requires immediate restructuring.

1.2 Objective

Decoupling the update operations that modify the abstraction from operations that modify the tree structure itself have been been suggested[3] to reduce the overhead of contention trading off with restucutring requirement. Also, Technique of *lazy splaying* to balance the tree in concurrent context have been studied [4] that have been claimed to be the efficient self-adjusting binary search tree. We propose the new *self-adjusting concurrent contention-friendly binary search tree* in which we borrow the similar ideas from above two approaches and make some changes to have the advantages of both. We segregate the restructuring requirement and abstract modification requirement to reduce contention, speedup concurrent operations of the BST and yet producing self-adjusting BST. It is surprising that the known self-adjusting BST all modify the root with each access making it an immediate sequential bottleneck. Our approach minimizes this bottleneck by incorporating splaying technique on the tree in lazy manner.

Binary search tree stores an ordered set of items that supports insert, contains and delete operations and additional operations like range operations and predecessor/successor operations. The balanced tree guarantees the time-complexity of each operation to be bounded by *O(logN)*, where N is the number of itmes in the tree. The average and worst case time complexity remains same if the probability of each item being accessed is uniform. In practice, as per principal of locality, most access sequences are non-uniform, *e.g.* 80% of the accesses are to 20% of the items [5] [6]. Various self-adjusting BSTs have been suggested that moves the frequently accessed items towards the root or the tree that gives the better average access time over many access of the items. The widely accepted such BST is splay-tree that brings the accessed item, either by insert or contains operation towards root with the help of technique called splaying by multiple rotations. So, the frequently accessed items tends to be around the root of the tree and will be found faster in subsequence accesses. Since almost all of the operations modifies the root, the root becomes the hot *sequential bottleneck* resulting in non-scalable structure.

So, we will use the technique called *Lazy splaying* that performs at most on local tree rotation *i.e.* re-structuring per access as a function of historical frequencies as a subroutine of the *eager-abstract modification* technique. With this approach, those items that are accesed very frequently will get fully splayed but over few accesses and will appear higher up in the tree, on the other hand infre-

quently accessed items will get enough pushes down the tree and will remain at the bottom part of the tree.

Our approach uses a partially internal binary search tree data structure that implements a *key-value* store, that decouples the update operations into *abstract operations with lazy splaying* and *structural operations*. *Abstract operations with lazy splaying* is responsible either to search for, logically delete or insert an item to the tree and does at most one local tree rotation. Whereas, *Structural operations* rebalance the tree by executing *longsplaying, semisplaying* that may cause multiple rotations as well as physically removing nodes that have been deleted logically.

Moreover, as the re-structuring of the tree creates the accessive contention, separating this process from abstract modification produces less contention during concurrency and the lazy splaying resolves the standard splaying root-bottleneck, making this technique efficient, scalabale and highly concurrent BST. Modification and rotate operations are to be done with proper locking and in a consistent way. However counting the total no. of operations in different parts of the tree can be done asynchronously withoug using lock. This may create a synchronization issue on these counters but is tolerated as the inaccuracies in these counters will have negligible effect on the performance and have been extensively examined experimentally [7]. So, this eliminates the extra locking operations necessary to synchornize counters and improves the perfomance significantly.

1.3 Outline

In chapter 1, we briefly discussed the reason behind choosing this particular area as a thesis research topic. We then discussed about objective of the research topic.

In chapter 2, we give brief overview of *concurrent vs sequential* programming and discuss why they are important in the multi-core era. We will then explain about concurrent data structures. In the various subsections under this section, we will explain how those data structures are designed, how are they classified, and how are they verified to check if they are implemented correctly. As, we will use some idea from regular sequential splay tree, we will briefly discuss about it.

In chapter 3, we will go over various existing implementation ideas that are related in some way to our implementation. We will give the short implementation of the algorithms presented on those papers that we have researched during our implementation. We will see their suitable usage and discuss about their pros and cons. Our main focus will be in the *lock-based* implementaion of binary

search trees and the techniques used to self balance those trees. Many papers related to contention friendliness have been researched.

In chapter 4, we will give the overview of our approach. We will discuss all the fundamentals that are needed in our implementation. These includes, lazy splaying, eager-abstract modification, lazy-structural adaptation. We will explain briefly about those techinques and how they are used in our implementation.

In chapter 5, we first introduce the data-structure that are needed in our implementation. The data structure includes the node of the tree, the various fields that are needed for the node and the binary-search tree that we will be referring in our implementation. we will then give the pseudocode algorithm of all the operations involved. To make the working of those algorithms clear, we will give some figures too that shows the working of the operation.

In chapter 6, we conclude our implementation and give some ideas that can be done as future work.

Chapter 2

Background

2.1 Multi Processor Programming

In computer technology, *Moore's* law has been existed for almost 50 years and it is still valid. Though, more and more transistors can be packed into a single chip, the clock speed can't be increased because of overheating. This is the reason behind the development of multi-processor and multi-core architecture in order to speedup modern computers. These computers are capable of handling operations at the same time in hardware level increasing performance of the computer. With the invention of this technology, parallelism in both hardware and software program are developed. Parallelism in software program is also called concurrency. Hardware and software program both executes the threads simultaneously in a safe manner in the physical cores or logical cores of the machine.

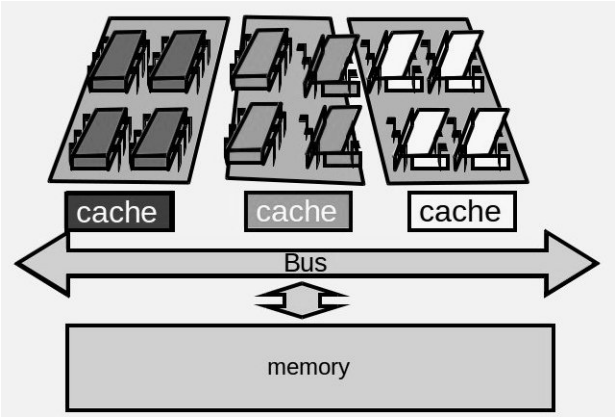

Figure 2.1: The shared memory architecture

Shared-Memory multi processors systems are the systems in which the multiple threads or processors executes concurrently and communication between threads are done via shared memory. Fig. 2.1[8] shows the shared memory architecture. Based on the location of shared memory there exists SMP

symmetric multiprocessor and NUMA *non-uniform memory access* architectures. In an SMP architecture, both processors and memory hang off a bus. This works well for small-scale systems. In a NUMA architecture, each processor has its own piece of the memory. Accessing your own memory is relatively fast, and accessing someone elses is slower. Usually NUMA machines also have caches, in which case they are called CC-NUMA machines, for cache-coherent NUMA [8]. The figures for these architecures are shown in Fig. 2.2[8].

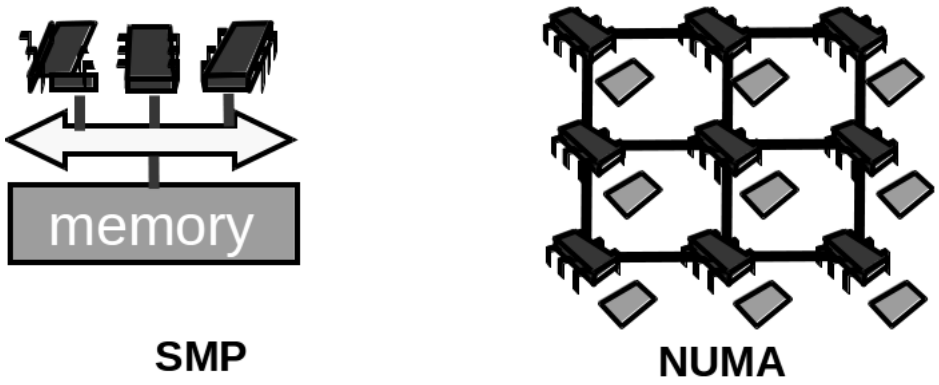

Figure 2.2: The SMP and CC-NUMA architecture

2.1.1 Sequential and Parallel Programming

A *process* is an instance of a program running in a computer. It is the basic entity that can be executed in a computer. *Thread* is contained inside a process and same process can have multiple threads in it. A computer program contains several processes and each process may have multiple threads. In general, given the same input data, a sequential program will always execute the same sequence of processes and each process is associated with a single thread of control and it will always produce the same results and its execution is deterministic. Sequential program can only execute one task at a time. Fig. 2.3 shows the sequential execution of the program.

These days almost every computer supports processes with multiple threads within a single process itself. Threads shares the same address space within that process. These multiple threads in a process can be executed at the same time in multi-processor or multi-core architecture and is called parallel execution of the threads. This allows the flexibility of doing multiple tasks at once. This greatly enhances the performance of operating system and efficient use of CPU. However, programming in such architecture is difficult and requires special attention and knowledge. It may sometimes causes waste of resources if program is inefficiently written. Parallel program exectues as shown in Fig. 2.3

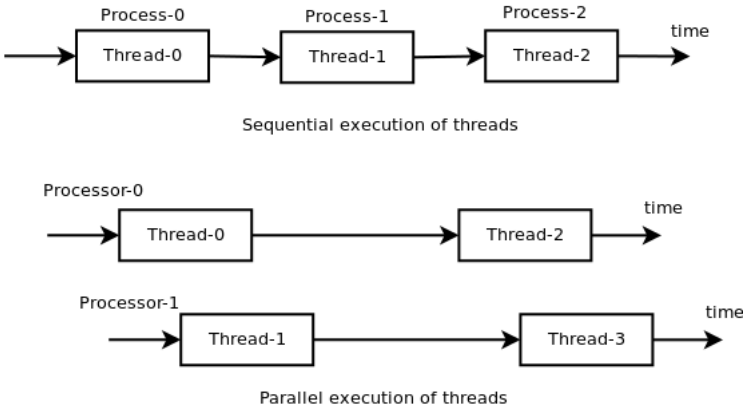

Figure 2.3: Sequential and parallel execution of threads

2.1.2 Concurrent Programming

Concurrent programming is the one in which several computations are executing during overlapping time periods concurrently instead of sequentially (one completes before the next starts) [9]. If the multiple software threads running at different cores tries to access same memory or some shared resources, concurrency arises. In single CPU system the concurrency is observed only logically. They use the time-sharing technique to share the same CPU within multiple threads.(e.g. Fig. 2.3 shown time sharing). So, pure concurrent programming is actually achieved by software not the hardware. However, parallelism in hardware like multi-processor system or multi-core systems are more efficient than single processor system in terms of concurrency.

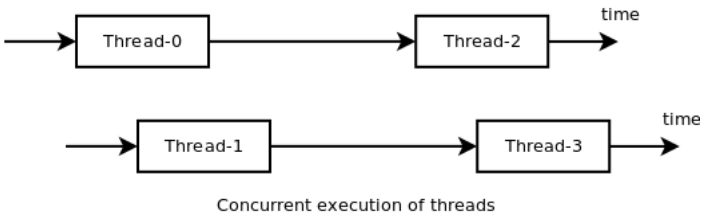

Figure 2.4: Concurrent execution of threads

2.2 Concurrent Data Structures

Concurrent data structure allows concurrent threads to store, organize and access the data on a shared memory system. It is not as simple as sequential data structure to design because the concurrent threads are executing asynchronously and may interleave their steps in many ways, many of them may not be desirable and produces unexpected outcome. In this section, we will discuss the challenges involved in designing concurrent data structures, the correctness criteria and various implementation techniques.

2.2.1 Designing Concurrent Data Structures

Designing concurrent data structures are significantly more difficult and its verification to check correctness is even more difficult than their sequential counterparts. The primary source of this additional difficulty is concurrency, exacerbated by the fact that threads must be thought of as being completely asynchronous: they are subject to operating system preemption, page faults, interrupts, and so on. This is because of the several features of shared-memory multi processors system at hardware level. On today's machines, the layout of processors and memory, the layout of data in memory, the communication load on the various elements of the multiprocessor architecture all influence performance. Furthermore, there is a tension between correctness and performance: algorithmic enhancements that seek to improve performance often make it more difficult to design and verify a correct data structure implementation.

2.2.2 Performance

The *speedup* of an application when run on P processors is the ratio of its execution time on a single processor to its execution time on P processors. This measures the utilization of the machine by the running application. We want *linear speedup i.e*, we would like to achieve a speedup of P when using P processors. If the data structure's speedup grows with P, we call it scalable.

In practice, we can't make all the threads execute every-time utilizing all the cores. There may be a chance at most one thread executes for some small period and we call this sequential bottleneck (all other threads remains idle for that period of time). And, such sequential bottlenecks can have a surprising effect on the *speedup* one can achieve.

Amdahls Law [10] characterizes the maximum *speedup* S that can be achieved by n processors collaborating on an application, where p is the fraction of the job that can be executed in parallel. Assume, for simplicity, that it takes (*normalized*) time *1* for a single processor to complete the job. With n concurrent processors, the parallel part takes time p/n and the sequential part takes time $1 - p$. Overall, the parallelized computation takes time: $1 - p + p/n$. *Amdahls Law* says that the speedup, that is, the ratio between the sequential (single-processor) time and the parallel time, is: $S = 1/(1 - p + p/n)$. This implies that if just 10% of our application is subject to sequential bottleneck, the best possible speedup we can achieve on a 10-way machine is about *5.3*: we are running the application at half of the machine's capacity. This means, reducing the number and length of sequentially executed code sections is thus crucial to performance and in the context of locking, reducing the number of locks acquired, and reducing lock granularity, a measure of the

number of instructions executed while holding a lock.

2.2.3 Blocking and Non Blocking Techniques

Shavit and Moir discussed about some standard techniques in designing such concurrent data structures [10]. Mainly there are two broad categories of concurrent algorithms *viz Blocking* and *Non Blocking*. Each of these techniques have their own pros and cons.

Blocking technique uses lock to prevent race conditions in the shared part of the data structure. In blocking technique also there exists two kinds of locking *Coarse-grained* and *Fine-grained* locking depending on granularity of the lock.

In Coarse-grained locking, single lock of large granularity is used to protect the entire data structure. One thread runs until it is blocked by an event that normally would create a long latency stall. Such a stall might be a cache-miss that has to access off-chip memory, which might take hundreds of CPU cycles for the data to return. Instead of waiting for the stall to resolve, a threaded processor would switch execution to another thread that was ready to run. Only when the data for the previous thread had arrived, would the previous thread be placed back on the list of ready-to-run threads [11]. This technique however have mainly three problems [10] *sequential bottleneck* , *memory contention* and *progress delay*.

In Fine-grained locking multiple locks of small granularity is used to protect different parts of the data structure to allow concurrent operations to proceed in parallel when they do not access the same parts of the data structure. This technique is widely used to avoid excessive contention for individual memory locations. It is also not and ideal choice as it may suffer from problems like unnecessary memory traffic due to local spinning, false sharing and deadlocks.

Non Blocking technique tries to overcome the various problem associated with the blocking technique. There exists various nonblocking progress conditions-such as wait-freedom [12] [13], lock-freedom [12], and obstruction-freedom [14].

Wait-freedom is the strongest non-blocking guarantee of progress, combining guaranteed system-wide throughput with starvation-freedom. An algorithm is wait-free if every operation has a bound on the number of steps the algorithm will take before the operation completes.

Lock-freedom allows individual threads to starve but guarantees system-wide throughput. An algorithm is lock-free if it satisfies that when the program threads are run sufficiently long at least one of the threads makes progress. All wait-free algorithms are lock-free.

Obstruction-freedom is the weakest natural non-blocking progress guarantee. An algorithm is obstruction-free if at any point, a single thread executed in isolation (*i.e.*, with all obstructing threads suspended) for a bounded number of steps will complete its operation. All lock-free algorithms are obstruction-free.

Among the progress conditions, stronger progress condition seems always desirable, but implementation of such condition in designing and verifying its correctness is difficult than the weaker progress conditions. In practice weaker progress conditions are tolerated by employing a technique called back off [15] to make the implementation simple and easier.

2.2.4 Correctness

For a sequential data structure to be correct it has to satisfy safety condition and should agree with the legal sequential specification of that object. For, a concurrent data structure to be correct, safety condition only is not good enough, it has to satisfy certain liveness condition as well. However, safety condition in concurrent data structure is not as straight forward as in sequential data structure. So, following are the various correctness properties defined for concurrent data structure.

Quiescent Consistency

Given a concurrent execution history of an object, if all operations appear to occur in some sequential order and nonoverlapping operations appear to occur in real-time order then the object is called quiescently consistent. We should assume, each operation accesses a single object. Operations not separated by quiescence may not occur in program order. *E.g.*, *A* enqueue x and then y; our dequeue operation overlaps both enqueues, and we come out with y.

Sequential Consistency

Given a concurrent execution history of an object, if all operations appear to occur in some sequential order and the order is consistent with each thread's program order then the object is called sequentially consistent. It requires that method calls act as if they occured in sequential order consistent with program order. We should assume, each operation accesses a single object.

Linearizability

Given a concurrent execution history of an object,if the system is sequentially consistent and the sequential order is consistent with real time; *i.e.*, all operations appear to happen between their invocation and response then the object is called linearizability. We should assume each operation accesses a single object.

2.3 Splay Tree

A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. Because the shape of a BST is determined by the order that data is inserted, we run the risk of trees that are essentially lists in the worst case. Worst case for a single BST operation is *O(N)*. It is not so bad if this happens only occasionally, but its not uncommon for an entire sequence of bad operations to occur. In this case, a sequence of M operations take *O(M * N)* time and the time for the sequence of operations becomes noticeable. Splay trees guarantee that a sequence of M operations takes at most *O(M * log N)* time. We say that the splay tree has amortized running time of *O(log N)* cost per operation. Over a long sequence of operations, some may take more than *log N* time, some will take less.

The basic idea of the splay tree is that every time a node is accessed, it is pushed to the root by a series of tree rotations. This series of tree rotations is knowing as *splaying*.

If the node being splayed is deep, many nodes on the path to that node are also deep and by restructuring the tree, we make access to all of those nodes cheaper in the future.

Chapter 3

Literature Review

Lots of big computer organization are pushing themselves towards multi processors and multi-core architecture. To support the growing need of those companies plenty of research are being done in the field of concurrent data structure. Most of the research are focused in making the concurrent version of sequential common data structure like stack, queue, linked list and binary search trees. The concurrency creates lots of technical and implementation difficulties in designing the concurrent data structure due to bad interleaving of the steps of various operations.

Implementation of various concurrent search structures based on all the techniques discussed above in section 2.2.3 has been done in the literature.

3.1 Universal Transformation

Herlihy [16] has a translation protocol that takes as input the sequential algorithm and produces equivalent non-blocking concurrent algorithm. His first transformation does the copying of the entire object, making necessary changes to it and trying to replace the old object by CAS operation. "It compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail" [17].

For large objects, copying the whole object made this an impractical solution. He then proposed another transformation protocol. Each data structure is made up of blocks connected by pointers. Only the blocks which are modified or contain the pointers to the blocks which are to be modified need to be copied. Again it has some drawbacks, first there was still a lot of copying, second programmer had to go extra mile to break the structure into proper blocks and third for some data structure like priority queue implemented as linked list no decomposition performs well.

3.2 Stacks and Queues

Stacks and Queues are the fundamental sequential data structure. However, there arises plenty of issues in designing concurrent version of these data structures that clearly illustrates the challenges encountered in designing concurrent data structures that supports multi-processor shared memory system.

A concurrent stack is a data structure linearizable to a sequential stack that provides push and pop operations with the usual LIFO semantics.

Treiber [18] introduced a lock-free implementation of stack. He used the singly-linked list to represent the stack with top pointer and used CAS atomic instrucution to modify the value of top pointer atomically. However, top pointer is a sequential bottleneck in this approach so it suffers from scalability problem as concurrency increases. Importantly, this implementation will have ABA problem [19] that plagues many CAS-based algorithm.

Michael and Scott presented a simple lock-based concurrent stack implementation based on sequential linked lists with a top pointer that uses global lock to control access to the stack [20].

A concurrent queue is a data structure linearizable to a sequential queue that provides enqueue and dequeue operations with the usual FIFO semantics.

Michael and Scott [19] presented a simple lock-based queue implementation based on sequential linked list and that uses two separate locks for the head and tail pointers. This will allow execution of enqueue and dequeue operations parallel.

Herlihy and Wing [21] proposed a lock-free array-based queue that works if one assumes an unbounded size array.

3.3 Search Structure

Data structures that allows the efficient retrieval of an element from the set of elements are called search structures. This includes unordered list, array, hash-table, binary search trees etc. Due to the excessive use of these data structures in the computer industry these days, lots of researches are being carried on in implementing the efficient concurrent search structures. Binary Search Tree

(BST) being the most prevalent one.

3.3.1 Binary Search Tree

A binary search tree is a data structure that supports three main operations, *insert(k)*, *delete(k)* and *find(k)*. Concurrent implementation of any search tree can be done by protecting it using a single exclusive lock (global lock). In this implementation, concurrency can be improved by using reader-writer lock to allow the *find(k)* operation to execute concurrently with each other while holding the lock in shared mode. This may be inefficient if there are many update operations. The exclusive lock does produce *sequential bottleneck* that degrades the overall performance substantially. If fine-grained locking technique is used that employs one lock per node instead of single lock for entire tree, we can improve concurrency further.

There exists various self-adjusting binary search trees (BSTs) that restructures itself based on certain conditions to make the overall operations efficient. The restructuring is done using tree rotations that does not effect the BST's property. AVL tree is a balanced binary search tree which triggers the rebalance operation once the length of longest path exceeds the shortest path by *2* [22]. A red-black tree however triggers the rebalance operation only when the length of longest path exceeds the shortest path by two times [23].

The splay tree is also a self-adjusting BST that moves the accessed node towards the root by rotation, which is called splaying. Rotation is done based on the position of the involved nodes: currently accessed node, its parent node and its grand parent node. Unlike self-balancing trees like AVL tree or Red-Black tree, splay tree doesn't self balance itself. CBTree (Counting based tree) is a variation of splay tree that moves the node towards the root based on the number of times the node is accessed by *insert* and *find* operation.

Bronson *et al.* presents a concurrent relaxed balance AVL tree algorithm that is fast, scales well, and tolerate contention [24]. Implementing a full rebalancing is generally a bottleneck in a concurrent tree. Indeed, it must acquires locks for all nodes that will be rebalanced. That is why the author decided to use a relaxed balance AVL Tree instead of a strict one. In a relaxed balanced tree, the condition can be violated by the operations and is not always restored by rebalancing, but may be. It introduces the concept of version numbers and a hand-over-hand optimistic validation to achieve mutual exclusion when different thread intend to re-balance the tree after insertions or deletions. It uses the optimistic concurrency control, but carefully manage the tree in such a way that all atomic

regions have fixed read and write sets that are known ahead of time. Hand-over-hand optimistic validation is a concurrency control mechanism for searching and navigating a binary search tree. Version number is the key to achieve mutual exclusion in the algorithm. Each node's version number represents a status of the node that is being accessed. The tree used in this algorithm has the particularity to be a partially external tree. An external tree is a tree in which all the information is only contained in leaves. An external tree has the advantage to make removal easy at the cost of increasing the average path length and the memory footprint. In a partially external tree, routing nodes (nodes with no information) are only inserted during removal operation. Moreover, routing nodes with only one child are removed during rebalancing. This makes removal easier and does not increase too much the average search path length. Another advantage of this technique is that a routing node can be converted to a value node and vice-versa only by changing its value.

It is difficult to design an efficient non-blocking data structure that guarantees wait-freedom. There exists universal technique to derive a concurrent wait-free data structure from its sequential version [25]. But the the tree from this transformation is quite inefficient because the universal construction will work either applying operations to the data structure in serial mannner or copying the entire data structure or part of it and applying the operation to the copy and then updating the relevant part of the data structure to point to the copy.

Ellen *et al.* proposes the first complete, non-blocking, lineraizable BST implementation using only reads, writes, and single-word compare-and-swap (CAS) operations [26]. It does not use large words, so it can be run directly on existing hardware. Updates to different parts of the tree do not interfere with one another, so they can run concurrently. Searches only read shared memory and follow tree edges from the root to a leaf so they do not interfer with updates, either. It uses leaf-oriented BST , in which every internal node has exactly two children, and all keys currently in the dictionary are stored in the leaves of the tree. Internal nodes of the tree are used to direct a *find* operation along the path to the correct leaf. Helping strategy is used similar to Barne's technique [27] to improve performance of the operations. Helping can often contribute to poor performance because several processes try to perform the same piece of work. Thus they choose a conservative heling stragety: a process P helps another process's operation only if the other operation is preventing P's own progress.

Natarajan, Savoie and Mittal suggested a wait-free implementaion of concurrent red-black tree in an asynchronous shared memory system that supports search, insert, update and delete operations us-

ing single-word compare-and-swap instruction [25]. Compared to other existing wait-free algorithm, this one has better properties that: it uses only single word CAS instruction, which is common in most of the hardware these days, it doesn't uses additional underlying system support such as transactional memory and it doesn't allow the tree go out of [28] for deriving a concurrent wait-free tree-based data structure from its sequential counterpart.

Raynal *et al.* proposed the non-blocking implementation of skip list [29] a new non-blocking skip list algorithm. This algorithm minimizes contention by localizing synchronization at the least contended part of the structure without altering consistency of the implemented abstractions. They proposed the first binary search tree algorithm designed for speculative executions [30]. In contention friendly BST implementation [3], they discussed the implementation of lock-based concurrent binary tree using some unique methodology called contention friendliness. All these three implementation uses the same concept of dividing update operations into two parts. One that modifies abstraction state of the tree that returns rapidly for efficiency reason and the other that modifies structural state that runs in the background. The key idea is to diminish the contention induced due to concurrency.

Afek *et al.* proposed the implementation of concurrent search tree by lazy splaying [7]. Though this technique can be used to replace re-balance operation of any binary search tree to make it balanced, they used the Bronson *et al* tree [24] and replaced the re-balancing code with the lazy splaying re-balancing technique. The main idea of this implementation is to eliminate the sequential hot spot at the root caused by various self adjustment operations involved. This is done by introducing lazy splaying because it is fast and highly scalable and makes at most one local adjustment to the tree on each access as function of historical frequencies (no. of accessed made to that particular node).

Afek, Tarjan and company presented the implementation of a practical concurrent self-adjusting search tree called CBTree (Counting Based Tree) [4] that scales with the amount of concurrency, and has performance guarantees similar to the splay tree. CBTree maintains a weight for each subtree S, equal to the total number of accesses to items in S. The CBTree uses the operations similar to splay trees, but rather than performing them at each node along the access path, decisions of where to rotate are based on the weights. The CBTree does rotations to guarantee that the weights along the access path decrease geometrically thus yielding a path of logarithmic length.

Chapter 4

Methodology

In this section, we will discuss the overall ideas of our implementation technique. In most of the self-adjusting implementation of concurrent binary search tree (BST) the contention is produced when there is an overhead of rotations induced by various operations that tends to access the same part of the tree towards the root. As long as property related to binary search tree is satisfied we can postpone the self-adjustment or restructuring of the tree in-order to minimize the contention during highly concurrent access of the tree. And when there is less contention the re-balancing operation can be performed. The most frequently used operation in any search structure including binary search tree is the *search* (or *contains*) operation, if we figure out someway to implement this operation as partial-blocking implementation then this operation itself makes the overall algorithm efficient. We will discuss the technique used to avoid contention during traversal of the tree that still preserves its correctness in later chapters. The main idea in contention friendly methodology is to separate each update into an eager abstract modification and a lazy structural adaptation. The supporting ideas includes lazy deletion (a technique in which the node to be deleted is logically deleted) and lazy splaying that rotates the tree locally at most once and helps in adjusting the tree structure bringing frequently accessed items towards the root.

4.1 Lazy Splaying

A binary search tree is a recursive data structure containing a set of items each with a value from a totally ordered domain. Each node n have a value associated with it $n.key$ such that, the left subtree of n holds items smaller than $n.key$ and the right subtree holds items larger than $n.key$. All the operations on BST starts by searching an item in the BST going down from the *root* an goes left or right at each node depending the searched item is smaller or larger than the item at n, respectively until reaching a node containing the item, or an empty subtree indicating the item is not in the tree. This will be the spot where new items can be inserted.

Similar to splay tree technique, the tree rotation is performed at the node containing the item we operate on, moving subtrees with more frequently accessed nodes one level up whereas moving less accessed subtrees one level down. The operations we perform in this technique are called *zig-zag* or *zig* operations as in regular splay tree. Unlike regular splay tree, splaying operations that involves other rotation like *zig-zig*, we perform only two operations *zig-zag* or *zig*. These rotations are also only performed if certain conditions are met at the particular accessed node. We maintain the counters in each node to keep track of the total number of operations performed to that particular node. So, based on the values of counters we perform *zig* or *zig-zag* operation.

As these operations are performed at most once and done locally this doesn't produces contention that much during concurrency. Also, we need to lock only few nodes and are taken in order from the parent to child and sometimes grand parent too which affects the tiny portion of the tree and will be local. we will discuss about these operations in detail later in Chapter 5.

4.2 Eager Abstract Modification

In most of the existing self-adjusting binary search tree, once the structure gets updated either by insertion or deletion, the structure is checked if it satisfies its height property (like different threshold for different trees e.g AVL, red-black, 2-3 tree) and the tree is structured accordingly and will be considered this as the part of the same operation. Though, the update operation effects only the smaller portion of tree this restructuring may affect globally and potentially conflicts with other concurrent updates resulting higher contentions [3].

So, our approach is to minimize this global modification caused by re-balance operation. This is done by returning eagerly as soon as modifications of the update operation that only cares the BST property with at most one local adjustment is done. This way the update operations by re-turing eagerly, each individual process can move on to the next operation prior to adapting structure.

However, we will introduce lazy splaying in the update operation itself if necessary that still doesn't effect the tree globally but gives the advantage of re-structuring and frequently accessed items are brought towards the root at least by one level.

This technique does not guarantee the big-oh complexity of access operations of BST as the tree is not structured as soon as update operation is performed but such complexity may not be of that much importance during concurrent executions.

The node that are to be deleted by delete operation are not deleted physically but marked as deleted and is called lazy deletion. As delete operation changes the structure of a tree and causes lots of contention, this technique of lazy deletion is implemented. This will have advantage in later insert operations too, those nodes that have been marked as deleted but not physically removed can be added back by simply un-marking the node if it happens to be the same node and will not produce contention due to insert operation.

4.3 Lazy Structural Adaptation

This technique mainly deals with the restructuring of the BST that was postponed during above operation. As this is done when there is no concurrency so it is called *lazy*. Separate background thread is dedicated to perform this operation. This operation has nothing to do with the BST property so it mainly involves multiple rotations of the tree to balance the tree. So, the lineraziablity of update operation of this algorithm doesn't depend on this technique but the abstract update operation [3].

There can be any re-balancing method inside this technique. As we are dealing with self-adjustment that balances the tree and brings the frequently accessed items towards the root, we will basically implement the technique called *"Semi and Long Splaying"* in it. We will discuss *"Long Splaying"* in next section.

The main advantage of postponing the re-structuring operation is to enable merging of multiple adaptation in one simplified step. So, many abstract modifications may induce single structural adaptation which in turn reduces the contentions produced by abstract modifications. Also, various adaptations may compensate each other as the combination of two restructuring can be idempotent *e.g.* left rotation and right rotation at the same node may lead back to initial state.

In addition to above task, this operation has one extra responsibility of deleting the node that were lazily deleted during delete operation if it is safe to do so and if the removal of node generates less contention. As, removing a node in a tree is expensive operation that requires locking and invalidating a larger portion of the structure. So, nodes that are lazily deleted and have at least one of their children as empty subtree are removed.

4.4 Top Down Splaying

Bottom-up splaying requires traversal from root to the node that is to be splayed, and then rotating back to the root. In other words, we need to make two tree traversals. We would like to eliminate one of these traversals.

Its very easy to do this, each time we follow a left link (from let us say, node x), then x and its right subtree are all >than the node which will eventually become the *root*. So, we save x and its right subtree in a separate tree, which we will call R. The symmetric case (following a right link) identifies subtrees which will become part of the new roots left subtree, which we will call L.

The three reorganization cases for Bottom Up Splay Trees were zig, zig-zig, and zig-zag. Top-Down Splay Trees use only 2 cases: zig and zig-zig. zig-zag is reduced to a zig, and either a second zig, or a zig-zig. Each of these cases are shown in Fig. 4.1, Fig. 4.2 and Fig. 4.3

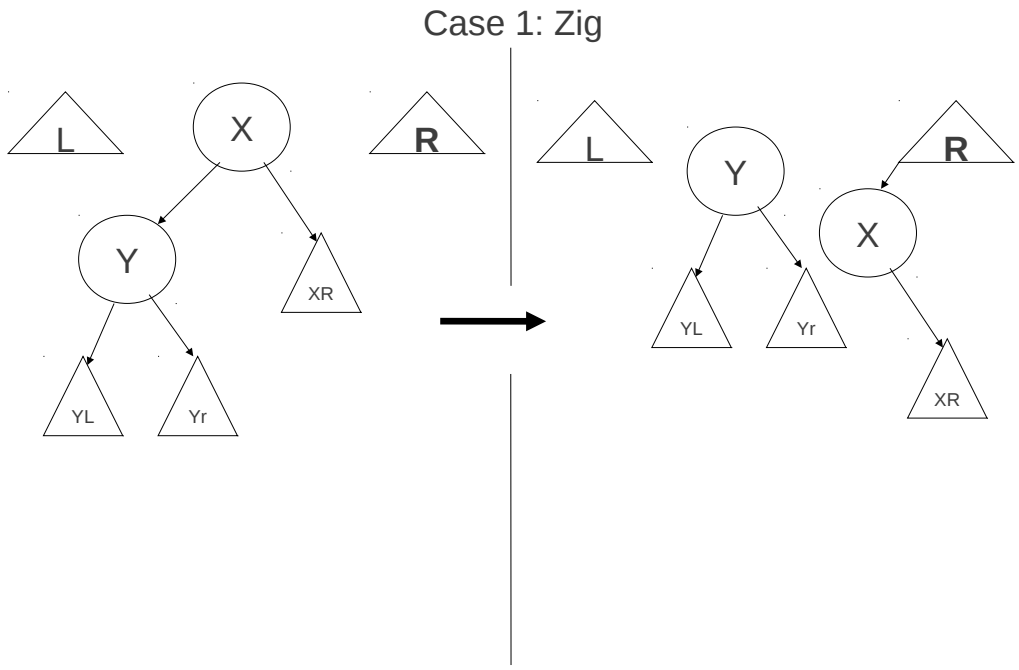

Figure 4.1: Zig case of Top-Down-Splaying. If Y should become root, then X and its right subtree are made left children of the smallest value in R, and Y is made root of center tree

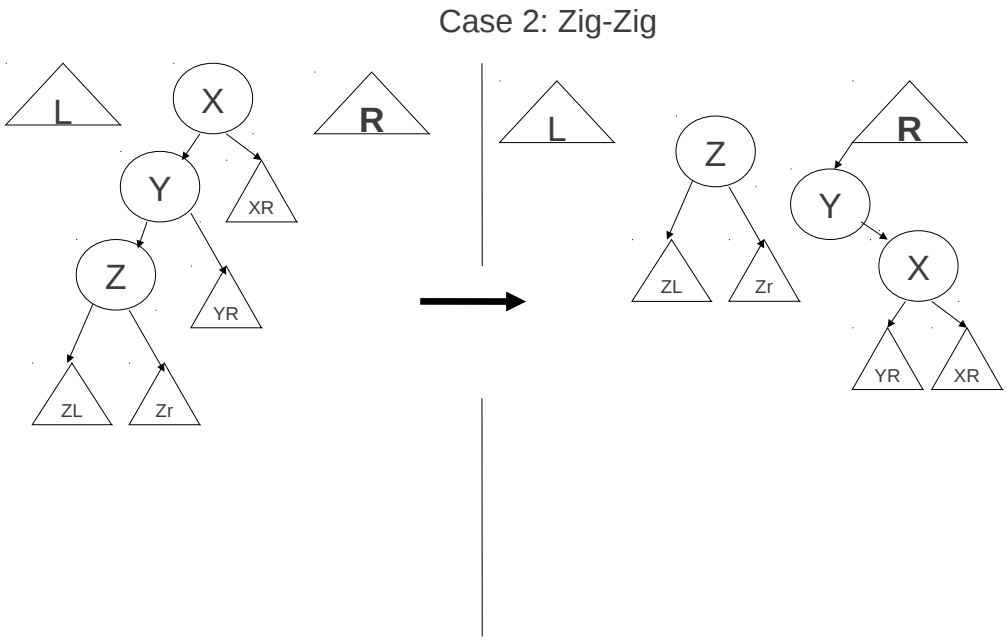

Figure 4.2: Zig-Zig case of Top-Down-Splaying. The value to be splayed is in the tree rooted at Z. Rotate Y about X and attach as left child of smallest value in R

4.5 Semi and Long Splaying

As mentioned before, lazy splaying uses the values of counter of each node that holds the number of operations performed to that particular node. But in concurrent fashion, the maintenance of counter itself is an overhead that needs synchronization and co-ordination. In order to get rid of this overhead the counter update is done asynchronously which is un-safe. This is because, the locks are not used during counter update and may suffer from race hazards. This incorrect values however will not impact the correctness of the operations and may only effect during re-structuring of the tree. But, it has been shown by the experiment that these inaccuracies have negligible effect on the performances [4].

4.5.1 Semi Splaying

A major drawback of regular splaying is the large amount of restructuring it does. Each splaying operation may induce rotations all the way to the root. We will now discuss the technique that reduce the amount of restructuring but preserve at least some of the properties of splay trees [31].

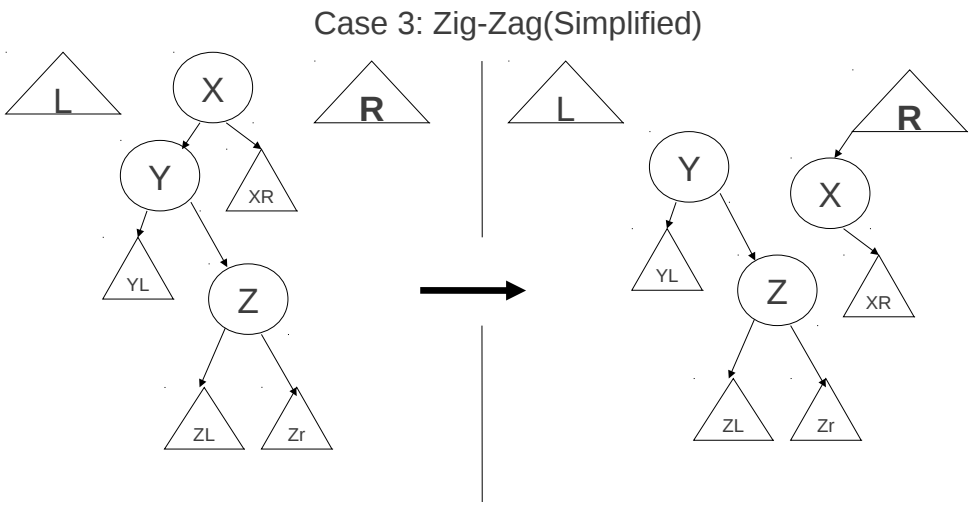

Figure 4.3: Zig-Zag case of Top-Down-Splaying. The value to be splayed is in the tree rooted at Z. To make code simpler, the Zig-Zag rotation is reduced to a single Zig. This results in more iterations in the splay process.

SemiSplaying technique modifies the restructuring rule of Splaying so that it rotates only some of the edges along an access path, thus moving the accessed node only partway toward the root. Semisplaying, differs from ordinary bottom-up splaying only in the *zig-zig* case: after rotating the edge joining the parent *p(x)* with the grandparent *g(x)* of the current node x, we do not rotate the edge joining x with *p(x)*), but instead continue the splaying from *p(x)* instead of x [31]. SemiSplaying steps are shown in Fig. 4.4.

Semisplaying operation is performed to reduce the depth of every node on the access path to at most about half of its previous value. Furthermore, only one rotation is performed in the zig-zag case, but two steps are taken up the tree. There are different variants of semisplaying, but we will be using top-down version of semisplaying technique.

As in topdown splaying, we maintain a left tree, a middle tree, and a right tree. In addition we

maintain a top tree and a node *top* in the top tree having a vacant child. The relationship among the trees is that all items in the left tree are less than the accessed item i and also less than those in the middle tree. All items in the right tree are greater than i and also greater than those in the middle tree. All items in the left, middle, and right trees fall between the item in *top* and the item in its predecessor in the top tree if the vacant child of top is its left, or between the item in top and the item in its successor in the top tree if the vacant child of top is its right. Initially the left, right, and top trees are empty and the middle tree is the entire original tree.

Let i be the item to be accessed. Each splaying step requires looking down two steps in the middle tree from the root and restructuring the four trees according to whether these steps are to the left or to the right. If i is in the *root* of the middle tree, we combine the left, middle, and right trees as in the completion of top-down splaying and then make the root of the combined tree (which contains i) a child of top, filling its vacancy. This completes the splaying. On the other hand, if i is not in the *root* of the middle tree, we carry out a *zig*, *zig-zig*, or *zig-zag* step as appropriate.

The *zig* and *zig-zag* cases are exactly as in topdown splaying. They do not affect the top tree. The *zig-zig* case is as illustrated in Fig. 4.5. Suppose that the access path to i contains the root x of the middle tree, its left child y, and the left child of y, say z. We perform a right rotation on the edge joining x and y. Then we assemble all four trees as in the terminating case, making node y (now the root of the middle tree) a child of top and making the left and right trees the left and right subtrees of y. Finally, we break the link between z and its new parent, making the subtree rooted at z the new middle tree, the rest of the tree the new top tree, and the old parent of z the new top. The left and right trees are reinitialized to be empty.

4.5.2 Long Splaying

As the counter are updated in asynchronous and un-safe manner, the height of tree may potentially go larger than *2*LogN*. To avoid this, We will add a safety belt watch-dog check in the code of each operation, that checks if a node at depth larger than *2*LogN* is reached then full semisplaying is performed. Though this happens rarely, this is done as a measure of protection and guarantee on worst- case per operation performance.

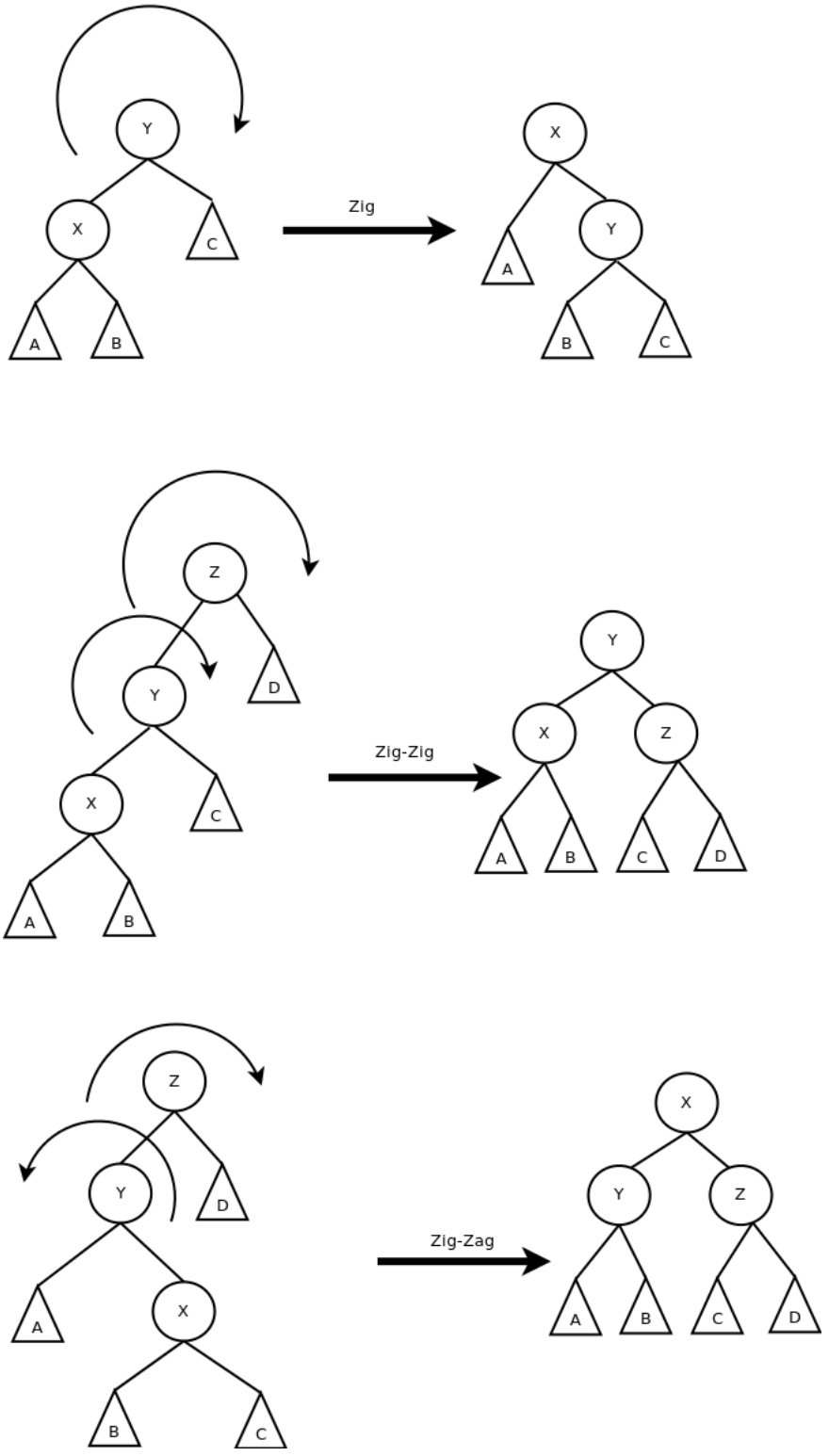

Figure 4.4: Semi-Splaying. Node x is the current node of splaying

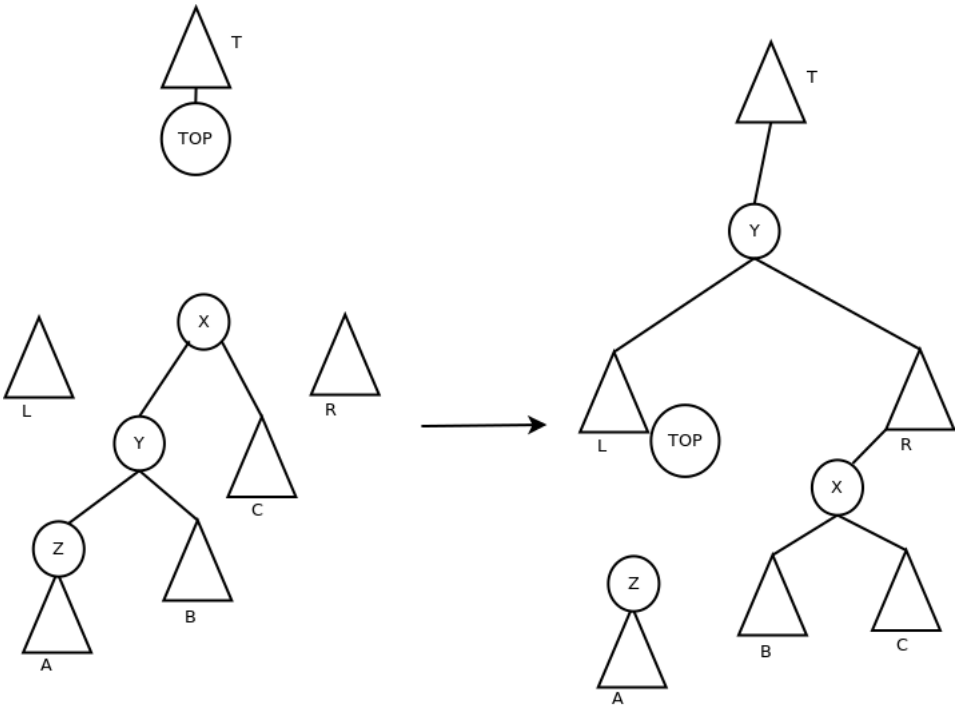

Figure 4.5: Zig-Zig case of Top-Down-SemiSplaying.

Chapter 5

Proposed BST Algorithm

We discussed the overview of techniques and functions used in our approach in previous section. In this chapter, we will mainly discuss the detailed implementation of all those operations and design the model of our implementation. These includes, the data structure used to represent a tree, node representation and its attributes, the locking technique used. Finally, we will write detail algorithms to all the operations of abstract and structural categories. We need some helper function in-order to complete the main operations and we will give algorithm to those functions as well.

5.1 Data Structures

We are designing a partial lock-based concurrent binary search tree that implements all the classic insert/delete/find operations. We mean partial because the traversal or search operation doesn't use lock. The tree is designed as a map object that supports remove and rotate operations, re-structuring operations and abstract operations. we will present the pseudo code to those operations later.

5.1.1 Node

Each node contains the following fields: a key k, pointers l and r to point to the node's left and right child, a *lock* field to lock the node when needed, a *delete* flag for lazy deletion, a *remove* flag that indicates if the node is physically removed, *selfCnt* which in an estimate on the total number of operations that has been performed on the node x, *rightCnt* and *leftCnt* which are an estimate on the total number of operations that have been performed on items in the right and left sub-trees of node x respectively.

5.1.2 Binary Search Tree-BST

In concurrent environment, binary search tree are broadly classified as either internal or external. In internal trees, key-value are stored at every node, while In external trees, values are only stored in leaf

nodes and the non-leaf nodes are referred as routing nodes, each of these has two children. External trees are mostly used for non-blocking implementation as both the delete and insert operations effect the leaf nodes and routing nodes are manipulated accordingly. We use regular internal binary search tree whose implementation is simple and to make the remove operation easy, we use lazy deletion technique.

5.2 Operations

We will now discuss the various operations that our BST supports. We call our algorithm a partial blocking implementation of concurrent BST. This is because, some operations are non-blocking and some are blocking. Those operations that are frequent and occurs concurrently that generates contention are made non-blocking. As every operation needs tree traversal and it is the longest part of the operation, we will make the traversal contention friendly. Overall operations are broadly classified into three type. They are contention friendly operations, background re-structuring operations and basic abstract operations.

5.2.1 Contention Friendly Operations

Basic abstract operations like, insert, delete involves the traversal operation to find the proper location or the node in the tree to apply those operations. Once the location is identified only a single node is effected. In case of delete just flagging the node to indicate the node is deleted is done. In case of insert proper child pointer update to point to new node (or if it happens to be the lazily deleted node, unflagging that node) is performed. Also, the lazy splaying operations requires to lock few neighboring nodes in order to perform local rotations like zig and zig-zag rotations which may effect the parallel traversal initiated by other threads. There exists various technique to obtain synchronization during traversal such as hand-over-hand locking that ensures the traversal is in track during concurrent rotations [24] or optimistic strategy using transactional memory approach and validating the traversal and if not valid re-starting the traversal [8].

We will use the same technique as Raynal described in contentional friendly BST [3] that modifies some rotation operations to make the traversal contention friendly even during lazy splaying process. Following is the list of different operations:

1. **removeNode**

   As we are implementing concurrent internal binary search tree that holds key-values at each internal nodes as well. The delete operation is overhead and requires precaution that may

need to lock many nodes in order to ensure the other concurrent traversal is in right track. Instead, we will just the flag the node as deleted allowing rest of the operation proceed via that node. The physical removal of nodes that has already been flagged deleted are done once in a while by background thread if it is safe to do so and it produces less contention. Those nodes that are lazily deleted and have only one or no child are physically removed during this process.

The algorithm for this operation is given as Algorithm. 5.1 and performed as shown in Fig. 5.1. The node to be deleted and its parent are locked. The child pointer of parent node p is updated to point to the child $x.right$ or $x.left$ of given node x to be removed. Also, the left and right both pointers of the node x are updated to point to its parent p. This is done to allow traversal towards right track without using any synchronization technique. If there were any thread stuck in the node x, they will go back to parent p and continue traversing.

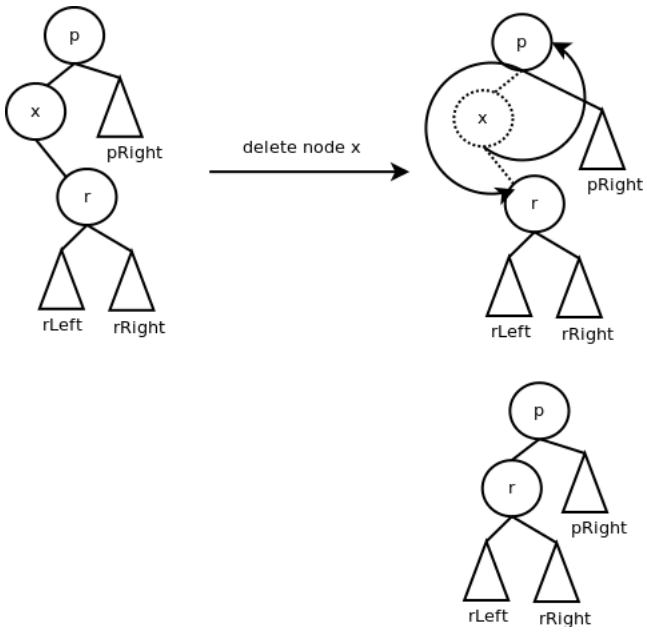

Figure 5.1: The physical removal of node operation in contentional friendly manner

2. **zigRightRotation**

This is a part of lazy splaying operation. Depending on certain pre-condition at each node based on access count of node and its left and right subtrees, different rotation operations are performed. Traditional splay technique involves Zig, Zig-Zig and Zig-Zag rotation operations. Those rotation operations are performed as shown in Fig. 4.4. However, in our lazy splaying technique we will perform only Zig and Zig-Zag rotations.

As rotation involves locking of few local nodes. Consider there are concurrent traversal that is preempted on node x during rotation. The zigRightRotation shown in Fig. 5.2 causes node l (left child of x) to be shifted up and node x to be shifted down and becomes child of l. The traversal preempted on node x will now avoid node l which was on traversal path before rotation thus violates correctness if these nodes are in the correct path.

To solve this problem, the rotation operation is modified that produces the same output as corresponding traditional rotation operation would produce. The algorithm for zigRightRotation operation is described as Algorithm. 5.2. Three nodes l, x, and p are locked. A new node x' is introduced to take x's place in the tree. Node x is now flagged as removed. This way, the rotation operation is preserved and the preempted traversal continues towards right and correct path.

The zigLeftRotation is mirror operation for this. But, the remove flag of node x in left rotation is flagged as *left-rotate*, to direct the thread towards correct path during traversal which will be explained in detail in later operations.

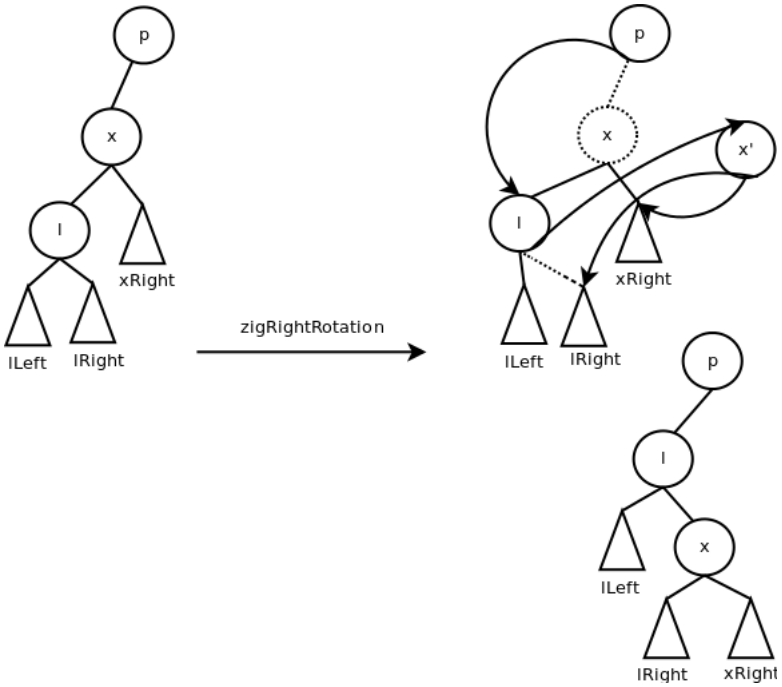

Figure 5.2: The zig rotation of node in contentional friendly manner

3. **zigLeftZagRightRotation**

This is the second rotation operation based on the other pre-condition at certain node. Technically its a two subsequent rotations (either left and right or right and left), but this is done as single rotation that includes two steps.

This operation is performed as shown in Fig. 5.3 and causes multiple nodes to move their positions in the tree. The grand child node r will move to top and both parent p and x nodes moves down. The algorithm is desribed in Algorithm. 5.3 As in zigRightRotation, we will introduce two new nodes p' and x' that will take place of original parent's and x's position after rotation. parent$p$ and x will be marked as removed with *true* and *left-rotate* flag respectively. This way, the concurrent traversal preempted at x and p will follow right traversal path after rotation as well. Thus, any concurrent traversal preempted on x or parent will still be able to reach any node that was reachable before the rotation.

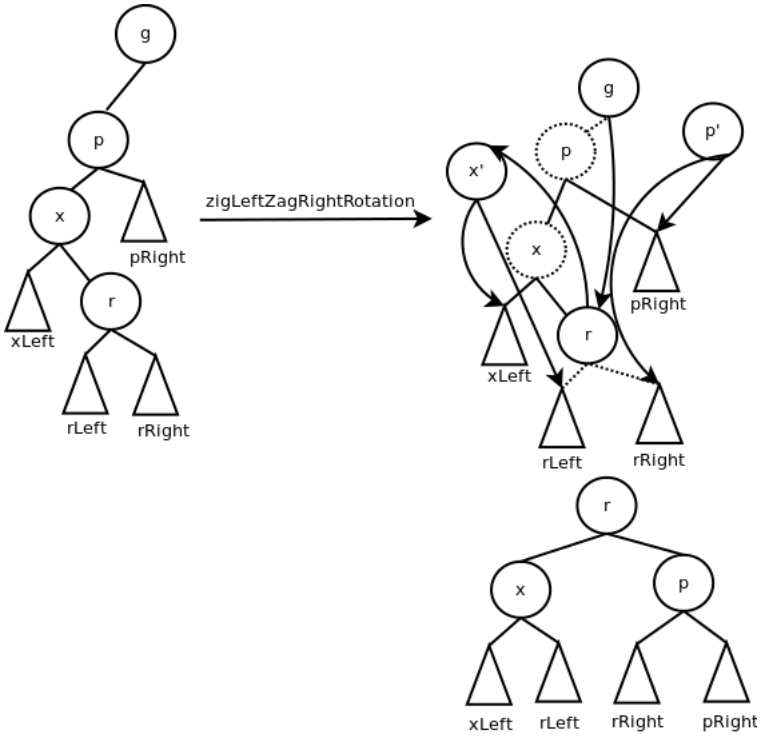

Figure 5.3: The zig-zag rotation of node in contentional friendly manner

Algorithm 5.1 Physical node removal operation

```
function removeNode(Node parent, Node x)
    if parent.remove == true then                ▷ Check if parent is physically removed.
        return false
    end if
    if x = ⊥ then                                ▷ Check if node x exists.
        return false
    end if
    lock(parent)
    lock(x)
    if x.delete == false then                    ▷ If node x is not lazily deleted do not remove it.
        unlock(parent)
        unlock(x)
        return false
    end if
    if x.left ≠ ⊥ ∧ x.right ≠ ⊥ then             ▷ If node x has two children do not remove it.
        unlock(parent)
        unlock(x)
        return false
    else if x.left ≠ ⊥ then                      ▷ If node x has only left child.
        child ← x.left
    else                                         ▷ If node x has only right child.
        child ← x.right
    end if
    if x == parent.left then                     ▷ If node x is left child of a parent.
        parent.left ← child
    else                                         ▷ If node x is right child of a parent.
        parent.right ← child
    end if
    x.left ← parent
    x.right ← parent
    x.remove ← true
    unlock(parent)
    unlock(x)
    return true
end function
```

Algorithm 5.2 Zig right rotataion operation

```
function zigRightRotation(Node parent, Node x, Node l)
    if parent.remove == true then                    ▷ Check if parent is physically removed.
        return false
    end if
    if x = ⊥ then
        return false
    end if
    if l = ⊥ then
        return false
    end if
    lock(parent)
    lock(x)
    lock(l)
    lRight ← l.right
    xRight ← x.right
▷ create new node x′ to replicate x
    x′.key ← x.key
    x′.left ← lRight
    x′.right ← xRight
    l.right ← x′
    if x == parent.left then                         ▷ If node x is left child of a parent.
        parent.left ← l
    else                                             ▷ If node x is right child of a parent.
        parent.right ← l
    end if
    x.remove ← true
    unlock(l)
    unlock(x)
    unlock(parent)
    return true
end function
```

Algorithm 5.3 Zig left Zag right rotataion operation

```
function zigLeftZagRightRotation(Node grand, Node parent, Node x, Node r)
    if grand.remove == true then                    ▷ Check if grand is physically removed.
        return false
    end if
    if parent = ⊥ then
        return false
    end if
    if x = ⊥ then
        return false
    end if
    if r = ⊥ then
        return false
    end if
    lock(grand)
    lock(parent)
    lock(x)
    lock(r)
    xLeft ← x.left
    rLeft ← r.left
    rRight ← r.right
    parentRight ← parent.right
▷ create new node x′ to replicate x
    x′.key ← x.key
    x′.left ← xLeft
    x′.right ← rLeft
    r.left ← x′
▷ create new node parent′ to replicate parent
    parent′.key ← parent.key
    parent′.left ← rRight
    parent′.right ← parentRight
    r.right ← parent′
    if parent == grand.left then                    ▷ If node parent is left child of a grand.
        grand.left ← r
    else                                            ▷ If node parent is right child of a grand.
        grand.right ← r
    end if
    x.remove ← true
    parent.remove ← true
    unlock(r)
    unlock(x)
    unlock(parent)
    unlock(grand)
    return true
end function
```

5.2.2 Background Re-structuring Operations

In this section, we will discuss about the re-structuring process that will be executed independently by the background thread. These operations are responsible for *lazy structural adaptation with long/semi splaying* process. There are four operations involved in the background re-structuring process and they are *longSplayDFS*, *propagateCounter*, *backgroundLongSplay* and *splayNode*.

1. **longSplayDFS**

   This is a recursive depth first search algorithm that traverses entire tree starting from root of the tree. It checks if any node is lazily deleted and is safe to remove then calls the removeNode procedure. It then executes the *propagateCounter* procedure to update the access count counters at each node. Finally It checks if lazy splaying is to be done at the particular node and calls the splayNode procedure to perform splaying. The algorithm is shown as Algorithm. 5.4.

2. **propagateCounter**

   As the access counter of each node are updated on each access operation. It needs to propagate towards the root once any change is made. This procedure updates the *leftCnt*, *rightCnt* based on its up-to-date children's counters. However, this is implemented asynchronously the values may not be correct. However, this will not make any difference in the correctness of the algorithm. The algorithm is as shown in Algorithm. 5.5.

3. **backgroundLongSplay**

   This procedure runs in background in an infinite loop and repeatedly calls the *longSplayDFS* procedure on the root node. That way it keeps on performing the splaying task on each node and balances the tree height and keeps the frequently access items towards the root of the tree. The algorithm is as shown in Algorithm. 5.6.

4. **splayNode**

   This is the procedure to perform lazy splaying on the node that is accessed either by find operation or insert operation. Depending on the access counter values of neighboring nodes it performs either zig or zig-zag operation. zigRightrotation and zigLeftrotation are mirror to each other and we call zig operation in general. zigLeftZagRightRotation and zigRightZagLeftRotation are mirror to each other and we call zig-zag operation in general. Zig-zag is carried out if the total number of accesses to the node right subtree is larger than the total number of accesses to the node-parent and its right subtree. If zig-zag was not performed then zig is performed if the total number of accesses to the node and its left subtree is larger than

the total number of accesses to the node-parent and its right subtree. The algorithm works as shown in Algorithm. 5.7.

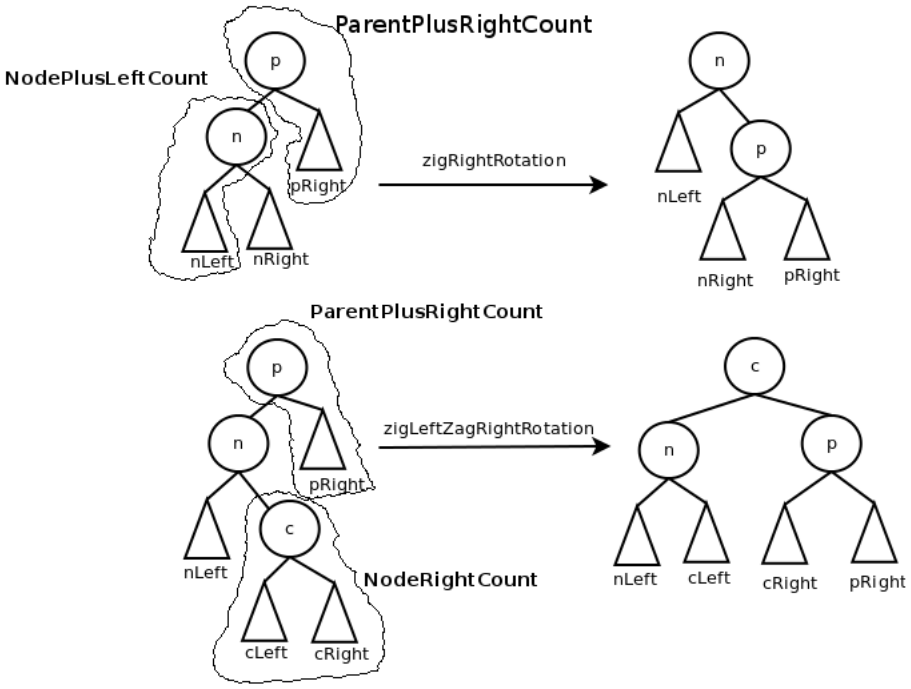

Figure 5.4: Lazy splaying conditions

5.2.3 Basic Abstract Operations

In this section, we will discuss the primitive operations of BST that are responsible to preserve the binary search tree property. Each of these operations requires traversal of the tree starting from the root node. The three basic abstract operations are *find*, *insert* and *delete*. The helper procedures that those basic abstract operations uses are *getNext* and *isValid* procedures. These procedures are also same as Raynal's idea [3], but we incorporated our lazy splaying technique in some of the operations.

Algorithm 5.4 Long splay using depth first search

```
function longSplayDFS(Node x)
    if x = ⊥ then
        return
    end if
    longSplayDFS(x.left)
    longSplayDFS(x.right)
    if x.left ≠ ⊥ ∧ x.left.delete == true then
        removeNode(x, x.left)
    end if
    if x.right ≠ ⊥ ∧ x.right.delete == true then
        removeNode(x, x.right)
    end if
    propagateCounter(x)
    if x.left ≠ ⊥ ∨ x.right ≠ ⊥ then          ▷ If left or right child is present then check for splay
        splayNode(parent, x, x.left, x.right)
    else
        return
    end if
end function
```

Algorithm 5.5 Propagate access count towards root

```
function propagateCounter(Node x)
    if x.left ≠ ⊥ then
        x.leftCnt ← x.left.leftCnt + x.left.rightCnt + x.left.selfCnt
    else
        x.leftCnt ← 0
    end if
    if x.right ≠ ⊥ then
        x.rightCnt ← x.right.leftCnt + x.right.rightCnt + x.right.selfCnt
    else
        x.rightCnt ← 0
    end if
end function
```

Algorithm 5.6 Back ground long splaying

```
function backGroundLongSplay
    while true do
        longSplayDFS(root)
    end while
end function
```

Algorithm 5.7 Lazy splay operation

```
function SPLAYNODE(Node parent, Node lChild, Node rChild)
▷ check for zigRight and zigLeftZagRight only
    nodePlusLeftCount ← lChild.selfCnt + lChild.leftCnt
    parentPlusRightCount ← parent.selfCnt + parent.rightCnt
    nodeRightCount ← lChild.rightCnt
    if nodeRightCount ≥ parentPlusRightCount then                ▷ zigzag condition
        grand ← parent.parent
        zigLeftZagRightRotation(grand, parent, lChild, lChild.right)
        parent.leftCnt ← lChild.right.rightCnt
        lChild.rightCnt ← lChild.right.leftCnt
        lChild.right.rightCnt ← lChild.right.rightCnt + parentPlusRightCount
        lChild.right.leftCnt ← lChild.right.leftCnt + nodePlusLeftCount
    else if nodePlustLeftCount > parentPlusRightCount  then      ▷ zig condition
        grand ← parent.parent
        zigRightRotation(grand, parent, lChild)
        parent.leftCnt ← lChild.rightCnt
        lChild.rightCnt ← lChild.rightCnt + parentPlusRightCount
    end if
▷ zigLeft and zigRightZagLeft are symmetric to above two cases
end function
```

1. **getNext**

   This procedure is as described in Algorithm. 5.11. Given a node x as the input to this procedure, it will try to find the successor node in the traversal path. It checks if the node is removed and if the flag is *left-rotate* then the node was concurrently removed by the *zigLeft-Rotation* procedure. As, we know the left rotation causes the node (that is removed in this procedure) to be moved donwards in a traditional rotation, so the *getNext* procedure needs to direct the traversal towards the right child as it contains at least as many nodes in its path that were in the path of the node before the rotation. If the flag is *true* it was removed either by *zigRightRotation* or *removeNode* operation, so the traversal continues towards left child. If the flag is *false* the node's key value is checked and if the key matches with they *key* then the traversal is finished. Otherwise, it continues as per general BST traversal rule, right when key is greater than *node.key* or left when the key is less than *node.key*. The Fig. 5.3, Fig. 5.2, Fig. 5.1 shows the traversal path situation for different cases.

2. **isValid**

   This procedure is responsible to check the status of the node during concurrent access of the node. It checks if the node is valid or safe to perform some operations on it. It works as in Algorithm. 5.12. It checks if the node has been physically removed or not. It then checks, if the key of the node matches to the input *key* and if they matches the traversal is finished. If

they do not matches, it checks if the node has children or not on the place where key would exist. This will be the place where new node is inserted.

3. **find**

   This operation calls the *getNext* procedure in a loop until the node with *key* is found or NULL is encountered. If it finds the desired node, it increments its *selfCnt* (access counter) and tries lazySplaying on that node. The algorithm works as shown in Algorithm. 5.8.

4. **insert**

   This operation works like a sequential insert operation. It starts from root and tries to find the location of node to be inserted. However, this operation does modify some node, it needs to lock that node. During locking process the concurrent operation might have modified the node (like may have removed it). It needs to check if the node is valid, if the node is not valid, it continues traversing to find the another place of insertion. As, the nodes are lazily deleted, it checks if simply unflagging the delete status performs the insert operation. This operation works as shown in Algorithm. 5.9. After insertion, it increments its selfCnt (access counter) and tries lazySplaying on that node.

5. **delete**

   This works similar to delete operation. It starts looking the node from the root. It will lock the node before deletion. It uses *isValid* procedure to ensure it is deleting the targeted node. Otherwise, it continues traversing to find the node to be deleted. This operation works as shown in Algorithm. 5.10.

Algorithm 5.8 Find operation

```
function FIND(key)
    x ← root
    result ← false
    while ((next ← getNext(x, key)) ≠ ⊥) do
        x ← next
    end while
    if x.key == key then
        if x.delete == false then
            result ← true
            SplayNode(x, x.left, x.right)
            x.selfCnt + +
        end if
        return result
    end if
end function
```

Algorithm 5.9 Insert operation

```
function INSERT(key)
    x ← root
    result ← false
    Lbl:
    while ((next ← getNext(x, key)) ≠ ⊥) do
        x ← next
    end while
    if next == ⊥ then
        lock(x)
        if isValid(x, key) == false then
            unlock(x)
            x ← next
            go to Lbl:                          ▷ node is invalid continue searching
        else
            if x.key == key then
                if x.delete == true then        ▷ node x is lazily deleted, undelete it
                    x.delete ← false
                    result ← true
                end if
            else                                ▷ create new node new to be inserted
                new.key ← key
                if x.key > key then
                    x.right ← new
                else
                    x.left ← new
                end if
                result ← true
            end if
        end if
        SplayNode(x, x.left, x.right)
        x.selfCnt++
        unlock(x)
        return result
    end if
end function
```

Algorithm 5.10 Delete operation

```
function DELETE(key)
    x ← root
    result ← false
    Lbl:
    while ((next ← getNext(x, key)) ≠ ⊥) do
        x ← next
    end while
    if next == ⊥ then
        lock(x)
        if isValid(x, key) == false then
            unlock(x)
            x ← next
            go to Lbl:                          ▷ node is invalid continue searching
        else
            if x.key == key then
                if x.delete == false then       ▷ delete the node lazily
                    x.delete ← true
                    result ← true
                end if
            end if
        end if
    end if
    unlock(x)
    return result
end function
```

Algorithm 5.11 Get next node operation

```
function GETNEXT(Node x, key)
    remove ← x.remove
    if remove = "by − left − rot" then
        next ← x.right
    else if remove == true then
        next ← x.left
    else if x.key > key then
        next ← x.right
    else if x.key == key then
        next ← ⊥
    else
        next ← x.left
    end if
    return next
end function
```

Algorithm 5.12 Is node valid operation

```
function isValid(Node x, key)
    if x.remove == true then
        return false
    else if x.key == key then
        return true
    else if x.key > key then
        next ← x.right
    else
        next ← x.left
    end if
    if next == ⊥ then return true
    end if
     return false
end function
```

Chapter 6

Conclusion and Future Work

In this thesis report we have borrowed the different ideas of implementing concurrent binary search trees and merged those ideas, modified it and presented the new approach of implementing self adjusting binary search tree. We proposed the implementation that is scalable and contention friendly using the re-balancing technique called lazy splaying that enhances the performance as well as scalability of self adjusting binary search trees. Unlike other blocking implementation of self-adjusting binary search tree, our implementation is partial blocking that makes the algorithm efficient and unlike other non-blocking implementation of self-adjusting binary search tree, our implementation is simple as the algorithm doesn't involve complexity of non-blocking implementations.

We conclude that, various techniques (contention friendly methodology, lazy splaying, semi/long splaying, segregation of operations into abstract modification and structural adaptation) that we have used in our implementation makes the algorithm scalable, efficient and easier to design on multi-core environment. Contention-friendly methodology helps in enhancing the performance of lock-based data structures. Lazy splaying tries to localize the rotations that reduces the overhead of high contentions and avoids the sequential bottlenecks. This process however in concurreny will still produce the self adjusting tree as global effect. Semi/ Long splaying helps in controlling the unwanted monotonic growth of the tree and maintains the access counters up to date. Segregation of operations into abstract modification and structural adaptation allows the binary search tree to scale with a reasonably large number of threads.

Plugging in the lazy splaying and contention friendly methodology to the existing lock-based implementation of binary search tree to improve the performance of BST could be one of the future work.

Implementing this algorithm in Java and conducting experimental evaluation to compare the performance of this algorithm against the existing algorithm will be an another interesting future work.

Using the similar technique to implement another concurrent data structure could also be the next future work.

Bibliography

[1] J. Bell and G. Gupta, "An evaluation of self-adjusting binary search tree techniques," *Software: Practice and Experience*, vol. 23, no. 4, pp. 369–382, 1993.

[2] L. Bougé, J. Gabarro, X. Messeguer, N. Schabanel, *et al.*, "Height-relaxed avl rebalancing: A unified, fine-grained approach to concurrent dictionaries.," 1998.

[3] T. Crain, V. Gramoli, and M. Raynal, "A contention-friendly binary search tree," in *Euro-Par 2013 Parallel Processing*, pp. 229–240, Springer, 2013.

[4] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan, "Cbtree: A practical concurrent self-adjusting search tree," in *Distributed Computing*, pp. 1–15, Springer, 2012.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1, pp. 126–134, IEEE, 1999.

[6] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM Computer Communication Review*, vol. 29, pp. 251–262, ACM, 1999.

[7] Y. A. B. K. A. Morrison, "Concurrent search tree by lazy splaying,"

[8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[9] wiki, "Concurrent computing." `http://en.wikipedia.org/wiki/Concurrent_computing`.

[10] M. Moir and N. Shavit, "Concurrent data structures," *Handbook of Data Structures and Applications*, pp. 47–14, 2007.

[11] wiki, "Multi threading." `http://en.wikipedia.org/wiki/Multithreading_\%28computer_architecture\%29`.

[12] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.

[13] L. Lamport, "A new solution of dijkstra's concurrent programming problem," *Communications of the ACM*, vol. 17, no. 8, pp. 453–455, 1974.

[14] M. P. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pp. 522–529, IEEE, 2003.

[15] A. Agarwal and M. Cherian, *Adaptive backoff synchronization techniques*, vol. 17. ACM, 1989.

[16] M. Herlihy, "A methodology for implementing highly concurrent data objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 5, pp. 745–770, 1993.

[17] wiki, "Compare and swap." `http://en.wikipedia.org/wiki/Compare-and-swap`.

[18] R. K. Treiber, *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[19] S. Prakash, Y. H. Lee, and T. Johnson, "A nonblocking algorithm for shared queues using compare-and-swap," *Computers, IEEE Transactions on*, vol. 43, no. 5, pp. 548–559, 1994.

[20] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 51, no. 1, pp. 1–26, 1998.

[21] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[22] M. AdelsonVelskii and E. M. Landis, "An algorithm for the organization of information," tech. rep., DTIC Document, 1963.

[23] R. Bayer, "Symmetric binary b-trees: Data structure and maintenance algorithms," *Acta informatica*, vol. 1, no. 4, pp. 290–306, 1972.

[24] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," in *ACM Sigplan Notices*, vol. 45, pp. 257–268, ACM, 2010.

[25] A. Natarajan, L. H. Savoie, and N. Mittal, "Concurrent wait-free red black trees," in *Stabilization, Safety, and Security of Distributed Systems*, pp. 45–60, Springer, 2013.

[26] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pp. 131–140, ACM, 2010.

[27] G. Barnes, "Wait-free algorithms for heaps," *Computer Science and Engineering, University of Washington, Tech. Rep TR-94-12-07*, 1994.

[28] K.-L. Lin, C.-K. Lo, and R.-S. Tsay, "Source-level timing annotation for fast and accurate tlm computation model generation," in *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pp. 235–240, IEEE, 2010.

[29] T. Crain, V. Gramoli, and M. Raynal, "No hot spot non-blocking skip list," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pp. 196–205, IEEE, 2013.

[30] T. Crain, V. Gramoli, and M. Raynal, "A speculation-friendly binary search tree," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 161–170, ACM, 2012.

[31] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 652–686, 1985.

Vita

Graduate College

University of Nevada, Las Vegas

Mahesh Raj Regmee

Degrees:

Bachelor of Computer Engineering 2008

Tribhuwan University,IOE, Pulchowk Campus, Nepal

Thesis Title: Self Adjusting Contention Friendly Concurrent Binary Search Tree by Lazy Splaying

Thesis Examination Committee:

Chairperson, Dr. Ajoy K. Datta, Ph.D.

Committee Member, Dr. Lawrence Larmore, Ph.D.

Committee Member, Dr. Yoohwan Kim, Ph.D.

Graduate Faculty Representative, Dr. Emma E. Regentova, Ph.D.