Concurrent Binary Search Trees via Logical Ordering: A Report

Roman Larionov
University of Central Florida
4000 Central Florida Blvd
Orlando, Florida
rlarionov@cs.ucf.edu

Harsh Patel
University of Central Florida
4000 Central Florida Blvd
Orlando, Florida
harshpatel@knights.ucf.edu

Khoa Hoang
University of Central Florida
4000 Central Florida Blvd
Orlando, Florida
maximus64@knights.ucf.edu

ABSTRACT

The paper provides a brief summary and an evaluation of *Practical Concurrent Binary Search Trees via Logical Ordering*[2]. The significance of the paper's use of logical ordering of BSTs to provide lock-free scalable search operation is evaluated versus other open-source binary search tree implementations.

An explanation for each of the four primary algorithms: contains, insert, remove, and rebalance is provided. Moreover, correctness properties like linearizability, deadlock freedom, and progress guarantees are discussed for each of the algorithms. Performance evaluation of a concurrent C++ implementation of the paper's algorithms relative to skiplist and NBBST implementations are also provided.

It is observed that the improvement in the relative performance of the implementation is proportional to the frequency of contain calls. It is concluded that the paper successfully uses logical ordering to provide a lock-free scalable search algorithm.

1. INTRODUCTION

The paper focuses on concurrent algorithms for binary search trees. The key challenge in designing efficient concurrent BST algorithms is to provide a lock-free lookup operation. Few existing algorithms combat this challenge by maintaining all of the keys in the leaves of the tree. Others provide no support for remove operations. Despite the difference in approaches, they all base their synchronization on the tree layout.

The paper presents a different approach to achieve lock-free lookups using the logical ordering of the tree. It exploits the fact that a binary search has a total ordering between all the elements. This logical ordering can be maintained explicitly in the data structure by storing the predecessor and successor of each node. The logical ordering is not affected by layout manipulations so lookup operations can proceed concurrently.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

2. ALGORITHM

2.1 Search

Starting at the root of the tree, the search algorithm (shown below) iteratively traverses down the tree by comparing the key to each node. It returns either the node with the search key (if found) or the leaf node (the predecessor or the successor) if it reaches the end of the tree. The search algorithm is a lock-free scalable algorithm that is called by insert, remove and contains operations.

2.2 Contains

The contains algorithm (shown below) begins by calling the search method. The search algorithm may return the search node, its predecessor, or its successor. If search does not find the key node, lines 2 and 3 in the contains algorithm exploit logical ordering to determine if the value exists in the tree. If a valid matching key is found in the tree, the contains method returns true. Otherwise, the value did not exist in the tree when the contains method was called and false is returned.

2.3 Insert

The insert algorithm calls the search method and checks if the returned node's key is greater than the key to be inserted. If so, it sets predecessor to the returned node's predecessor; otherwise, it sets the predecessor to the returned node. After logically locking the predecessor, it checks if the key to be inserted is in the [predecessor, successor] interval and that the interval is valid. If yes, it checks if either the predecessor's or the successor's key equals the key to be inserted. If so, it returns false because this violates one of the constraints of the BST: BST cannot contain duplicate values. Otherwise, it allocates the memory for the new node (to be inserted) and acquires the physical lock. After performing a logical insertion, the logical lock is unlocked. Then, a helper function insertToTree() is called to perform the physical insertion and the tree rebalancing.

2.4 Remove

Remove is algorithmically similar to insert. It also begins by calling the search method. It performs the same check (returned key vs key to be removed) as insertion. Similarly, it checks if the key to be removed is in the [predecessor, successor] interval and that the interval is valid. If the successor key is greater than the key to be removed, the key does not exist in the tree. This leads to an unsuccessful remove for which we unlock the predecessor node and return false.

Otherwise, we obtain the logical and physical locks respectively and perform logical and physical deletions in order. A helper function removeFromTree() performs the physical deletion of the node and calls rebalance() if necessary.

2.5 Rebalance

The rebalance operation is designed to perform sweeping rotations on an entire subtree, specified by a provided root subtree node and one of its children. The algorithm begins by determining whether the child is on the left or right of the root node. It then updates the heights of the two nodes to determine the sub-tree's balancing factor. If the balancing factor is determined to be within an acceptable range [-1, 1], the operation is halted. If a rotation is required, it continues by determining whether the left or right subtree needs to be altered. If it is determined that the left subtree needs to be altered but child == node->right (or vice versa), the algorithm attempts to unlock to current child and lock the alternate child. Once the correct child is locked, the appropriate grand child is then locked. All three nodes are required to perform rotations to avoid possible data race conditions when manipulating left/right/parent pointers. Once the rotation has been performed, the loop begins again with the child as the root.

2.6 Progress Guarantee

There are two types of locks: SuccLock (logical ordering lock) and TreeLock (physical layout lock). The algorithm maintains a locking order that threads follow to avoid deadlocks. Between a succLock and a treeLock, a succLock is always acquired first. Between two succLocks, the lock of the node with the smaller key is acquired first. Between two treeLocks, the lock of the node that appears lower in the tree is acquired first.

2.7 Linearization Points

A successful contains linearizes when the node's valid field is determined to be false. An unsuccessful contains linearizes when the nodes is marked as removed or when k is not found in the tree.

A successful insert linearizes when the predecessor's successor field is updated to point to the new node. At this point, any future operation will be able to observe the new node. An unsuccessful insert linearizes when it returns false.

A successful removal linearizes as soon as the node is logically deleted. Future insert and remove calls will observe that the node has a different successor. Future contains will observe the logical marking and return false. An unsuccessful removal linearizes when it returns false.

2.8 Synchronization

The tree synchronization is based on two locks: succLock and treeLock. Each update operation is applied by acquiring the succLock and treeLock respectively. The logical ordering is then updated before releasing the succLock. Then the physical tree layout is updated before releasing the treeLock.

3. BACKGROUND

There were several state-of-the-art BST and AVL trees described and compared against within the paper. Many of these are optimized for various other aspects of concurrent binary trees. Some of these trees are able to perform rebalancing, while others retain a worst-case O(n) traversal. Some trees exhibit non-blocking properties for insert/remove, while others focused their efforts on optimizing logical deletion ordering with helper threads. This shows that the authors of our paper obviously wanted to select from a wide range of alternative data structures to perform comparisons with. There are many features that a concurrent tree may possess. Every example from popular literature picks and chooses features that it deems important

3.1 Internal vs. External Trees

Internal trees hold the property that data is stored within all nodes in the graph, whereas an external tree only stores data in their leaf nodes. Crain el al have an blocking, external "contention-friendly" tree[1] which has an interesting take on node removal. Instead of incrementally deleting logically removed nodes or immediately deleting nodes, they have a separate "helper" thread which periodically scans the tree and performs rotations and physical removals.

3.2 Non-Blocking

Non-blocking data structures have the guarantee that regardless of thread suspension, some operation will make system-wide progress. There are several non-blocking variations that have been made available; the most prominent of which is Ellen et al's non-blocking BST[3]. This variation has an external format but does not support rebalancing in any way.

3.3 Key Differences

The major distinguishing aspects of our paper includes the ability to maintain a fast, lock-free contains operation while physically deleting nodes in place. The logical ordering layout is a sweet spot between code complexity and optimal performance. Unlike external tree formats, which can have long traversal heights, this tree maintains a classic internal format but takes an optimistic traversal approach by skipping around the tree using pred and succ pointers. Since these pointers are incrementally updated, there are no disproportionate calls to insert/remove (with exception to rebalance operations). Logical deletion approaches, such as Crain et al's contention-friendly tree[1], suffer from elongated periods between physical removals which can fill the tree up with wasted space and decrease traversal times (especially when rotations take place).

4. TESTING AND EVALUATION

For testing and evaluation, we wrote a benchmark program in C++ to compare our implementation with other open source implementations of the binary search tree. Our main source for test data structures came from Wicht from his personal GitHub profile[4]. The repository has five distinct implementations of concurrent binary trees, though we only made use of his SkipList and Non-Blocking BST implementations. Our benchmarking program first generates random data to populate the tree, then spawn multiple threads (1 to 32), each thread will randomly choose insert, remove, or contains operation to run. The distribution of each operation is controllable by the user. After it runs through all the iterations, the program will print to the console the time it takes to complete each run with different threads count. The platform we chose to run our empirical evaluations was an Intel x64 architecture, with a Core i7 4770K 3.9GHz running Arch Linux.

4.1 Procedures

We performed several different tests; comparing our implementation both with other concurrent data structures as well as a sequential AVL tree. Under both testing circumstances, we executed a series of randomly chosen operations (insert/remove/contains) and output the mean execution time from a sample of 1000 test runs. The random numbers used were pre-generated to avoid corrupting any profiling results.

To display the strengths and weaknesses of each data structure type, we visually displayed the results in graphical form. Figures 1 through 3 show the results between our tree and other concurrent structures. Our tree performed best with a high percentage of contains operations. This is an expected result, as our tree was designed to be extremely performant for a high ratio of lookup operations. The non-blocking BST performed best in all other test cases, mainly due to its lock-free insert/remove.

Figures 4 through 6 show the results between a sequential AVL tree with our concurrent version. As expected, a large amount of contains operations resulted in a significant performance disparity. An unexpected result came from the test with equal proportions of insert/remove/contains operations. The concurrent structure gradually becomes less performant in comparison with the sequential implementation. We take this to mean that the blocking nature of the insert/remove operations results in a noticeable drop in throughput, as more threads are liable to wait for locks to become available.

Figure 1:

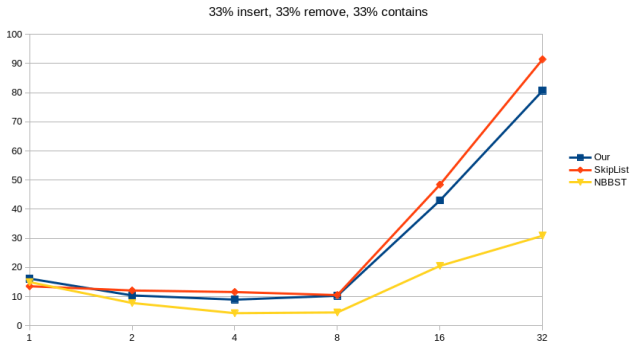


Figure 2:

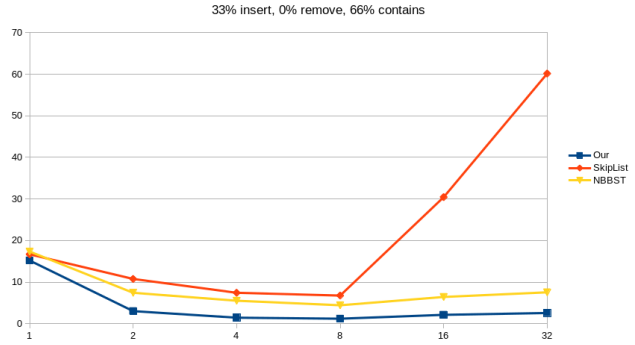
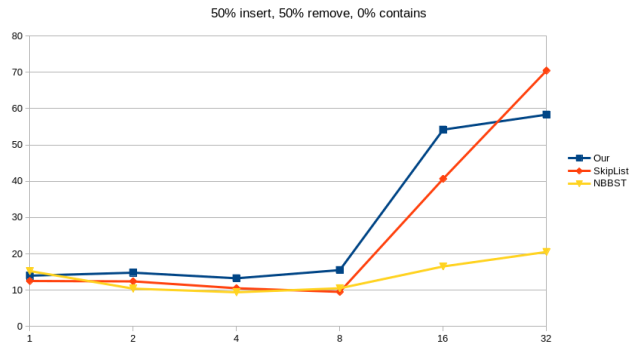


Figure 3:



5. OUR IMPLEMENTATION

We prioritized implementing an overall structure that was effectively the same as the pseudo code provided, but needed to change various pieces to suit our needs. While the paper was written with a garbage collected language in mind, we wanted to write our implementation in C++. This presented various difficulties which we had to work around.

5.1 Obstacles

5.1.1 Linearizability

One of the main obstacles that we faced had to do with the liberties that the pseudo code took in compacting several operations into singular irreducible operations. We needed to design our algorithm in such a way as to "restart" if certain conditions haven't been met. For instance, within our insert/remove operations, we need to lock the logical layout of the tree in order to guarantee linearizability. More specifically, we need to lock both the pred and succ values surrounding our queried node, as those are both liable to create data races when inserting/removing a node. Since these locking operations require two operations to complete, we needed to implement a failure state where we can unlock pred if we observe succ to be held or not within the range of the key we're trying to insert.

5.1.2 C++ semantics

The remainder of the challenges pertain mainly with the semantics of the C++ language. Since our algorithm is inherently recursive in nature, we required the use of ReentrantLocks to avoid self-blocking within a thread. C++ provides a similar mechanism to achieve this in the form of std::recursive_lock. While this solves one problem, it

Figure 4: Comparable performance when operations are evenly distributed.

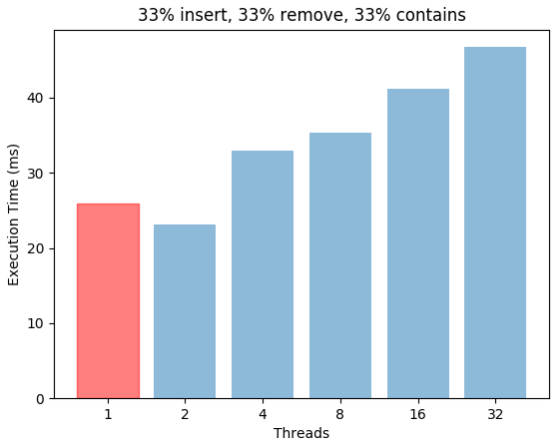


Figure 5: Expected disparity between sequential and concurrent AVL on heavy contains workload.

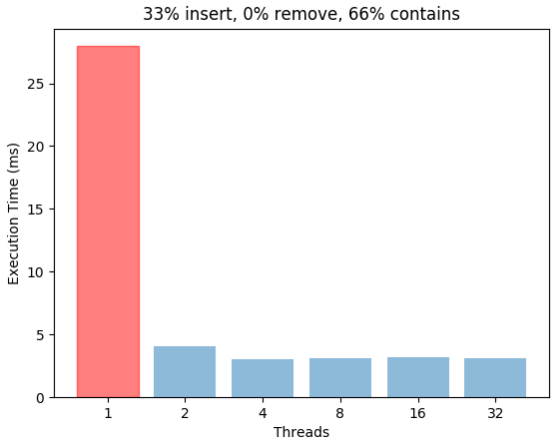
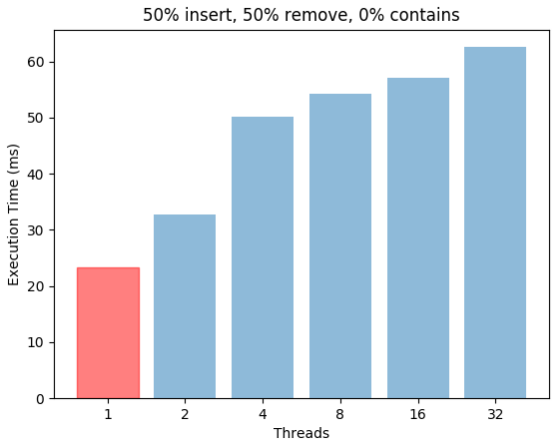


Figure 6: Sequential algorithm outperforms Logical Ordering when heavy insert/remove workload.



disregards another. Mainly, our algorithm demanded the ability for a thread to test ownership of tree locks within rebalance. C++ has a built in mechanism in the form of std::unique_lock, which is an RAII wrapper for a mutex that provides ownership testing. The issue is that RAII wrappers call their implicit destructor, which automatically unlocks the contained mutex, upon exiting its local scope. To make it work with recursion, we were forced to construct our own lock primitive, based on std::recursive_lock, that performs manuel ownership testing. This ended up working really well, although it split our structure into separate files.

5.1.3 *Implementation*

Our experience researching and implementing this paper was at times straightforward and frustrating. Our initial implementation was not bug free and we were forced to become acquainted with various concurrent debugging tools. We initially used a mixture of gdb and lldb for debugging on linux-style systems, but quickly found these to be too lacking for our needs. We opted to use Visual Studio's, which has nice concurrent debugging tools. Splitting our debugging and building environment was ultimately a good decision.

5.1.4 *Team Coordination*

Another challenge we faced was coordination. Contributing to a shared, single source file is prone to merge conflicts. Early on we realized that a paired programming style would be the best method of tackling the implementation. We would periodically gather to incrementally build up our concurrent solution. One person would program, while the others would read and investigate potential problems. This was essential for the completion of the project. Though it became relatively easy to cooperate towards the end of the project, we initially struggled to become acquainted with one another.

6. CONCLUSIONS

Concurrent BST/AVL algorithms provided in *Practical Concurrent Binary Search Trees via Logical Ordering*[2] that use logical ordering were discussed and explained. Primarily, it was shown how the authors successfully provided a simple lock-free lookup operation. The correctness properties of these algorithms were examined and analyzed.

Some of the challenges in implementing the algorithms provided in the paper in C++ were discussed. Graphs for evaluation metrics of the implementation compared to two other open source implementation were shown and analyzed. It is concluded that the authors successfully designed a BST, that provides a lock-free lookup, competitive with other state-of-the-art balanced BST implementations.

7. REFERENCES

[1] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly binary search tree. In *European Conference on Parallel Processing*, pages 229–240. Springer, 2013.

[2] D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. *ACM SIGPLAN Notices*, 49(8):343–356, 2014.

[3] 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, pages 131–140. ACM, 2010.

[4] B. Wicht. C++ implementation of concurrent binary search trees. https://github.com/wichtounet/btrees, 2015.