Article title: An Improved Lock-Free Binary Search Tree

Authors: Nicholas Whitlock[1], Jose Luis[2], Sam Shannon[3], Mark Alano[4], COP 4520[5]

Affiliations: University of Central Florida[1]

Orcid ids: 0000-0002-4592-0850[5]

Contact e-mail: cop4520@outlook.com

License information: This work has been published open access under Creative Commons Attribution License <http://creativecommons.org/licenses/by/4.0/>, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. Conditions, terms of use and publishing policy can be found at <https://www.scienceopen.com/>.

Preprint statement: This article is a preprint and has not been peer-reviewed, under consideration and submitted to ScienceOpen Preprints for open peer review.

DOI: 10.14293/S2199-1006.1.SOR-.PP8OYVU.v1

Preprint first posted online: 30 May 2019

Keywords: binary search tree, Lock-Free Binary Search Trees

Add() or Contains() operations, the algorithm checks for this flag and completes the physical deletion of the node. This algorithm is proved to be linearizable, as per Chatterjee, Nguyen, and Tsigas [1]. We explored the linearizability of this algorithm in our implementation and found that, through the use of CAS operations, the proposed operations specified in the algorithm have valid linearization points. The main contribution of the algorithm is that any modify operation, such as Add() or Remove(), assist does not restart the concurrent modify operation from the root but rather from the vicinity of failure. In this report, we explored this algorithm and implemented our interpretation of the proposed Efficient Lock-free Binary Search Tree and provide further technical details and experimental results to back our findings [1].

II. RELATED WORK

Similar works include lock-free implementations of data structures such as wait-free singly linked-lists [2], [3], lock-free doubly linked-lists [4], lock-free hash tables [5], and lock-free skip lists [6]. These implementations provide an abundant collection of data structures that one can use when writing software intended for a system with multiple cores and can utilize full system performance. Although an abundant collection, in literature, there is not much work on performance-efficient concurrent BST implementations that are lock-free. There are improvements to this publication that we explored in an effort to properly benchmark its performance and reported errata on the author's online repository [7] were taken into account when implementing this algorithm.

This work aims to provide a BST implementation that is lock-free and efficient even with a large number of Add() / Remove() / Contains() operation in use by multiple concurrent threads. This paper, of course, does not stand alone as there are other proposed solutions, such as work by Zhang and Dechev [8], that also aim to provide an efficient binary search tree. The work by Zhang and Dechev, however, states that skip-lists probabilistically serve as a formidable alternative to a self-balancing binary search tree. This is a valid proposal but there is a problem with these trees when locating a node. In an attempt to Locate() a node, the traversal operation is working concurrently with other Add() / Remove() operations that may be running on separate threads. When Locate() fails to find the key, it is unclear to the working thread whether or not the node is not present or if it has been relocated in a balancing act. To remedy this, the implementation uses a flag property of nodes to allow marking a logically-deleted node upon Remove(). This is a performance-increasing property the algorithm proposed by Chatterjee, Nguyen, and Tsigas has over other proposed lock-free binary search tree algorithms [7].

III. TECHNICAL APPROACH OVERVIEW

For our implementation, we decided to use Java to implement the data structure as it proved simpler to work with when compared to other alternatives such as C++. Lower-level languages like those in the C family tend to bring

hurdles, primarily in regards to memory management. A benefit Java has over the other languages and a primary reason why we went with it for our implementation is its own garbage collector. Memory management is a much simpler task in Java and it is quicker to write safer, error-free code. Additionally, the members of our team are much more familiar with the Java language and are more comfortable writing Java code. We attempted to replicate and re-implement the binary search tree algorithm specified in the paper, using the provided pseudocode as a starting point and skeleton. As development transpired, we noted the pseudocode contained ambiguity errors and concerns that had to be addressed, as described in the Replicability section of this paper. Correctly parsing the pseudocode and making judgement calls regarding proposed functionality of some obscure methods may have impacted the results of this report. In order to produce an accurate reimplement of the algorithm, we wrote all operations and made sure their linearizability was unaltered. We used the Java Atomic library, which is part of the Java standard library, to perform the atomic operations supported by multi-core processors and used in the publication. To ensure linearizability, we maintained the use of a single-word atomic compare-and-set instruction per operation.

IV. TECHNICAL DETAILS

The original algorithm's intended functions for each of the following operations is outlined below. In general, the idea is to maintain a backlink and an order link in each node. The order links, when unraveled, cause the tree to form a linked list, where the nodes are in numeric order. The back-links are simply pointers from the child elements to each parent in the tree.

Fig. 1. A demonstration of how order-links are established. The dotted lines represent the order-links of the tree. [1]

A. Remove()

In order to begin removal, first nodes must be categorized into three distinct categories. Category 1 nodes have their order-links pointing to themselves. They are removed by simply changing the incoming parent's left child to point to the node's right child. Category 2 nodes have the previous order-link pointed at them by their left child directly. They are removed by changing the incoming parent's left link to point to the node's left link and modify the order link to point to the one that its right child is pointing to. Finally, Category 3

nodes are categorized by having an incoming order link from a distant left child. When a Category 3 node is removed, it is replaced by its parent in terms of both its position and its order-links. The left link of the removed node then fills the position of the removed node.

Fig. 2. An example of categorization of nodes for removal. Category 1 nodes are represented in blue; Category 2 nodes are represented in yellow, and Category 3 nodes are represented in red. [1]

B. Add()

To add a node to the tree, only a single pointer needs to be modified: the order-link of a previous node in the tree. Chatterjee et al [1] reason that this makes it no more complex than performing an add operation in a singly-linked list.

C. Locate()

In order to perform a Remove(), Add() or Contains() operation, a Locate() operation must be performed in order to find the node. Locate() takes three parameters: a current node reference, a predecessor node reference, and a key. If the current node's key matches the key parameter, the function returns a 2 (specifying that the key was found). Otherwise, it performs a comparison. If the key was determined to be less than the current node's key, the algorithm will proceed left, keeping track of the previous and current nodes. Otherwise, it will proceed right. Whichever direction it takes, the algorithm will clean up nodes that are flagged for removal. The authors reason that this will prevent contention in the case where a traversal is taking place but a node was already deleted.

D. Contains()

The Contains() operation simply performs a Locate(). If it was able to be found, then the function returns true, or false otherwise.

V. IMPLEMENTATION

The algorithm designed in the paper is different from the authors GitHub implementation. In the paper, there exists a pointer to a backlink but is removed in this implementation. Many functions described in the original specification of the algorithm are not found and additional functions that are not described in the paper are added. The author does state that this is a modified version of the papers implementation. This served as a guide for us to verify the publication as we implemented the data structure.

VI. STATE OF THE ART

The original paper did not have any testing benchmarks. We do not know how it would have compared to existing solutions at the time. The original implementation of the binary search tree is nowhere to be found. We tested the authors modified implementation with another binary search tree that were presented in 2014. It would appear as though several advancements have been made since the publishing of this paper; for instance the binary search tree described in [7] seems very promising in terms of performance. The NM[!] lock-free binary search tree (NMLFBST) tree performed 30x better than the author's modified algorithm.

VII. REPLICABILITY

We faced difficulty replicating the pseudocode for the algorithm's operations provided in the specification [1]. There are parts where usage of the proposed methods do not match the declared prototypes in some cases and correctly match in others. There are issues with method names that are randomly declared in the pseudocode causing ambiguity causing method functionality that is not defined in specification and only in found in pseudocode and not explained why. There are also several variables in the pseudocode that are not declared anywhere but are accessed and cases of variables that are accessed outside their respective scope leading to confusion with expected variable states. We deem the publication unclear and hard-to-replicate which is troubling given the algorithm's subpar performance. We are not confident in the authors claims of efficient lock free binary tree. There are no benchmarks performed in the paper. We could not find any implementation of this original algorithm, and we were unable to produce it ourselves. The authors modified algorithm performs much worse than binary search tree algorithms that were also published in 2014. Additionally, we seemed to encounter a race condition where the Remove() operation did not successfully delete elements from the tree, although it is possible that this was missed in the author's original testing, as there are no benchmarks to go off of. A few of the same authors in [7] also contributed on this particular algorithm, and some of the benchmarks performed in that paper were done on much more powerful hardware than we had access to.

VIII. SOFTWARE TRANSACTIONAL MEMORY

For the software transactional memory (STM) implementation of the Binary Search Tree, we chose another algorithm that the author implemented in their GitHub page. Software transactional memory serves as a software-based approach to grouping atomic operations into transactions and facilitating their linearizability. We re-implemented the NMLFBST into a software transactional memory-aware algorithm because it had less linearization points than the author's HelpAwareInternalLFBST algorithm. Our implementation of Software Transactional Memory utilizes the Deuce library. Deuce is capable of making any Java class into one that supports STM by providing an @Atomic Java annotator to use on functions that are treated as transactions. By wrapping larger blocks of

Fig. 3. A benchmark of equal adds to removes. The non-STM version of this benchmark performed much worse than the STM version.

Fig. 4. A benchmark consisting of three times as many adds over removes. Once again, the non-STM version of this benchmark performed dramatically worse than the STM benchmark.

code in @Atomic, we can vary the transaction size. Deuce does this by acting as a Java agent. However, it is quite old and requires us to use an older version of the Java Runtime Environment and Java Development Kit, specifically Java 8. In our tests, we had two variations of transactional memory - one with larger transactions (by wrapping entire operations in transactional objects) and one with smaller transactions (by wrapping the compare and set operations in transactional objects). Our benchmarking seemed to suggest that the larger/coarse transactions performed better than the smaller/finer transactions. This was contradictory to what we had expected, as larger operations are more sequential and thus, should perform worse on a parallel system.

IX. EXPERIMENTAL BENCHMARK RESULTS

Our benchmarks were performed on a machine with 16 gigabytes of RAM, and an AMD Ryzen 1700x 3.4ghz 8 core processor, with 16 megabytes of L3 cache and 4 megabytes of L2 cache.

The following are for the modified version of this algorithm as presented by the author. Note the scale of the Y axis in each of these benchmarks. In general, what we observed is that the STM version of this algorithm performed much better than the original implementation, which runs counter to conventional wisdom regarding Software Transactional Memory.

The following are versions of Chatterjee et al's Help Optimal Binary Search Trees as described in [7]. Note the difference in scale of the Y axis of this algorithm when compared to the ones listed prior.

X. CONCLUSIONS

The algorithm as described in the paper most likely does not function as the author claimed. In fact, we aren't certain

Fig. 5. A benchmark consisting of less adds than removes.

that the algorithm ever operated as described in the paper, and even modified versions of this algorithm seem to perform worse than already-existing solutions. We remain uncertain of the algorithm's utility, although there has yet to be a working version of the original, unmodified algorithm as described in the assigned paper. Additionally, Software Transactional Memory seemed to work better than the traditional method of performing this algorithm, which is counterintuitive.

Fig. 6. A benchmark of equal adds to removes.

Fig. 8. A benchmark consisting of less adds than removes.

Fig. 7. A benchmark consisting of three times as many adds over removes.

REFERENCES

- [1] B. Chatterjee, N. Nguyen, and P. Tsigas, "Efficient Lock-free Binary Search Trees," in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, ser. PODC '14. ACM, 2014, pp. 322–331. [Online]. Available: <http://doi.acm.org/10.1145/2611462.2611500>
- [2] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank, "Wait-free linked-lists," *ACM SIGPLAN Notices*, vol. 47, no. 8, p. 309, Sep. 2012.
- [3] Mikhail Fomitchev and Eric Ruppert, "Lock-free linked lists and skip lists," in *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing - PODC '04*. ACM Press, 2004.
- [4] Hakan Sundell and Philippas Tsigas, "Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap," in *OPODIS*, 2004.

- [5] Maged M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures - SPAA '02*, ser. SPAA '02. New York, NY, USA: ACM Press, 2002, pp. 73–82.
- [6] Håkan Sundell and Philippas Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609–627, May 2005.
- [7] I. W. Bapi Chatterjee and Philippas Tsigas, "Help-optimal and Language-portable Lock-free Concurrent Data Structures," Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, Tech. Rep., 2016.
- [8] D. Zhang and D. Dechev, "An Efficient Lock-Free Logarithmic Search Data Structure Based on Multi-dimensional List," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2016, pp. 281–292.