Implementation of a Non-Blocking Binary Search Tree in C++ Final Report

Akshay Jajoo, Devesh Kumar Singh, Sergei Uversky

Purdue University

May 05, 2016

1 Introduction

Sequential data structures are unable to take advantage of the unique capabilities of modern shared-memory multiprocessor machines, leading to poorer-than-optimal performance. This has brought about a need for concurrent data structures in shared memory, through which threads can communicate and synchronize effectively, allowing programs to harness the power of concurrent computing. [6] In this course project, we aim to implement a non-blocking concurrent version of one of the most common data structures: the binary search tree.

2 Motivation

Throughout the course of this semester, we have studied various approaches to making concurrent data structures like stacks, queues, sets etc. satisfying both lock-based and lock-free liveness guarantees. Although it is relatively simple to reason about and implement lock-based concurrent data structures, they come with a number of drawbacks: most importantly, they introduce a *sequential bottleneck*; further, in the context of this course, they are useless for anything but the most naive (crash-free) failure model, as a process that holds a lock and crashes while in the lock's critical section renders progress impossible. Thus, when dealing with distributed systems, we ideally seek data structures that satisfy non-blocking liveness guarantees such as lock-freedom and wait-freedom, as these are more applicable in a realistic distributed setting with non-trivial failure models.

There has been a good amount of progress on making non-blocking implementations of simple data structures like arrays, stacks and queues, but very little for binary search trees. To this end, we seek to implement a non-blocking version of a binary search tree in a non-managed language (C++). As far as we know, no concrete implementation for non-blocking BSTs exists in a non-managed language. We hope that by tackling such an implementation in a non-managed language, we can identify some non-trivial implementation-specific pain points that are not identified in the theoretical discussions, as well as benchmark performance.

3 Related Work

The bulk of our project is aimed at implementing the theoretical non-blocking binary search tree found in the work by Ellen, Fatourou, Ruppert and van Breugel. [2] In this work, Ellen et al. outline the theory and provide the pseudo-code for the first known implementation of a non-blocking binary search tree. Additionally, they provide a proof sketch for the correctness of their work. However, they do not actually have an implementation of their work available in the paper, and thus there are no benchmarks comparing this implementation to other state-of-the-art implementations, either sequential or lock-based concurrent.

Lock-based concurrent BSTs have been tackled in theory as early as 1978; see the work done by Guibas and Sedgewick [3] and Kung and Lehman [5]. Implementations of these lock-based BSTs appear in both unbalanced and balanced forms; one notable example of the latter is the Java implementation by Bronson, Casper, Chafi and Olukotun [7].

Non-blocking concurrent BSTs have been described by other researchers since Ellen's work. One example is the work by Chatterjee, Nguyen and Tsigas [1], which includes an analysis of computational complexity.

4 Algorithms

4.1 Problem Formulation

As stated above, we aim to provide a C++ implementation of the non-blocking binary search tree

described in the work of Ellen et al. The crux of the algorithm described in that work is a state setting for each node – if one process is in the middle of running an update operation on a node, other concurrent processes can tell how far along that process is based on the bits of the state setting. The FIND operation is read-only and does not change any child pointers in the node, and thus can be safely run concurrently (i.e. it need not CAS this state setting). The INSERT and DELETE operations first start by marking the nodes which might get affected by the operations, help other concurrent operations on which the operation might be waiting for complete, and then proceed to perform the actual insert and delete. Marking the nodes allows other processes to "pick up the slack" for any stuck/crashed processes, providing a non-blocking liveness guarantee.

The implementation we aimed at involved gathering a deep understanding of both the theory and pseudo-code given in the paper and the practice of non-blocking concurrent programming in C and C++ (e.g. the interface to dealing with CAS cross-platform, etc.) We expected that this in itself will be no easy task, and that it would form the bulk of our work. To this end, we began by handling the comparatively simpler INSERT and FIND operations. We hope to continue by modifying the INSERT operation to maintain at least a relaxed semblance of balance in the tree. Implementing DELETE turned out to be beyond the scope of this project, as it requires a thorough understanding of concurrent memory management and garbage collection; the authors believe that this could be a fruitful avenue for future work.

4.2 Overview

As in the work by Ellen et al, we consider a *leaf-oriented BST*, where nodes with relevant data are found in the leaves of the tree. The keys stored in the internal nodes of the tree are used to direct the search operation that the algorithm conducts to find where to insert a tree. We maintain the invariant that for each internal node n with children $n_l := x \rightarrow left$ and $n_r := x \rightarrow right$, $\text{KEY}(n_l) < \text{KEY}(n) \leq \text{KEY}(n_r)$. (In the original algorithm described by Ellen et al, the keys of internal nodes could potentially not correspond to existing keys in the tree due to deletions; as we do not yet support deletions, our internal node key set is equal to our leaf key set.) The $\text{FIND}(k)$ operation is comparable to its sequential implementation. An $\text{INSERT}(k)$ operation, if it is successful, replaces a leaf with a subtree, i.e. an internal node with two children (see Figure 1, pulled from Ellen's paper).

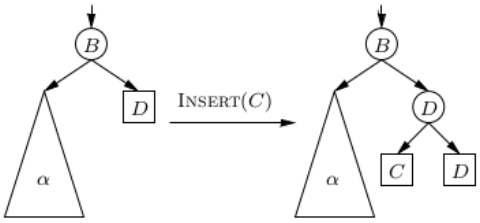


Figure 1: $\text{INSERT}(k)$ (credit to Ellen et al.)

4.3 Sequential BST

We implemented a naive sequential version of a BST for benchmarking purposes. This implementation leverages the interface used by the non-blocking implementation but avoids the use of CAS as well as data structures specific to the non-blocking implementation. The implementation is relatively trivial and does not differ appreciably from any standard BST implementation that can be found in any relevant reference material.

4.3.1 Transactional memory-based extension

As an additional benchmark, we have leveraged our naive sequential implementation to make a concurrent leaf-oriented BST by using the transactional memory constructs of GCC. Recall from class that *transactional memory* is an approach that attempts to combine the performance benefits of concurrency with the simplicity of sequential programming. In this approach, transactions consisting of one or more operations run in isolation and either commit (pushing all of their operations atomically) or abort (discarding the potential changes made by the operations), resulting in a consistent view of memory.

GCC provides support for transactional memory by supplying some annotations and some wrappers around existing sequential code. This is straightforward to implement and again does not need an appreciable amount of elaboration; the only thing to note is that the non-thread-safe portions of the sequential code are wrapped in `__transaction_atomic` blocks.

4.3.2 Lock-based extension

For benchmarking, we will also extend this sequential implementation with locks to make a simplistic blocking thread-safe implementation.

4.4 Non-blocking BST

As of C++11, the C++ standard library provides support for atomic objects via the `<atomic>` library. Instead of using a simple `foo` object,

we can instead use an `atomic<foo>` to guarantee memory ordering in multithreaded contexts. Crucially, these `atomic<foo>` objects support a `compare_exchange_strong`[1] operation, which is a platform-independent implementation of CAS.[2]

The Ellen paper describes an approach in which update operations CAS on a node's status to make progress – in their work, the status is one of CLEAN (no operations pending), MARK (child node currently being deleted), IFLAG (node currently being inserted) or DFLAG (node currently being deleted). The status is used to ensure that the internal state of the tree is consistent during concurrent operations. Nodes progress through the statuses via CAS, ensuring that at most one process is able to further a node's status; because of the internal consistency this imposes, linearizability is possible. An astute observer might note that a process which CASes a node's status from clean to dirty could potentially cause a deadlock if the process dies before cleaning the status again. This is mitigated by the use of *help functions*, in which processes altruistically attempt to help other processes currently attempting an update if they detect that the current node's status is unclean.

Our approach is made slightly simpler due to a lack of focus on deletion. In the original paper, race conditions between two concurrent deletions or a concurrent deletion and insertion necessitated the use of four different states. As we are not concerned about deletion, the only race condition we worry about is when two concurrent insertions happen[3]. Thus, we only require one bit of state, which for simplicity's sake we have modeled as a `bool isDirty`. Each node has its state flag along with an information record (described below) stored in an *update record*.[4]

The SEARCH operation does not differ appreciably from the pseudocode given by Ellen et al; the only difference is that we do not need a pointer to the grandparent of a leaf or that grandparent's update record, as those are only used in deletion in the original algorithm. The INSERT operation is modified only slightly from Ellen et al.'s original pseudocode. The general intuition is that a process first searches to find the leaf node which is to be replaced with a subtree. If the key is already found in the tree, then the process returns `false` as we disallow duplicate keys. If the leaf node is dirty, there is a concurrent insert operation happening, so the process helps that operation complete and then starts over from the beginning. If not, the process creates the subtree that will replace the existing leaf in the tree, and stores it in an *information record* along with the leaf and the leaf's parent. This information record is used by a process entering the help phase and contains all of the information needed to finalize the state of the tree after the insertion takes place. The process then attempts to CAS the parent's state from clean to dirty – if this fails, again another concurrent insertion has beaten this process to the punch, so the process will help the other operation finish. Otherwise, the process is cleared to insert the new key; thus, it "helps itself" finalize the tree structure (to ensure that a concurrent process does not repeat this process's finalization step) and then returns `true`.

The help function is also simplified from Ellen et al. – the only help operation we need is help with insertion. The help function first attempts the CAS that replaces the pointer to the leaf in the parent with the pointer to the new subtree. Then it attempts a CAS on the update record of the parent, from `(dirty, info)` to `(clean, info)`. At the end of each of these CASes, at most one process has done the appropriate changes to the node, so at the end of the help function, the new key has been inserted.

5 Proof Sketch of Correctness

Searching is trivially correct combined with the linearization point given below, as the implementation does not differ from a sequential implementation.

Showing the correctness of INSERT hinges on demonstrating that even on concurrent insertions, the invariants demanded by INSERT are not violated. If there is no contention and an inserting process stays live throughout the insertion, then this argument is trivial – if the value already exists in the tree then `false` is returned; otherwise, the tree structure is updated and `true` is returned. If the inserting process dies in the middle of an insertion and has already dirtied a node, by definition the CAS that it used to dirty the node must have also supplied the node with all of the additional information necessary to finish its insertion. Any process that attempts to insert will then see that the node is dirty and use this information to finalize updating the state of the tree. Hence even if a process crashes after marking a node as `dirty` it doesn't stops another process form continuing.

[1] There is also a `compare_exchange_weak` variant, which is potentially faster at the expense of allowing spurious failures.

[2] As far as the authors understand, the C++ STL CAS implementation uses a hardware CAS where available for 8-byte words or smaller, and uses mutexes to emulate CAS otherwise. More investigation into the implementation is necessary for future work.

[3] Note that any number of FIND operations can run concurrently with up to one INSERT operation; they are trivially linearizable because the FIND operations do not modify the tree.

[4] One future extension of this work could entail simply using a low-order bit on a word-aligned information record pointer to represent this state, guaranteeing that the C++ `atomic` library would use a hardware CAS when prompted.

If there are contending inserts, then the same process as above will take place – one process will succeed in dirtying the relevant node, and then the processes will both attempt to help the first process finish its insertion. Because the help operation uses one CAS to update the tree state from its expected state to its final state and then another CAS to clean the dirty node, each of these steps will be executed no more than once, ensuring that a node won't be doubly-inserted, and, in the non-trivial case where there are more than 0 live processes, a node that started to be inserted will finish inserting.

Due to the structure of the `while` loop to the help function, if two contending inserts are attempting to insert the same value, one will finish first (potentially with the help of the other); the other will then, on its next iteration of its `while` loop, see that the value has already been inserted and then return `false` accordingly.

6 Proof of Linearizability

6.1 Insert

To prove that every call to INSERT is linearizable, we need to show that linearization points exist for following two cases:

1. INSERT returns `true`, i.e. the given item was not in the list before this point and it is inserted to the list after this point.
2. INSERT returns `false`, i.e. the item must already have been in the list before this point.

Let us first consider the case where the value to be inserted does not exist. Let the linearization point for the INSERT operation be the point immediately after the call to the CAS to actually insert the new subtree completes successfully – call this point iCAS. iCAS is immediately followed by another CAS that cleans the node – call this cCAS. If two concurrent INSERT(k) operations for same element both fail to find the element, only one of them will be able to dirty the parent node. The other process's CAS to dirty the parent will fail, so that process will call `helpInsert`; concurrently, the process which dirtied the parent will be trying to help itself insert. Only one of the concurrent CASes for insertion and one for cleaning will then succeed. Once one of the concurrent INSERT operations succeeds, the other process iterate through its `while` loop once more, and will now find k in its search of the existing elements of the tree, so it will return `false` – this second INSERT(k) operation will be linearized here, as explained below. Changes at any other part of the tree are possible but that will not affect the result of this INSERT, as contention only occurs between concurrent local updates. As the current implementation does not support the removal of nodes, there are no other concurrent operations that can alter the path between the root and the target node (searching does not modify the structure of the tree). Success of the iCAS is the point where insertion takes effect; any search which is called before this CAS will come out of its `while` loop without finding it.

We can linearize the case where INSERT(k) returns `false` at the point where `k` was found by the search.

6.2 Search

SEARCH is trivially linearizable at the point that the `while` loop finishes and a node with two null children is reached. Even if e.g. the tree does not currently contain k but there is a concurrent INSERT(k) and SEARCH(k), the insert will either linearize before the search (in which case the loop will detect a non-null child and continue one more step), or after the search (in which case the search will return `false` because at the time of the search, the value did not exist in the tree).

7 Evaluation

We compared the performance of the non-blocking BST with various other BST implementations. To accomplish this, we generated 10 test cases with $10 \times 2^k, k = 1 \rightarrow 10$ inserts, and compared the execution time in of our non-blocking implementation vs others. The thread-safe implementations used 2^k threads in the k^{th} test case, with each thread inserting 10 random values; readings for each configuration were averaged over 10 runs. The sequential implementation is not thread-safe, so all of the insertions are performed on one main thread. Our results seem to indicate that the non-blocking BST is more amenable to parallelization than the than other approaches. We describe the detailed results below. Figure 2 shows the execution time in milliseconds vs. the number of inserts for each of the three implementations, but this is an early result and needs more study.

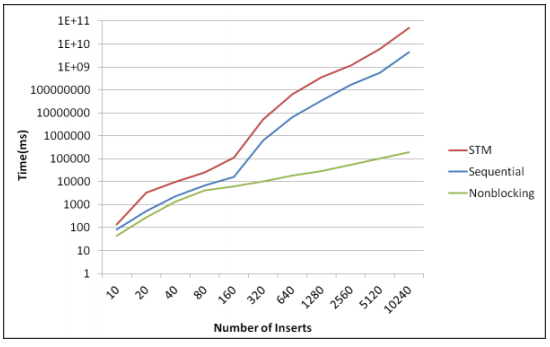


Figure 2: Execution time vs. number of INSERT operations for three implementations

7.1 Synchrobench

We had intended to tool our implementation to work with Gramoli's Synchrobench, a microbenchmark framework for concurrent data structures, but regretfully were unable to do so with any meaningful results. This would be an interesting avenue for future work.

8 Balancing

8.1 Overview

Balancing is essential for maintaining efficient lookup in binary search trees – after all, a pathologically unbalanced binary search tree is just a linked list. Balancing ensures that access and update operations happen in $O(\log n)$ time rather than $O(n)$ time worst-case.

Sequential implementations of BSTs have taken several approaches to balancing. In some cases, each update operation also takes care of balancing the tree (e.g. AVL tree rotations). However, this is not optimal in a highly parallel environment, because the internal structure of the tree is altered drastically on every update. As a result, there are many more potential race conditions, and thus very few parts of the tree that can be trivially accessed concurrently.

8.2 Related work

In [8], Nurmi and Soisalon-Soininen discuss a lock-based approach for implementing chromatic trees, which are a variant of red-black trees with relaxed balancing constraints. Their implementation decouples updates and balancing with an approach similar to the help functions used by Ellen et al. – updating processes leave a record of contextual information that can be used by a separate balancing process. There can be one or more balancing processes, and they can either run in the background concurrently with update processes during execution or can be invoked during downtime, similar to garbage collectors. The authors focus on splitting the balancing operation into several small updates to minimize sequential bottlenecking. This approach seems promising but was not explored further for this project, as Nurmi and Soisalon-Soininen's implementation is lock-based; this would necessitate a large-scale rewrite to work with the lock-free code base.

8.3 Partial balancing in AVL trees

8.3.1 Overview

Another approach for balancing is proposed by Kessels in [4]. Kessels posits that it is possible to modify the criteria for AVL tree updates to partially balance the tree upon accesses (insertions and searches), instead of doing a full rebalance upon every insertion. Instead of calculating based on the static height of the tree (i.e. the length of a path from the root to the leaf in question), Kessels introduces a metric called *dynamic height*, which has each node maintain a carry bit to determine whether a node contributes to the dynamic height of the tree in that path. In addition to the carry bit, the internal nodes also maintain a ternary balance variable that indicates the relative dynamic heights of the two children of the internal node. These two additional variables lead to three different balancing transformations that happen when particular preconditions are met after access to a particular node.

8.3.2 Technical details

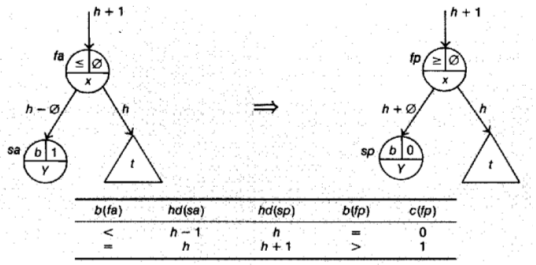
As mentioned above, in this approach each node contains two additional pieces of information: a *carry bit* which indicates that a node does not contribute to the dynamic height of the tree and is propagated up to the root of the tree during transformations, and a *balance factor* which indicates the balance status of the subtree rooted at this node (whether the two children are of equal dynamic height, or if one of the children is dynamically taller than the other). The invariants maintained by this approach are that (a) the dynamic heights of two children of a subtree differ by at most one (zero if the balance factor is zero), and (b) the subtree rooted at the current node has dynamic height 0, or its carry bit is not set, or its subtrees are unbalanced.

When inserting a new node (i.e. replacing an empty leaf with a subtree with two empty leaf children), the new internal node has its carry bit set; the bit either propagates upward toward the root of the tree or disappears, depending on the transformations that are carried out in the partial balancing process. If a node n has its carry bit set but its parent p does not, then one of three transformations is carried out:

1. If p's balance factor is 0 or if n is the shorter child of p, no balancing takes place but the carry and balance bits are updated appropriately.
2. If n is the taller child of p and p and n have equal balance factors, then balancing occurs between two nodes.
3. If n is the taller child of p, and p and n have differing balance factors, then balancing occurs across three nodes.

These operations are illustrated generally in the following three figures, taken from Kessels (they are Figures 2, 3 and 4 respectively in Kessels' paper and

are not our original illustrations). For additional details, please refer to the paper. (Note that the internal nodes are partitioned to have the balance factor in the upper left hand corner, the carry bit in the upper right hand corner and the value in the bottom.)



b(fa)	hd(sa)	hd(sp)	b(fp)	c(fp)
<	h − 1	h	=	0
=	h	h + 1	>	1

Figure 3: Transformation case 1: no balancing but updates to carry bit and balance factor.

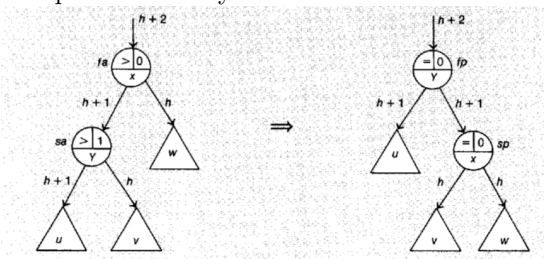
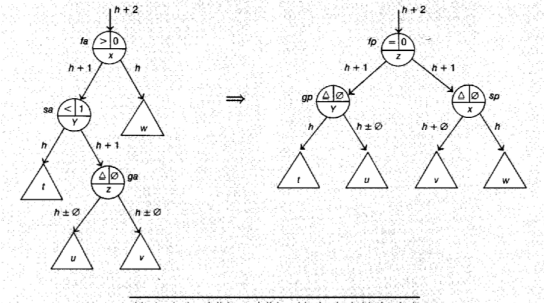


Figure 4: Transformation case 2: balancing across two nodes.



b(ga)	c(ga)	hd(u)	hd(v)	b(gp)	c(gp)	b(sp)	c(sp)
<	0	h − 1	h	>	0	=	0
=	0	h	h	=	0	=	0
>	0	h	h − 1	=	0	<	0
<	1	h	h + 1	=	0	>	1
>	1	h + 1	h	<	1	=	0

Figure 5: Transformation case 3: balancing across three nodes.

8.3.3 Implementation

The partial balancing approach shown by Kessels has similar locality to the BST implementation by Ellen et al. – that is, unlike a standard AVL tree where each insertion causes a full rebalance of the tree, accesses to a node cause a rebalancing in the subtree rooted (at highest) at the grandparent of that node. This means that the framework used by Ellen et al. – which, for deletion, changes the status of the grandparent of a node to prevent race conditions – can be leveraged to integrate this approach. The authors attempted this integration but were unable to complete it in the remaining time. However, some food for thought regarding this integration follows.

8.3.4 Thoughts and further work

To fit this balancing scheme in with the Ellen implementation, most likely insertions would have to dirty not just the parent of the newly-inserted node, but also the grandparent, as the subtree rooted at that grandparent might be involved in the third transformation case seen in the paper. However, the information necessary to complete a rebalancing operation – including the additional carry and balance variables introduced by the balancing approach – could be added to the update records for each operation, allowing the use of the same "helping" approach to finish any rebalancing currently in progress before continuing with the next rebalance. Further work would then be needed to investigate optimizations to this helping process, e.g. a process might be able to perform less balancing if it sees that the current partial balance that it is helping execute actually leaves the tree in a state amenable to trivial insertion afterwards. This scheme would help make sure the tree continues to allow $O(\log n)$ access times, but will hamper concurrency somewhat, as there is more work that needs to be done over a slightly larger region of the tree due to the overhead of the partial rebalancing.

One difficulty that would be encountered in adapting this scheme to the implementation of Ellen et al. is that the BST described in Kessels' work is not leaf-oriented, so changes would have to be made to reflect this.

One important avenue of investigation when integrating this approach is to consider the performance tradeoffs of running the partial balancing transformations on all access operations, as opposed to just inserts. In the paper, the author posits partial balancing after every access; this approach will make the tree converge to a balanced form quicker, but has the drawback that concurrency might be significantly reduced, especially in a single-writer, multiple-reader case – searching will no longer be trivially linearizable and multiple searching threads will no longer be able to run fully in parallel. On the other hand, partial balancing after insertions only would make convergence slower and would still slow down some searches (those local to the subtree rooted at the grandparent of any node which is concurrently being inserted); however, use cases with infrequent insertion and frequent searches might benefit from the additional concurrency allotted to search operations in this context. Depending on the results of performance testing, this might be given as a flag for a user to set depending on use case.

9 References

[1] B. Chatterjee, N. N., and Tsigas, P. Efficient lock-free binary search trees. *PODC* (2014).

[2] F. Ellen, P. Fatourou, E. R., and van Breugel, F. Non-blocking binary search trees. *PODC* (2010), 131–140.

[3] Guibas, L. J., and Sedgewick, R. A dichromatic framework for balanced trees. *Proc. 19th IEEE Symp. on Foundations of Comp. Sci* (1978), 8–21.

[4] Kessels, J. L. W. On-the-fly optimization of data structures. *Commun. ACM 26*, 11 (Nov. 1983), 895–901.

[5] Kung, H. T., and Lehman, P. L. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.* (1980), 5(3):354–382.

[6] Moir, M., and Shavit, N. Concurrent data structures.

[7] N. G. Bronson, J. Casper, H. C., and Olukotun, K. A practical concurrent binary search tree. *PPoPP* (2010).

[8] Nurmi, O., and Soisalon-Soininen, E. Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica*.