Implementation of fine-grained locking and lock-free binary search tree

Sean Zhou, Ye Lu

Abstract

In this project, we implement three types of concurrent binary search tree, each uses coarse-grained locking, fine-grained locking and lock-free version. After that, we create our own correctness and performance test to benchmark and compare their performance. Specifically, we benchmark their performance and analyze the influence of different types of workloads, different types of tree structures and different types of access patterns. Finally, we come up with the conclusion on the concurrency performance of these three types of binary search tree.

Background

Binary search tree

Binary search tree (BST) is a kind of efficient data structure to support search, insert and remove operations within it. It takes O(logn) time to do all the three operations. And the tree structure enables us to operate concurrently on different subtrees without having correctness issues. If designed properly, we can not only allow concurrent read operations. We can also allow concurrent read-write, write-write operations on the BST, which make BST more efficient under high concurrency workloads.

Operations

You can access and modify the binary search tree with mainly three operations. The first one is contains, which searches for the existence of a key in the tree. Contains is a read-only operation, and won’t modify the structure or content of the BST. The second one is insert, where you can insert a new key into the BST. Insertion will modify the structure of a BST. In our implementation, there is no rebalance operation, so the new key will be faithfully inserted into the tree as a leaf node. The last operation is deletion. In deletion, there are two scenarios to think about. One is to delete a node with zero or one child. The other one is to delete a node with two children. In this case, you will need to find the next largest key in this subtree and relocate that node to be the new root. And both scenarios will modify the structure of the BST. Similarly, we will not rebalance the BST after a deletion.

Data dependency

As mentioned in the previous section, for insert and delete operations, we need to modify the structure of BST, which makes concurrent operations a great challenge. For example, an ongoing insert operation expects the parent node to insert exists during the whole process, while this might not be true when there exists concurrent deletion. So there are delicate implications between child-parent nodes and also other nodes along the traversal paths from tree root down to the leaf node.

Implementations

Coarse-grained binary search tree

Data structure

Our coarse-grained BST is a sequential BST augmented with a global lock, implemented via a C++ mutex, on the tree. In our code this is implemented in the CGBST class.

Operations

The operations — add, remove, find — are identical to the sequential operations except they acquire the global lock at the beginning and release the global lock before returning.

Fine-grained binary search tree

Key data structure

Our fine-grained BST is a sequential BST in which every node is augmented with a lock implemented via a C++ mutex, in order to enable hand-over-hand locking. And to deal with the empty tree, there is one additional lock that is acquired before accessing the root node and dropped upon acquiring the root node's lock.

Operations

All operations are based on hand-over-hand locking nodes when traversing the tree: a thread holds on to its current node's lock and tries to acquire the lock of the child it intends to visit next. Once the child's lock is acquired, the current node's lock can be released, and the child becomes the new current node, so on and so forth. Specifically:

- Insertion: we traverse hand-over-hand until we find the leaf to which we insert the new node. As insertion only concerns the content of one node — the leaf that we insert to — during traversal we can release the current node as soon as we acquire its child's lock. So at most two nodes are locked, and usually only one node is locked.

- Find: find is almost identical to insertion. We only concern the content of one node, so at most two nodes are locked, and usually only one node is locked.
- Deletion: deletion requires control over the node we need to delete and its parent, so as we are traversing the tree to find the node to be deleted, we must hold on to the current node and its parent's locks while attempting to acquire the next child's lock. This means usually two and at most three nodes will be locked in this phase. Once we've found the node to delete, if it has less than two children, we can complete the operation by modifying its parent's pointers. Otherwise, if the node-to-delete has two children, we need to continue through its right branch to find its in-order successor, swap values with the node-to-delete, and delete the in-order successor. This means we'll hold on to the node-to-delete and its parent, while traversing the right subtree with hand-over-hand locking, meaning usually four and at most five nodes will be locked in this case.

Deadlock Safeness

This fine-grained BST is deadlock safe because locks on a node would only depend on locks on its descendants, therefore there could never be a circular dependence. In the worst case, the deepest node will be unlocked first, then the next, so on and so forth until all is unlocked.

Starter Code

We started with the sequential BST implementation on GeeksForGeeks (https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/). However, we modified it entirely by changing the implementation from recursive to iterative and adding synchronization. The current code no longer has any similarity to the starter code aside for some comments that are kept.

Lock-free binary search tree

Key data structure

We represent the binary search tree with class BST, which has one field to represent the tree root and three tree operations (contains, add, remove). We use class *Node* to represent tree nodes. Each node has four volatile fields:

- Key: an integer represent the node's key value;
- Left: a pointer to node's left child;
- Right: a pointer to node's right child;
- Op: a pointer to the operation currently performed on the node.

The *Operation* is an interface type, which has two concrete classes that inherit it, representing two types of operations that can be performed on the node. And an object of the operations contains all the information required to perform the operation. The first one is class *ChildCASOp*, which represents the operation to modify one of node's child pointers. It has three fields:

- Is_left: is the operation is performed on the left child pointer;

- Expected: the original value of that child pointer;
- Update: the new value that child pointer will be updated to.

The other operation is called *RelocateOp*. It represents the operation to remove a node with two child pointers and relocate the next largest key to replace it. It has five fields:

- State: an integer represent current state of the operations, which takes three value
    - 0, ONGOING: the initial state meaning a relocating operation is been performed;
    - 1, SUCCEED: the relocation operation has succeed;
    - 2, ABORT: the relocation operation has been aborted.
- Dest: a pointer to the node to be removed;
- Dest_op: a pointer to dest's operation field;
- Key_to_remove: an integer represents the key to be removed;
- key_to _put: an integer represents the next largest key value.

There is also one helper function *find,* which helps traverse the binary search tree and locate a target key value. If such a key is not found, it will return the node to which the key value will be inserted to. This helper function takes one integer input parameter as key value to find, and four parameters as output value. It will return an integer as the find status, which has four states:

- 0, FOUND: the target key is found, and reference to that key return with *curr*;
- 1, NOTFOUND_L: the target key is not found, and if inserted, it will be the left child of *curr*;
- 2, NOTFOUND_R: the target key is not found, and if inserted, it will be the right child of *curr*;
- 3, ABORT: the traversal couldn't finish for some concurrent ongoing operation on the tree.

ABA problem avoidance

As mentioned during lecture, lock-free data structure can suffer from the ABA problem if it is not carefully handled. The ABA problem happens when the system reuses the freed memory, and by checking the pointer to that reused memory unchanged to determine there is no concurrent modification on the data structure.

The update on child pointers and operation fields need to be protected by CAS operations, so we need to make sure those parts cannot suffer from the ABA problem. For any update on child pointers, we need to make sure there are no concurrent updates, this is done by checking if child pointers are marked as NULL. For operations field update, we need to make sure the pointer is marked as NONE. We cannot use another field to represent that status since that way we need to make the check of two fields atomic (we can use "wide" CAS for this purpose on x86 machines, but it is not widely supported on other machines). Instead, we embedded that field into the pointer by making use of the lowest 2 bits of that pointer as flag bits. We also need to make the pointers unique, so we cannot reuse the memory space allocated for child nodes and operations. This way, we successfully avoid the ABA problem in our data structure.

Correctness test

The correctness test suit is designed around catching the following race conditions:

- Insertion-Insertion race: this happens when two threads insert a node to the same place without synchronization (e.g. one thread inserting 3 and another inserting 4 to a leaf node with value 2). Both threads will update the same leaf node they insert to, resulting in one of the two insertions to be lost. To catch this race condition, we make 64 threads insert 1000 nodes each concurrently, and afterwards we check that no values inserted are lost.
- Deletion-Deletion race: this happens when two neighboring nodes are being deleted concurrently. For example, consider two threads deleting 2 and 3 each in the branch 1-2-3-4. The thread deleting 2 will modify node 1 to point to node 3 then deallocate node 2, while the thread deleting 3 will modify node 2 to point to node 4 then deallocate node 3. Thus the following events could happen: thread one reads that node 3 is node 2’s right child; thread two points node 2 to node 4 and deallocates node 3, completing the deletion of node 3; thread one points node 1 to node 3, not knowing that node 3 has already been deleted. The result at the end is that node 4 and all its descendants are lost, and node 1 is pointing to a garbage value that could potentially lead to a seg fault later. To catch this race condition, we start with a tree of 64000 nodes and make each of 64 threads delete 999 values from the tree. Afterwards, we check that the tree contains exactly the 64 values that shouldn’t be deleted. This test will thus catch both false positives and false negatives.
- Deletion-Insertion race: this happens when a leaf node that is being inserted to is deleted. The insertion is lost because it is inserted to a node that is deleted, and the thread deleting the node never noticed the insertion because it didn’t complete. To catch this, we make 64 threads each insert 1000 elements and delete all the even numbers, and do so in batches of 1 in order to ensure a mix of insertion and deletion operations. Afterwards, we check that all the odd numbers are kept and all the even numbers are deleted.
- Deletion-Find race: this happens when a child node that Find() is going to visit next is deallocated by Delete() before it could be visited, resulting in Find() reading garbage values and potentially getting segfault. To catch this, we repeat the test for catching Deletion-Insertion race, but instead of searching for the values afterwards we mix in the searches as deletion and insertion is still happening.

Since concurrence bugs do not necessarily manifest in one test, each test case is repeated 10 times in one trial of the correctness test.

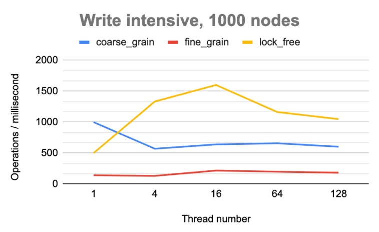
Results and Analysis

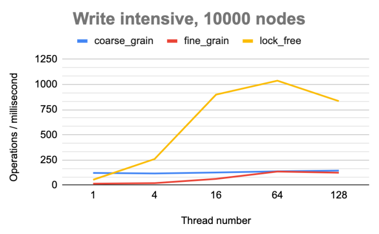
Benchmark environment

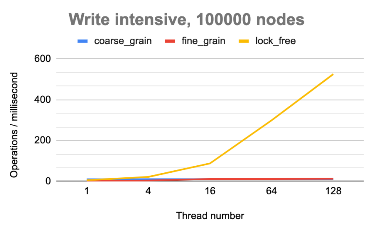
For all the test cases in this section, we are performing the benchmark on PSC machines.

Write intensive workload

To test the performance of all three versions of BST under write intensive workload, we create a BST with {1000, 10000, 100000} nodes, and then each thread will operate on the pre-constructed tree with 50% add, 50% remove. The benchmark result is shown below:





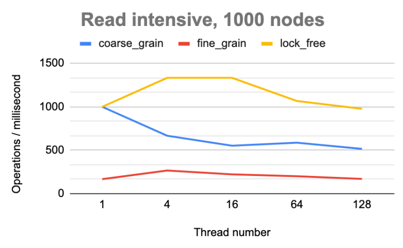


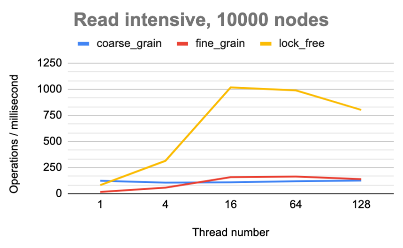
If we look across all three results, lock-free BST scales best as we increase the number of threads, especially for BST with more nodes. In theory, lock-free data structure should perform somewhere similar to fine-grained lock data structure. We think part of the reason is hand-over-hand locking used in fine-grained BST is not the optimal one, since if we lock an upper-level node, we cannot access all nodes below that, even those operations won't conflict. Another reason can also be that the PSC cluster is a shared resource, we are not guaranteed to fully occupy the resources and the working threads can get per-empted or scheduled out.

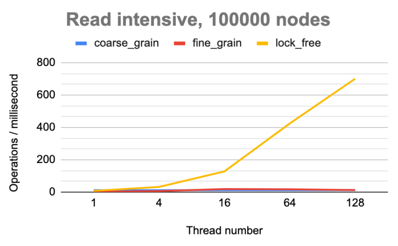
One interesting finding is that for small size BST and low thread number scenarios, the coarse-grained BST is actually better than fine-grained one. First, we think fine-grained works badly in small size BST because all writes end up contending for upper level nodes, which makes it degrades to coarse-grained one. Plus the overhead to lock/unlock every pair of nodes as we go down. And if we increase the size of the tree and have a higher thread number, more parallelism in the lower part of the tree will make up for fine-grained locking overheads so we get better performance than coarse-grained BST.

Read intensive workload

To test the performance of all three versions of BST under read intensive workload, we create a BST with {1000, 10000, 100000} nodes, and then each thread will operate on the pre-constructed tree with 9% add, 1% remove, 90% find. The benchmark result is shown below:





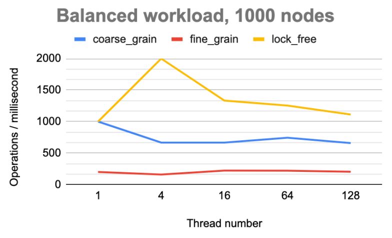


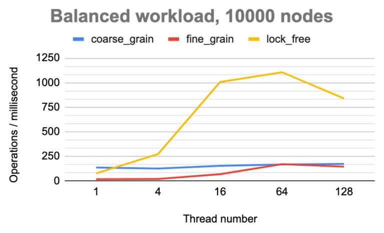
As shown above, we have similar results as in write intensive workloads. Where lock-free version scales quite well as we increase the size of BST and fine-grained BST will not overperform coarse-grained until large BST and high thread number. We think the reason is similar as we said in the previous section.

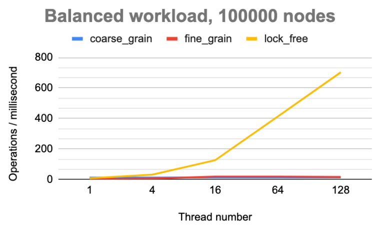
It’s not surprising to find coarse-grained BST performs similarly under read and write intensive workload. Since the overheads to lock for read and write operations are similar. And the same applies to fine-grained BST. However, we do expect to see higher operations per milliseconds for lock-free BST under read intensive workload than write intensive workload. Since we will don’t need to frequently restructure and relocate nodes under read intensive workload, which will incur more CAS operations. But we didn’t see the increase until the largest BST.

Balanced workload

To test the performance of all three versions of BST under read-write balanced workload, we create a BST with {1000, 10000, 100000} nodes first, then each thread will operate on the pre-constructed tree with 20% add, 10% remove, 70% find. The benchmark result is shown below:



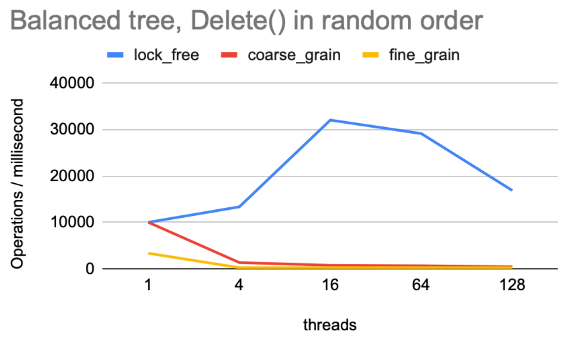


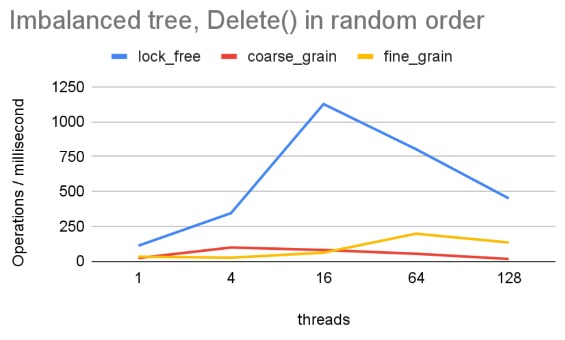


Not surprisingly, the performance under the balanced workload is similar to previous workloads. Where lock-free version scales quite well as we increase the size of BST and fine-grained BST will not overperform coarse-grained until large BST and high thread number.

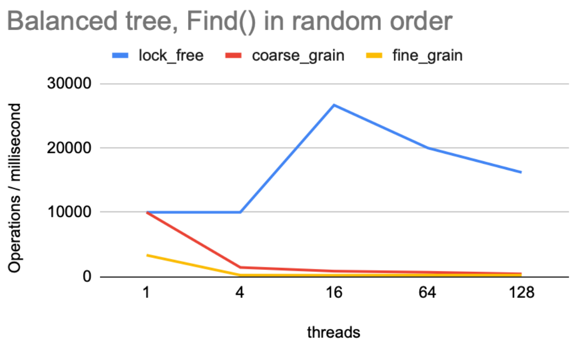
Effects of the tree structure

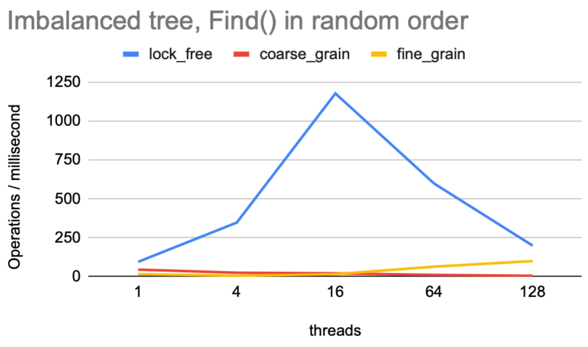
A vanilla BST, without self-balancing, can have different levels of balance depending on the operations performed. To examine whether tree balance has an effect on the different BSTs, we make each thread delete / find 10000 nodes on a relatively balanced tree versus an unbalanced tree. The threads use a random access pattern and the trees have size numThreads * 10000. The results are as follows, measured in operations per millisecond:





For deletion, the interesting observation is that fine-grain BST outperforms coarse-grain BST on an imbalance tree on high thread count, but not on a balanced tree. This is counterintuitive at first because a more balanced tree should have more branches to parallelize on. However, we observed that a balanced tree has much more nodes with two children than an unbalanced tree. Therefore, for a balanced tree, there will be more deletions performed on nodes with two children, which is the worst operation for our fine-grained BST, as up to five nodes can be locked when searching for the in-order successor to replace the node to be deleted. In addition, we came up with the hypothesis that threads will more likely block each other by locking nodes in a balanced tree because doing so will more likely block two subtrees instead of one.

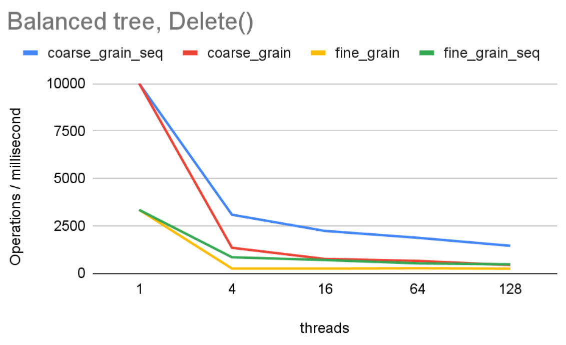


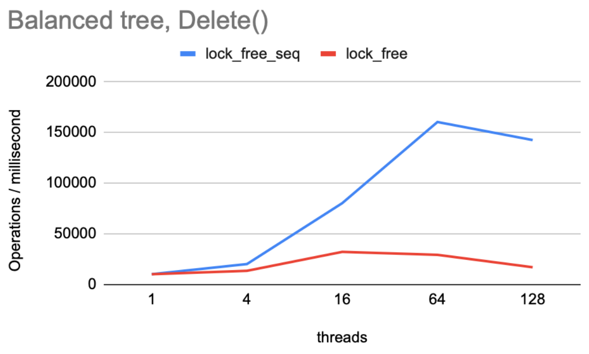


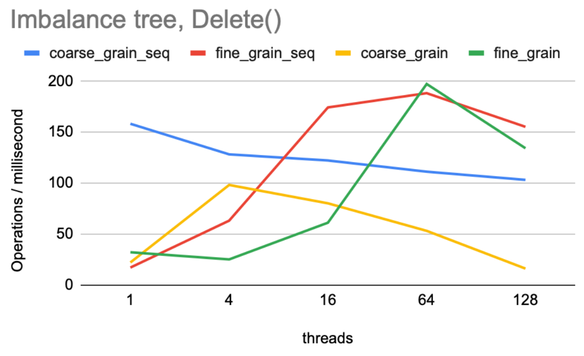
For find, we observe the similar phenomenon that fine_grain outperforms coarse_grain on the imbalance tree with high threads, but not on the balanced tree. Unlike Delete(), Find() doesn't have a costly special case. It will usually lock one and at most two nodes during the operation. So this further supports our hypothesis that node locking could lead to excessive traffic especially if the tree is more balanced.

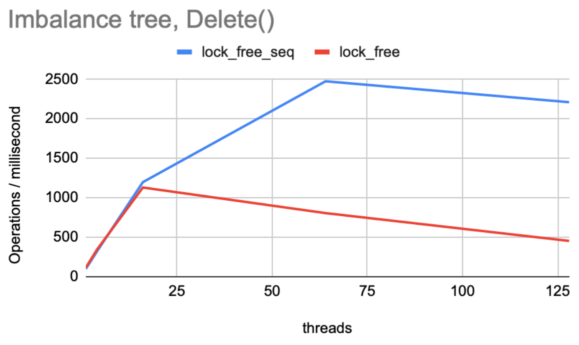
Effects of access patterns

We also wanted to see if the access pattern (sequential access VS random access) has an impact on lock contention, so we rerun the tests in the previous section, but this time making each thread access elements sequentially in order of magnitude rather than randomly. The results are as follows, we cross compared with using the random access pattern





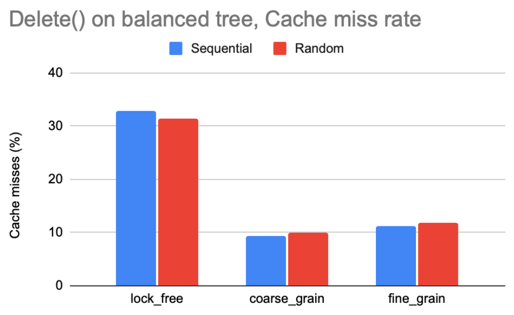


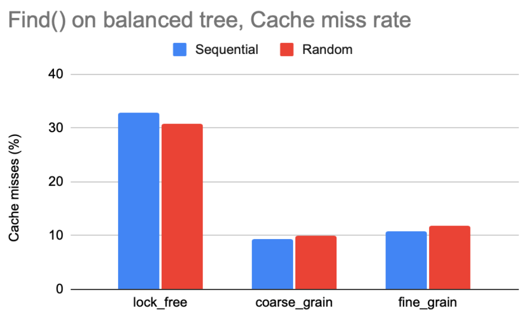


Regarding balanced VS unbalanced tree, using a sequential access pattern gives us the same conclusion as we get in the previous section using a random access pattern. So we don't see significant differences in this aspect.

However, when we cross compare the sequential access pattern and the random access pattern on the same tree, we see the sequential access pattern is almost always faster than their random access pattern counterpart (e.g. sequentially accessing a balanced lock-free tree is much faster than randomly accessing it). This is true even in the case where the tree is created by inserting nodes in random order.

We suspect that this difference is due to cache misses rather than synchronization, because a sequential access pattern could have better temporal locality — it will reuse previously visited ancestors better than a random access pattern. To check this, we used Perf to measure cache miss rates and obtained the following results:





We can see that, contrary to our belief, there is no significant difference in cache miss rate between sequential and random access patterns. So the performance difference we saw might indeed be due to synchronization: even on a BST whose nodes were inserted in random order, a sequential access pattern may result in less contention on nodes between threads compared to a random access pattern, thus resulting in less synchronization time.

Another interesting observation from the result is that operations on Lock-Free BST have significantly more cache misses than Fine-Grain or Coarse-Grain BST. We believe this is due to the design of Lock-Free BST in that often multiple threads would contend to update a node using compare-and-swap, thus resulting in more cache misses due to inherent communication, as processors contend to gain exclusive access to the cache line.

Conclusions

In general, the Lock-Free BST is significantly faster than the lock based BSTs. When the tree is large enough, Fine-Grain BST usually can make up for the overhead of locking nodes and beat Coarse-Grain BST.

There is not much influence on fine-grained and coarse-grained BST if we vary the type of workloads, since the overheads for read and write operations are similar. While for lock-free BST, we do observe better performance when intensively reading larger BST.

On very balanced trees, the Fine-Grain BST's parallelism advantage is somewhat reduced due to deletion happening more often on nodes with two children, which is the most costly case for our Fine-Grain BST. It also suffers from blocking both subtrees when locking nodes.

A sequential access pattern is faster than a random access pattern on all three implementations. Data shows that it's not due to cache miss, suggesting that a sequential access pattern can lead to less lock contention.

Work Distribution (50%-50%)

Sean	Ye
● Implemented Fine-Grain BST ● Implemented correctness tests ● Implemented performance tests to benchmark operations on trees with different balance and using different data access patterns	● Implemented Lock-Free BST ● Setup helper functions for ease of writing correctness tests ● Implemented performance tests to benchmark operations on different types of workloads

References

[1] Shane V. Howley and Jeremy Jones. 2012. ***A non-blocking internal binary search tree***. In Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '12). Association for Computing Machinery, New York, NY, USA, 161–171. DOI:https://doi.org/10.1145/2312005.2312036
[2] "Binary search tree," *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/binary-search-tree-data-structure/. [Accessed: 09-Dec-2021].