Near Wait-free Binary Search Tree

CS6868 : Concurrent Programming
Course Project

By

Aishwarya P (CS11B004)
Aravind S (CS11B033)
R Srinivasan (CS11B059)
Adit K (CS11B063)

Introduction

In this project, we have attempted to create a reasonably simple implementation of a wait-free Binary Search Tree (BST). Although this is not true in its entirety, we feel it should provide a good level of performance. And, the idea seemed very cool!

The first question to ask would be: Why parallelize such a simple ADT?

- It's easier for us to conceive!
- Complex ADTs are serially quick too. So, there is not much point in trying to parallelize them and speed them up. This particular ADT can get unwieldy after several insertions. So, maybe parallelism helps here.

We make an assumption before we begin. Assume that the number of threads coming in with operations to be done on the BSTis bounded by some constant, say N_T. We will refer to such threads as user threads and denote them by T_{op}. Since the number of threads accessing the tree at a time is assumed to be bounded, we can refer to them as $T_{op}^1, \ldots, T_{op}^{N_T}$.
The operations we expect to support are search, insert and delete. We also assume that duplicates are not allowed in the tree.

Structure of a Node

We would like to describe the fields used in a node of our BST, so for easier understanding of code snippets. They are shown, with their intended uses, in the code snippet below -

```
public class Node {
        private volatile int value;  // Value at the node
        public volatile int height;  // Used to balance the tree during clean-up
                                     // The value this holds is not valid except during
                                     // the clean-up operation
        private volatile AtomicBoolean isMarked;   // True if the node is marked for delete
        private volatile AtomicBoolean isSentinel; // True if the node is a sentinel
                                                   // Only the root node can be a sentinel
        private volatile Node left, right;
                                     // Left and right children of the node respectively
        private volatile ElementInserterThread elementInserter;
                                     // Thread that handles inserts to this node
                                     // The value is valid only if the node has
                                     // less than 2 children
```

Also, some statistics about the BST are maintained in a class called *TreeSize*. Their use will become clear as we proceed.

```
/**
 * Class that contains statistics about teh number of different types of
 * nodes in the BST. This is separate from the BST class because many
 * classes need access to these
 * @author Aishwarya P, Srinivasan R, Adit Krishnan, Arvind Sankar
 */
public class TreeSize {
        public static AtomicInteger numNodes = new AtomicInteger(1);
        public static AtomicInteger liveNodes = new AtomicInteger(0);
        public static AtomicInteger deadNodes = new AtomicInteger(1);
}
```

The Delete Operation

Deletion in a BST is quite a costly operation, potentially resulting in significant refactoring of the tree. If any insert operation was being performed concurrently, its region of operation could potentially shift significantly, making it difficult to design a wait-free insertion. So, we thought it made sense to try a lazy paradigm. The delete operation in our BST simply locates the node to be deleted using a standard BST traversal and marks it. The following is the code snippet for delete -

```
        /**
         * Deletes the @param key from the BST
         * @return true if the delete was successful (@param key was found)
         * and false otherwise
         */
        public boolean delete(int key) {
                Node tempNode = root;
                // Indicate that a thread is using the BST and wait till the
                // BST is not being cleaned up
                int sum = 2;
                while(true) {
                        while(numLiveThreads.get() % 2 != 0);
                        if(numLiveThreads.addAndGet(2) % 2 == 0) {
                                break;
                        }
                        sum += 2;
                }

                // Perform a standard BST search and mark the node
                // as deleted if found
                while(true) {
                        if(tempNode == null) { // Key not found
                                numLiveThreads.addAndGet(-sum);
                                                // Indicate that this thread is done
                                return false;
                        }
                        if(tempNode.getValue() < key) {
                                tempNode = tempNode.getRight();
                                continue;
                        }
                        if(tempNode.getValue() > key) {
                                tempNode = tempNode.getLeft();
                                continue;
                        }
                        if(!tempNode.isSentinel().get()) {
                                // Key found in a sentinel node
                                if(tempNode.CASMarked(false, true)) {
                                        // Marked the node for deletion
                                        TreeSize.deadNodes.getAndIncrement();
                                        TreeSize.liveNodes.getAndDecrement();
                                        numLiveThreads.addAndGet(-sum);
                                                // Indicate that this thread is done
                                        return true;
                                } else {
                                        // Marking failed due to contention
                                        // Key was deleted by some other thread
                                        numLiveThreads.addAndGet(-sum);
                                                // Indicate that this thread is done
                                        return false;
                                }
                        }

                        numLiveThreads.addAndGet(-sum); // Indicate that this thread is done
                        return false; // Key not found
                }
            }
```

Barring the modifications to the variable *numLiveThreads*, it can be seen that this is trivially wait-free. We defer the explanation of the use of the variable *numLiveThreads* and why the loop on that variable at the start of the function terminates ina finite number of steps. Under this condition, the delete operation is clearly wait-free.

The delete operation is also linearizable, with the linearization point as the successful / failed compareAndSet operation on the mark of the node (performed by the function *CASMarked*).

The Search Operation

The search operation in our BST is also the standard BST search algorithm. Here is the code snippet for search -

```
        /**
         * Search for the @param key
         * @return true if found, false otherwise
         */
        public boolean search(int key) {
            Node tempNode = root;
            // Indicate that a thread is using the BST and wait till the
            // BST is not being cleaned up
            int sum = 2;
            while(true) {
                while(numLiveThreads.get() % 2 != 0);
                if(numLiveThreads.addAndGet(2) % 2 == 0) {
                    break;
                }
                sum += 2;
            }

            // Standard BST search
            while(true) {
                if(tempNode == null) {
                    numLiveThreads.addAndGet(-sum);
                    return false;
                }
                if(tempNode.getValue() < key) {
                    tempNode = tempNode.getRight();
                    continue;
                }
                if(tempNode.getValue() > key) {
                    tempNode = tempNode.getLeft();
                    continue;
                }

                numLiveThreads.addAndGet(-sum); // Indicate that this thread is done
                return (!tempNode.isSentinel().get() && !tempNode.isMarked().get());
            }
        }
```

Search performs a standard traversal of the BST. If it does not find the node, it returns *false*. If a node is found with that value, search returns *true* if the node is neither marked nor a sentinel, else it returns *false*.

Again, deferring explanations related to *numLiveThreads* and assuming the loop associated with it terminates in a finite number of steps, this operation is clearly wait-free.

The Insert Operation

Our initial idea for insertion was very simple. To make insertion wait-free, we had to tackle the situation of concurrent inserts at the same position in the tree, as this would be the only case where the compareAdnSet operation required to link a node to its parent, could possibly fail repeatedly due to overtaking by other threads. Ironically, to handle this issue, we decided to borrow wait-free queues from *"Wait-Free Queues With Multiple Enqueuers and Dequeuers", by Alex Kogan and Erez Petrank.*

The idea was this - when a thread wants to insert at a position in the tree, it enters into a queue and waits for its turn. Clearly, it will not have to wait for more time than that taken by all other threads and if we ensure that they finish in a finite amount of time, so will the enqueued one. The use of a queue automatically ensures that no thread will overtake another at the same location. Hence it would only be necessary to show that the number of steps required after a thread's request is dequeued is finite, and that the request does indeed get dequeued.

The insert operation could then be described in an abstract manner as follows - there is a wait-free queue associated with each leaf node, when we begin, only the root. For each queue, we have a dequeue operation which is forever running. Whenever the dequeuer has an item to dequeue, the item at the head of the queue is removed and actually inserted as a child to the node with which the queue is associated. Then, the queue is split into two, one for the left node and one for the right each containing items less than and greater than the current item inserted, respectively. The two queues now start running their dequeue threads and the process goes on. Since the queue is wait-free, the insert procedure is also wait-free. From here on, we shall refer to these dequeuer threads as ElementInserters as their job is to actually insert a value into the tree.

The ROOT cause of all our misery

The above naive attempt at a solution turned out to be futile once the implementation began.

Let us revisit the initial solution and see where exactly the approach fails. The idea was to maintain a queue of elements to be inserted at each node where there is vacancy, i.e., nodes of out-degree 0 or 1, if we consider the BST as a rooted directed tree with leaves emerging as outward edges from their parents.

Our goal and gaurentee - wait freedom! Hence, we ensured a FIFO order of the incoming threads. The workers at each of these potential insert spots would inspect the queues continuously. The moment some element is found in the queue, they dequeue the head and carry out its insert operation. Once done, the queue would now have to be either split between the single node inserted and its parent or between the already existing child and the newly created one (in this case, the parent has in some sense been exhausted - its job is done, it can no longer facilitate inserts).

The problem: the splitting of the queue in fact changes the queues present in the structure. For instance, suppose A were the queue at the parent and we created two new queues B and C split from A when a new node x was being created. Let y have been the parent of x at the time and suppose y had no other child then. So A was y's queue. We would assign one of B and C to x and the other to y depending on the relative values of x and y. Whoever comes in after this would realize that A is no longer a valid queue to attack, in fact, that reference would not be present at y. However, it is very much possible that someone decided to insert as a child to y another z when this insert of x was occurring. If z were put into the right queue, say one of B and C, we are still fine. However, suppose z came into A after B and C were formed. This is very much possible since we form the queues first and then swap them. We do not know when no one else will try to work on A and we cannot wait since we want to be wait-free. So why not attach the queues first and then populate them? Well, we would lose out on FIFO order as newly added nodes like z may enter the new queue before I bring y from A to it, say. If such an overtake is possible, we are again not wait-free.

The "solution" to the problem which we came up with was the following. It is okay if z gets into the queue late. Once we form the queues B and C, we will re-read A once more and take the newly found elements and put them into B and C accordingly. This again maintains FIFO order to make sure we keep our promise. And as I write this out, I am unable to control my laughter as our solution is only exceeded by our stupidity. It is very clear that this does not work and any competent reader would see that, even over a cup of tea! Say I do read A again and still find it empty. Does it mean that z will not enter A just after I assume it is empty and before I swap queues? Obviously not. So how do we know when no-one will touch A again? We cannot wait and hence cannot know. This immediately brings to mind one thought. If lock-free was our goal, this is trivial. Wait-freedom is truly much harder.

A try at a solution would be to set flags on nodes that someone is actually going to try to insert into their respective queues. This would however result in our worker threads waiting for a green signal from incoming user threads, which obviously defeats the purpose of our endeavour. The interesting aspect is that several ideas that spring out of our minds needs something of the following form to work: in this case, we need to check the emptiness of A and swap y's queue with one of B or C atomically to prevent further references to A. There is no easy way to do this!

The Solution

After realizing that splitting queues is not really an option, we found an alternate strategy that would in-fact work. Since this has been implemented, we will use snippets from our code to explain the solution.

We still wish to use FIFO ordering of requests to ensure wait-freedom. Thus, we still have a wait-free queue. However, now there is only one such queue, which we shall call the main queue. Also, there is a single thread that dequeues elements from this queue, which we shall call the Master thread. In short, a user thread inserts a request into the main queue. The Master thread is continuously attempting to dequeue requests from the main queue. When it obtains a request, it finds the node at which this value is to be inserted and hands over the value to the appropriate ElementInserter. The ElementInserter now performs the actual insertion of the value. Since values are not queued at an ElementInserter, there is no question of splitting a queue. This is explained in greater detail using our implementation.

First, we would like to introduce the Nodes present in the queue. Though we have used the term Node again, these nodes have a different set of fields, as shown below.

```java
public class Node {
	volatile int value; // Value held by the node
	volatile AtomicReference<Node> next; // Next node in the queue
	volatile int enqTid;
	volatile AtomicInteger deqTid;
	volatile boolean isInserted;  // Set once the node has been inserted into the BST
	                              // or it has been decided that an insert cannot be done
	volatile boolean insertSuccess; // The value is valid only once isInserted has been
	                             // set. Then, a value of true indicates that the key
	                             // has been inserted into the BST and a value of
	                             // false indicates that the insert was not performed due
	                             // to the presence of duplicates
```

The last two fields are necessary because we want the insert operation to be linearizable and we want to be able to inform the user whether the insertion succeded or failed. Then, the insert operation is very simple for a user thread -

```java
	/**
	 * Insert @param key into the BST
	 * @return true if the insert succeeded and false otherwise (failure
	 * because the key already exists in the BST)
	 */
	public boolean insert(int key) {
		// Indicate that a thread is using the BST and wait till the
		// BST is not being cleaned up
		int sum = 2;
		while(true) {
			while(numLiveThreads.get() % 2 != 0);
			if(numLiveThreads.addAndGet(2) % 2 == 0) {
				break;
			}
			sum += 2;
		}

		wfq.Node node = new wfq.Node(key, 0);
		Master.mainQueue.enq(node);  // Push the node into the main insert queue
```

```
        while(!node.isInserted());   // Wait till the node is actually inserted
                                     // or it is determined that it cannot be inserted

        numLiveThreads.addAndGet(-sum); // Indicate that this thread is done

        return node.isInsertSuccess();  // The flag will be set by the thread that
                                        // performs the insert or detects a duplicate
}
```

Deferring the explanation of *numLiveThreads* as we have been doing so far, we can see that insertion just involves creating a queue node, inserting it into the main queue and waiting till a signal that the insert is done is received. If we did not care about insert being linearizable, we could simply return after enqueueing the node. The processing, done by a different thread, will eventually result in the value getting added to the tree. However, as we could not think of a situation where non-linearizable inserts are useful, we decided to wait until it is known that the value has been inserted so that the insert method "takes effect" between its invocation and rsponse, as desired by linearizability. By the time the user thread breaks out of the spin, the insert will be complete.

Now, let us trace what happens to a node once it has been enqueued. The Master thread is running in an infinite loop, attempting to dequeue value from the main queue. When a node gets dequeued, the following code gets executed -

```
wfq.Node deqNode = mainQueue.deq();

Node tempNode = bst.root;
Node parentNode = null;

// Perform a standard BST search for the parent to which the insert
while(true) {
        if(tempNode == null) {
                if(parentNode.getElementInserter().nodeToInsert != null) {
                        // ElementInserter of parent is busy. Wait till it is free
                        while(parentNode.getElementInserter().nodeToInsert.isMarked());
                }

                // When the ElementInserter becomes free, the value it just inserted
                // may have become the true parent of the node to be inserted
                // Check for this case
                if(deqNode.getValue() > parentNode.getValue()) {
                        if(parentNode.getRight() != null) {
                                parentNode = parentNode.getRight();
                        }
                } else if(deqNode.getValue() < parentNode.getValue()) {
                        if(parentNode.getLeft() != null) {
                                parentNode = parentNode.getLeft();
                        }
                } else {
                        // The value the ElementInserter was inserting is the value
                        // that the Master is holding now. Singal that the insert
                        // cannot be done to avoid a duplicate
                        deqNode.setInsertSuccess(false);
                        deqNode.setIsInserted(true);
                        break;
                }

                // Hand over the node to ElementInserter for insert
                parentNode.getElementInserter().nodeToInsert.compareAndSet(null,
                      deqNode, false, true);
                break;
        }
```

```
            // Control reaches here only if parent has not yet been found
            // Continue BST search for it
            parentNode = tempNode;
            if(tempNode.getValue() < deqNode.getValue()) {
                    tempNode = tempNode.getRight();
                    continue;
            }
            if(tempNode.getValue() > deqNode.getValue()) {
                    tempNode = tempNode.getLeft();
                    continue;
            }

            // Found a marked node with the same key
            // Attempt to unmark the node to insert the key
            if(tempNode.isMarked().get()) {
                    deqNode.setInsertSuccess(tempNode.CASMarked(true, false));
                    if(deqNode.isInsertSuccess()) {
                            TreeSize.liveNodes.getAndIncrement();
                            TreeSize.deadNodes.getAndDecrement();
                    }
                    deqNode.setIsInserted(true);
                    break;
            }

            // Found an unmarked node with the same key
            // Singal that insert cannot be done as it would result in a duplicate
            deqNode.setInsertSuccess(false);
            deqNode.setIsInserted(true);
            break;
    }
```

The documentation in the code more or less explains what is going on. A standard BST search is done to find the future parent of the node. During this process if a marked node was found with the value to be inserted, it can simply be unmarked. Also, in an unmarked node is found with this value, the Master signals that the insert has failed.

If the Master finds a suitable parent, it waits for the mark on the *nodeToInsert* field of the required ElementInserter to become false. This mark is true when the ElementInserter is working on a node and becomes false when it is free. When the ElementInserter becomes free, the Master has to do one additional check to see whether the value the ElementInserter was working on should be the parent of the node to be inserted, in which case, it shifts to that ElementInserter. A single check is sufficient because an ElementInserter can be working only on one value at a time and no thread other than the Master gives it work.

The ElementInserter is waiting for the Master to provide it with work as follows

```
    boolean[] mark = new boolean[1];
    wfq.Node head;
    while((head = nodeToInsert.get(mark)) == null);
                    // Wait till a node has to be inserted
```

The ElementInserter is freed from this loop when the variable *head* is not null. At this point, *head* will contain the node with teh value to be inserted. The first check to be done is whether the value to be inserted is the same as that of the parent, in which case, the insert should fail. This check could have been done in the Master but is preferred to be done here as it reduces the bottleneck at the Master. The Master checks some cases of duplicates but not all, hence the need for this last validation

```
        if (head.getValue() == parentNode.getValue()) {
                // Node to be inserted has same value as potential parent
                // Signal that insert failed as it would result in a duplicate
                head.setInsertSuccess(false);
                head.setIsInserted(true);
                nodeToInsert.set(null, false); // Signal that a new node can be given for
                      insert
                continue;
        }
```

The remaining functionality depends on whether the parent of the node already has a child or not. If teh parent already has a child, the current ElementInserter can be passed on to the new child being created, as ElementInserters are only required at nodes where inserts can happen. Then, the code is as follows -

```
        if(parentNode.getLeft() != null || parentNode.getRight() != null) {
                // Parent node has one child already
                Node newNode = new Node(head.getValue());
                                // Create a new node for the value to be inserted
                newNode.setElementInserter(this);   // Pass on the ElementInserter to child
                Node oldParent = parentNode;
                parentNode = newNode; // Inform the ElementInserter that it has been
                                        // passed on to the child

                // Link the child to the parent
                if(oldParent.getValue() > newNode.getValue()) {
                        oldParent.setLeft(newNode);
                } else {
                        oldParent.setRight(newNode);
                }

                // Update statistics of the BST
                TreeSize.liveNodes.getAndIncrement();
                TreeSize.numNodes.getAndIncrement();

                // Indicate that insert was successful
                head.setInsertSuccess(true);
                head.setIsInserted(true);

                // Indicate that the ElementInserter can be given a new node to insert
                nodeToInsert.set(null, false);
                continue;
        }
```

This code has a few subtleties that are of importance. First, the reader may be confused as to why the ElementInserter receives a Node and creates a Node with the same value. In fact, the node it receives is a Node of the queue and the node it creates is a Node of the BST, to both of which we have unfortunately assigned the same name. Second, the new node being created should be assigned an ElementInserter before it is linked to the parent because once it has been linked to the parent, it is possible that the Master will attempt to access the ElementInserter of the newly created node because it may be the parent of the next insert. Third, the linking of the parent to the child is the linearization point for the insert operation because before this, the node cannot be discovered in the tree by search, delet or another insert and after this, it is discoverable. Lastly, since the same ElementInserter is being passed on to the child, no value will be given to the child for insert until this insertion is complete, that is, the *nodeToInsert* of this ElementInserter is set to null.

Suppose the parent had no children. Then, the work of the ElementInserter is a little different. In this case, it has to spawn a new ElementInserter for the child node. This can be seen in the code -

```
// Parent node does not have any children
// Create node to be inserted
Node newNode = new Node(head.getValue());

// Child node requires a new ElementInserter. Create it
ElementInserterThread newElementInserterThread = new ElementInserterThread(newNode);
newNode.setElementInserter(newElementInserterThread);
ThreadPool.execute(newElementInserterThread);

// Connect parent to child
if(parentNode.getValue() > newNode.getValue()) {
        parentNode.setLeft(newNode);
} else {
        parentNode.setRight(newNode);
}

// Update statistics of BST
TreeSize.liveNodes.getAndIncrement();
TreeSize.numNodes.getAndIncrement();

// Signal success of insert
head.setInsertSuccess(true);
head.setIsInserted(true);

// Indicate that the ElementInserter can be given a new node to insert
nodeToInsert.set(null, false);
```

As in the previous case, the ElementInserter for the child has to be created before it is linked to the parent and the linearization point of the insert operation is when the parent is actually linked to the child.

A simple examination can show that these cover all possible cases for the insert operation. We can see that once the ElementInserter has been assigned a value, it performs a finite number of steps before it is ready to take another value. Thus, in the Master, the wait for an ElementInserter to become free must occur in a finite number of steps as only the Master assigns it nodes to insert. Looking back at the code for the Master, we can see that this implies that the once the Master has dequeued a node, the *isInserted* flag is set in a finite number of steps. The FIFO ordering in the queue ensures that once a user thread has enqueued a node, in time at most proportional to the number of threads currently performing inserts on the BST, the node will get dequeued, and a finite number of steps after that, the *isInserted* flag will be set, allowing the user thread to return. Thus, the insert method is wait-free and bounded by $\mathcal{O}(n)f(k)$ where n is the number of threads and k is the number of nodes in the tree. The function f will be clarified shortly.

An alternate solution

There is another possible solution to the queue splitting problem, whcih we have not yet implemented. However, we wish to present it here. We have already reduced the difficulty in the problem in some sense: In reference to the problem in our earlier solution, a worker had to realize that no-one would touch the parent node's queue before it assumed the "queue-splitting" process was complete. Here no-one refers to any one of the N possible user threads ravaging the system. However, now, we only have one master thread who is accessing the structure and the same communication issue exists with N user threads replaced by one master. This is much easier, right? Let's have another look at it.

The possible problem was due to that fact that we wanted the abstraction map of the system to reflect the fact that A, from our previous exposition, would no longer exist as a valid queue, and B and C are two newly added ones. This solution aims at doing exactly this. The master has a certain perspective of the system - the various queues present in it currently. It wishes to update its view every time it starts a new insert operation. So, if we assume there is a list of such queues, the master has a read of it before beginning any insert. In this process it notifies itself of the newly added queues in the system. Suppose each queue had an associated flag exactly for this purpose, call it `masterSawIt`. So when the master reads the list of queues, it sets this flag in all of them to be true. This is a check for itself as well as for the other worker threads who are waiting for a response from the master. The master in the process of scanning the list of queues knows which node will become the parent of the new node to be inserted and puts the node into the appropriate queue.

What are the workers doing. Their job is again to read their queues, pull out the head node and then split the queue. How does the splitting work here. We have created two new queues. If we know that subsequent inserts will happen in them and not in the old one, we are done. We can re-read the old one once and be sure that newer inserts will end up in one of the newly created queues. From the old example, we would create new queues B and C and put them into the list of queues with their `masterSawIt` bit as false and wait till they become true. We can give some wait-free gaurentees here by placing suitable restrictions on the size of the tree at any point of time which can be governed by our choice of when to cleanup the structure. The consequence of master scanning the entire list of queues once before every insert is that this solution would not scale but in cases of small sized trees with high levels of contention, it is probably a good choice.

Let us complete the logistics of the solution. Once the workers know that the master has updated its view, it must split up the queue. But it cannot allow new guys from the master into the new queues until the split is complete as otherwise there would be overtaking. So, we need one bit which is toggled by the workers. Every queue has another flag `workerIsReady` which the worker will set to true only after the queue split is done in the case of an ongoing insert, but will be true if the node is currently idle. The master on the other hand spins until the target node is free and then pushes the new element into the queue. This is again wait-free as we are holding every insert wait-free and this wait would be for at most one insert. The other detail is that the list which holds the queues would also have to be wait-free so that all workers can add entries to it in a wait-free manner.

This solution clearly[1] works. The downside is the waiting, which although is bounded, may be large and hence other solutions like the one we have implemented would work better for larger cases. But as an academic exercise at least, this solution can catch a place somewhere, perhaps in a textbook! But it is clear that this solution is inherently more parallel than the one implemented by us, but would not scale. One should choose the solution and implementation based on the requirement. After all, r-bounded wait-freedom with $r \to \infty$ is just silly!

The Clean-up

The construction we provided provides wait-free insert, delete and search operations. What more could we want! There is the fact that our deletions are logical deletions, and some-time, it would be nice to physically delet the nodes in order to avoid traversing too many dead nodes. Since our operations are all wait-free, it seems reasonable to once in a while do a messy cleanup operation.

For this, we needed a mechanism to track the number of live and dead nodes in the tree, which explains why all our operations were updating those very statistics. We have a clean-up thread that is constantly monitoring these statistics, and when the fraction of dead nodes crossed some threshold (a tunable parameter), fires a clean-up operation.

Since clean-up would require considerable refactoring of the tree, we would like to lock the tree during clean-up, which is what makes our BST "near wait-free", rather than wait-free. For this "locking", we need to ensure that no new operation starts once the cleaner thread decides that a clean-up is necessary and after this indication has been given, the cleaner thread must wait for all pending operations to finish before starting a clean-up. To know how many pending operations are there, one possible idea was to have every user thread increment a counter when it starts an operation and decrement the counter when it ends the operation. Suppose a different field was used to lock the BST, we would have a problem because if a user thread saw that the BST was unlocked and then tried to increment its counter, if the cleaner thread checked the counter in-between the two operations, it would assume there is no user and proceed with clean-up. The user thread too would assume that no clean-up is going on, as it saw the BST unlocked and hence proceed, which would very likely result in incorrect execution. The counter cannot be incremented before a user thread knows that the BST is unlocked because then any thread that comes in would increment the counter and only those that have seen the BST locked would wait. Thus the counter would no longer be indicative of the numebr of pending user operations.

The solution to this problem is surprisingly simple. We still use an atomic integer, but we use the parity of the integer as an additonal bit of information to decide whether the BST is locked or unlocked. This atomic integer is

[1] Probably, one should be more careful with his words!

the slightly inappropriately named variable *numLiveThreads*. The semantics of *numLiveThreads* is as follows. If the value is even, the BST is unlocked and if it is odd, the BST is locked. Further, a value of 1 indicates that there are no pending user operations and a clean-up is in progress, a value of 0 indicates that there are no pending user operations and no clean-up is in progress and a any other value indicates the presence of pending user operations. As we saw earlier, ever user operation starts with -

```
        int sum = 2;
        while(true) {
                while(numLiveThreads.get() % 2 != 0);
                if(numLiveThreads.addAndGet(2) % 2 == 0) {
                        break;
                }
                sum += 2;
        }
```

As long as *numLiveThreads* is odd, the user thread must wait as the BST is locked. When it breaks out of that loop, it performs as it performs an *addAndGet* of 2, the value it obtains has the same parity as the previous value of *numLiveThreads*. If this value is still even, the operation may proceed. If clean-up is tuned to happen with high frequency, it is possible that a user thread adds to *numLiveThreads* more than once. Thus, the local variable sum is used to store the exact amount this thread has incremented *numLiveThreads* by. At the end of the operation, the thread nullifies the increase it caused by performing

```
        numLiveThreads.addAndGet(-sum);
```

Now consider the clean-up thread. When it decides that a clean-up has to be done, it simply does

```
        // Indicate that the tree is going to be cleaned up
        bst.numLiveThreads.getAndIncrement();
        // Wait till pending user threads have completed their operations
        while(bst.numLiveThreads.get() != 1);
```

Since the cleaner thread just increments *numLiveThreads*, the parity is changed. Any new operations entering now will see that *numLiveThreads* is odd and hence will wait. Any operation that completed its *addAndGet* earlier (the only other possibility as both operations are atomic) will proceed to completion and the clean-up thread will wait for it because such a thread must have left *numLiveThreads* at a value of at least 2, since no thread decrements *numLiveThreads* by a value different from what it increments it by. If the value of *numLiveThreads* becomes 1, it means that all oending user operations are done because a user thread only decrements the value it incremented. Thus the value of *numLiveThreads* can become 1 only when the value added by all threads except the cleaner thread have been removed. Thus, when the cleaner thread breaks out of this loop, it is safe to perform a clean-up.

The code for clean-up has not been presented here for the sake of brevity. Essentially, all ElementInserters are stopped. Then, a BFS traversal is done on the tree and all unmarked nodes are inserted using an AVL-tree insert into an initially empty tree to obtain a balanced binary search tree containing only the live nodes. Following this, ElementInserters are created for all nodes that have less than two children. Then, the lock on the tree is released by performing

```
// Indicate that clean-up is over and user threads may use the BST
bst.numLiveThreads.getAndDecrement();
```

Since this would set *numLiveThreads* to 0, all waiting user operations will now resume.

One point to note is that the frequency of clean-up affects two things. Whenever a clean-up occurs, all user operations are made to wait, albeit not for other user operations. However, this is what makes the data structure deviate form the ideal wait-free BST. On the other hand, a clean-up results in balancing of the BST, which would reduce teh time taken for future inserts (recall the function f that was used when bounding the number of steps an insert would take - it depends on how balanced the tree is as insert involves $\mathcal{O}(\log h)$ operations after dequeueing the node, where h is the height of the tree).

Testing

To begin with, the BST was tested with 2 threads, and set of 5 *insert* operations per thread. Only *inserts* needed to be checked then because the delete and search implementations were relatively straightforward in comparison. A history was obtained containing prints at each step of the *insert*, and the correctness of the algorithm was verified by identifying suitable linearization points for each of the inserts.

After ensuring that the insert operation was working fine, we included the other operations and included a check to ensure that the tree obtained is indeed a BST. We do a BFS (Breadth First Search) traversal on the tree, and check that the BST property is maintained.

Then, the *cleanup* was tested for small number of threads, by setting a small threshold for the dead node ratio, making sure that it was called.

After bug fixing, the actual testing was carried on a specified number of Threads, which use values from a specified range and call the different operations with specified probabilities. The performance of the algorithm was evaluated by varying these parameters. The parameters that were varied to test are :

- Number of threads
- Key range of inputs

- Percentage of different operations.

We tested it for 128, 256 and 512 threads, on a system with 4 hardware threads. We evaluated the following operation wise probability splits:

- 33 % search, 33 % insert and 33 % delete. (Given by 1 in the graph)
- 25 % search, 50 % insert and 25 % delete. (2)
- 9 % search, 90 % insert and 1 % delete. (3)

The results obtained are as shown below :

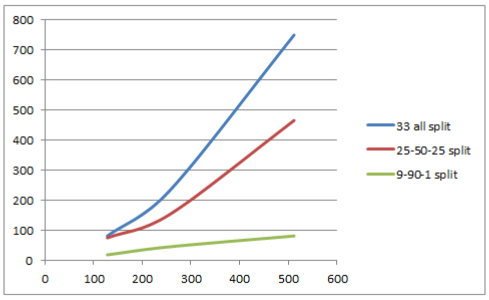

Figure 1: Time vs Number of Threads for a Key range of 512

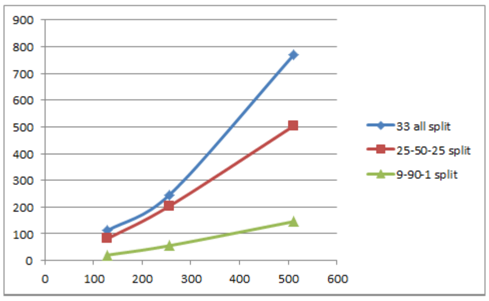

Figure 2: Time vs Number of Threads for a Key range of 2048

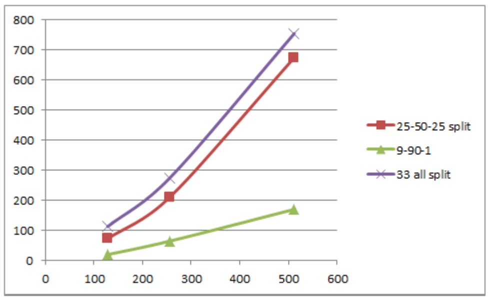

Figure 3: Time vs Number of Threads for a Key range of 8192

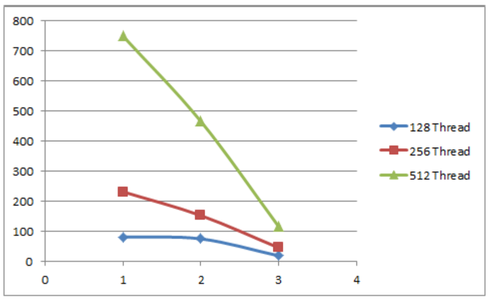

Figure 4: Time vs Operations percentages for a Key range of 512

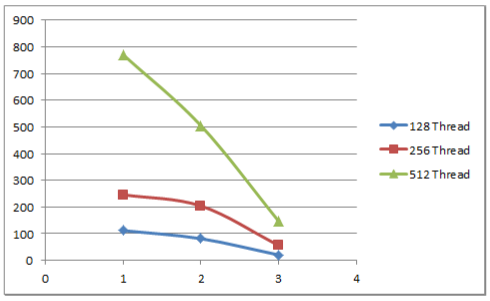

Figure 5: Time vs Operations percentages for a Key range of 2048

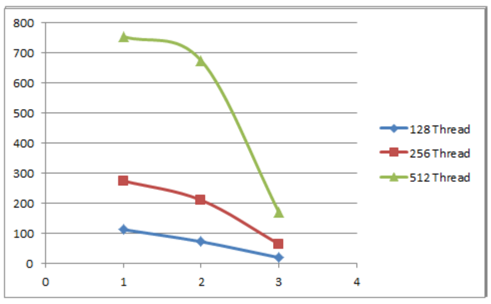

Figure 6: Time vs Operations percentages for a Key range of 8192

Here, we haven't plotted time vs key range, as the variation was found to be very less.

We can generally see that, on increasing the key range, the time taken is expected to be relatively higher, as the depth of the tree increases due to a larger number of successful *insert* inserts. Though on testing, it was seen that there wasn't too much of a difference in times, in case of very large key ranges.

On varying the percentage of operations, it can be seen that the *insert* operation is the bottle-neck and since the *search* is very simple, the time taken for a typical run, with large number of searches is less.

On increasing the number of threads, the time taken was observed to more, which is obvious because we were trying to run around 500 user threads (not considering element inserters) on a 4-core machine, which leads to heavy contention. On a system with larger of cores, the performance is expected to be much better.

Future Work

More exploration needs to be done with this idea, including

- Checking alternate solutions for insert, in terms of theoretical guarantess, scalability and feasibility in terms of implementation.
- Improving the efficiency of the clean-up operation.

Possibly there may also be mechanisms that result in more parallel, but scalable solutions for insert and clean-up.