Submitters: Adam & Shachaf

1 The Concurrent Binary Search Tree Algorithm

1.1 Abstract

This algorithm is an implementation of the set ADT. That includes:

– The *insert(x)* method adds x to the tree, returning *true* if and only if x wasn't already in the tree.
– The *remove(x)* method removes x to the tree, returning *true* if and only if x was in the tree.
– The *contains(x)* method returns *true* if and only if x was in the tree.

Due to a set being represented by such data structure like a binary search tree, there are certain mechanisms which require additional care in comparison to the *Lazy List*. Such additional care, is induced merely due to the fact that when *remove*-ing *binary nodes* from the tree (nodes who have both a left and a right child (and thus a left and a right subtree)) – we ought to replace such nodes with their successor in the tree, in order to maintain the *tree invariants*. This mechanism presents the following inconsistency:

Assume we have a tree with keys k_1, k_2, and that k_2 is the successor of k_1. Moreover, assume that the path between the node holding k_1 and the node holding k_2 is greater than one edge (i.e., there's at least one key in the tree which is in-between k_1 and k_2's path). Consequently, assume there is some operation named *op* pending which is traversing through the tree to find if k_2 exists in the tree. Moreover, assume that *op*'s *traversal* has yet to reach the node holding k_2 – and that it has already traversed through the node holding k_1 (i.e., the traversal is currently at a node holding a key in-between k_1 and k_2). Subsequently, assume another thread chose to run *remove*(k_1). We get that this thread will replace the node holding k_1 with the node holding k_2, since k_1 has a successor and it's k_2. Assume it has done so. Subsequently, assume *op's traversal* continues running, and that it finds that k_2 doesn't exist in the tree (since the node holding k_2 has moved to the location of the node that held k_1, and since the *traversal* has already passed the location of the node that held k_1). We therefore get a violation, since k_2 is present in the tree, and since no other thread is trying to remove it.

This violation suggests that the naïve way of *traversing* the tree is incorrect for such an algorithm – since a *traversal* can become invalid while it is running. Therefore, we suggest the new following approach to *traversing* the tree – which must end as valid *traversal*.

1.1.1 New Tree Traversal

The following approach recognizes that throughout the *traversal* of some operation, the key that it searches for might have moved to a location higher than where the *traversal* currently resides at. It recognizes so, by rerunning the *traversal* and checking for a repeating result (i.e., the same parent). Only after a repeating result should the *traversal* terminate. Notice: we rerun the *traversal* if and only if the key was not found.

```
1  public NodePair traverse(int key) {
2    NodePair first = new NodePair(null, null, false);
3
4    while (true) { // break if two consecutive traversals coalesce or if the key is found
5      Node second = findKeyOnce(key);
6      if ( (second.parent == first.parent) or (second.key == key) ) {
7        return second;
8      }
9        first = second;
10   }
```

Fig. 1: Pseudo-code for the traversal logic. We can see that the traversal terminates if and only if two consecutive traversals yield the same result (assuming both found the key to not exist in the tree), or if the key was found in some traversal in the while loop.

1.2 The *remove(x)* method

1.2.1 Implementation

When removing an element, we first traverse the tree to find the node to remove and its parent node. If the node was found, it and its parent should be locked and validated (a similar validation to the *Lazy List*'s), and then there are three cases: the node can be either binary, unary or a leaf.

- If the node is a leaf, the removal is trivial – just disconnect the node's parent from the node.
- If the node is unary, then it's still simple enough – just connect the parent to the single child of the node being removed.
- The third case is more complicated – as the "hole" created by the removed node must be filled with its successor. This "filling" must be done without ever disconnecting any node or subtree from the tree, otherwise concurrent calls to *contains* or *insert* might fail. This is done by first finding the node's successor – the leftmost child in the subtree rooted in the removed node's right child. The successor and its parent must be locked, too, to prevent them from changing. Then, again, there are two cases: either the successor is a leaf, or it has a right child.
    - If the successor is a leaf, it can be connected to the removed node's children, then connected to the removed node's parent, and finally disconnected from its original parent. Note that the order is important – any other order temporary unlinks some of the nodes from the tree, harming linearizability.
    - Otherwise, the removal must be more complicated, as demonstrated by **Fig. 2.** See **Fig. 3** for the technique used to allow this type of removal.

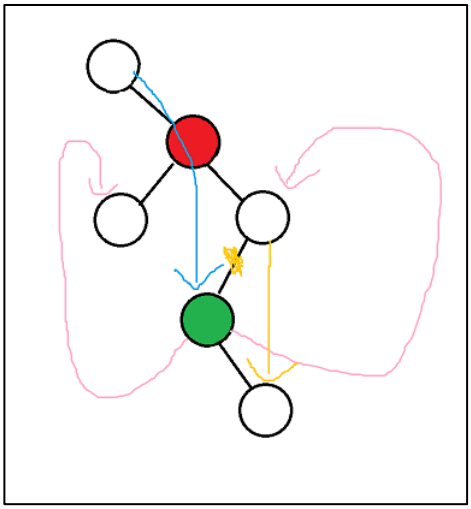

Fig. 2: The red node is being removed, and the green node is its successor. No order of setting up the links is correct, as any order temporarily unlinks some nodes.

The pink, orange and blue links should be created for the removal, but:

- Starting with the pink links disconnects the successor's child
- Starting with the orange link disconnects the successor and its children
- Starting with the blue link disconnects the node to remove and its subtree

Any order of modifying the links temporarily unlinks some nodes from the tree, and the removal can't be achieved. Hence, a more complicated removal function is required when the successor has a right child: First, the successor is turned into a leaf by finding its own successor and modifying the tree so that it becomes its left child. Then, once the successor becomes a leaf, it can be removed as previously explained.

Note that similarly to the removal in the *Lazy List*, the removed node is marked before its links are being modified, to prevent other functions from misusing bad nodes.

1.2.2 Linearization Point

- For a successful removal, the linearization point is when the node is marked as removed. This happens before the physical removal, but after all the relevant nodes (successor and maybe successor of successor, too) are found.
- For an unsuccessful call, the linearization point is when the parent of the place where the node should be is validated and found to be null.

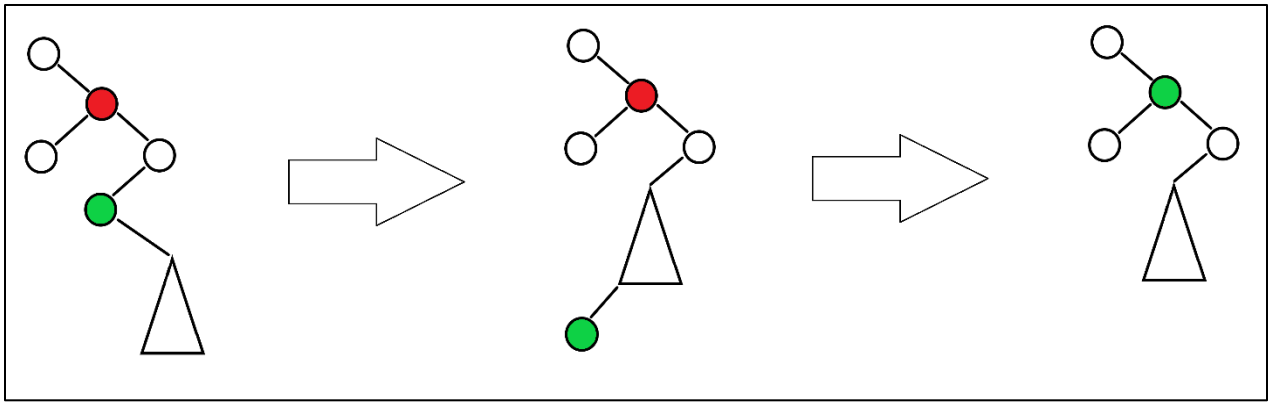

Fig. 3: The red node is being removed, and the green node is its successor. No order of setting up the links is correct, as any order temporarily unlinks some nodes, so we transform the successor into a leaf, and only then apply the removal techniques as before.

1.3 The *insert(x)* method

1.3.1 Implementation

The implementation of *insert(x)* is similar to its implementation in the *Lazy List* algorithm. We first traverse the tree to find the location of the node of where *x* would have been inserted. Notice: this is a regular binary search tree; therefore, all new nodes are inserted as leaves. Then, we grab locks over *pred* – the parent of the location of the node of where *x* should have been inserted, as well as a lock over *curr* – the location of the node of where *x* should have been inserted. After grabbing the aforesaid locks, we validate that the state of the tree among those two alleged nodes has been maintained (i.e., checking *curr* is still the child of *pred*), as well as assure that *pred* as well as *curr* aren't logically removed (i.e., *marked*).

If such validation/assurance fails – we simply rerun the logic of this method (similarly to the *Lazy List*).

If the validation/assurance succeeds, we continue on with similar mechanisms to the *Lazy List* insertion – that includes: checking if the location of the node of where *x* would have been inserted is an already existing node. In that case, we return **false**, since the traversal would have skipped that node in case it wouldn't have held *x*. The other case, is that the location of the node of where *x* would have resided isn't an already existing node. In such case, we simply create a new Node holding *x*, and set it as a left/right child correspondingly to the parent of the location of the node of where *x* would have resided. As stated above, the inserted node of *x* will always initially be a leaf.

1.3.2 Linearization Point

- For a successful *insert(x)* call (i.e., *true* being returned), we linearize the *insert(x)* method at the point of setting the left/right child of the relevant node to be a new node that holds *x*.
- For an unsuccessful *insert(x)* call (i.e., *false* being returned), we linearize the *insert(x)* method at the point of comparing *x* and the key of the node of where *x* would have resided (such comparison would result in them being equal – and therefore no insertion is needed).

1.4 The *contains(x)* method

1.4.1 Implementation

The implementation of *contains* is as simple as traversing the tree as described in the section describing the new tree traversal. After getting the traversal result, the function reports that the key exists if and only if it is found and unmarked.

Just like in the *Lazy List*, the function doesn't hold any locks – and the linearizability is enforced by the implementation of the other functions.

1.4.2 Linearization Point

- For a successful call, the linearization point is when getting a positive traversal result and checking that the resulting node is not marked.
- For an unsuccessful call, we linearize *contains(x)* within its execution interval at the earliest of the following points: (1) the point of observing a marked node containing the key and (2) a negative traversal result (i.e., no key holding *x*) and (3) the point immediately before a new node holding *x* is added to the tree.

1.5 Deadlock-Prevention

1.5.1 Lock Ordering Over Branches

We have ordered the sequence of lock acquisition in such way that locks of nodes in the same branch in the tree, are acquired in a total order. The **branch-specific total order** of node-lock acquisition, is defined by the height of the nodes in the tree. An operation must acquire higher nodes' locks in some branch in the tree, before acquiring lower nodes' locks in the same branch. Such **branch-specific total order** of node-lock acquisition, avoids a *resource-deadlock* in our algorithm, since no operation of our algorithm tries to acquire locks over two nodes from two different branches. Therefore, we are granted a *deadlock-free* algorithm.

1.5.2 Enforcing Lock Ordering Over Branches

Enforcing node-lock ordering over same branches in the tree would be done in the naïve way of manually acquiring node-locks in the same branch in a descending manner. Albeit, if there's a possibility of nodes on the same branch being reordered throughout the process of acquiring their locks, it would result in our **branch-specific total order** of node-lock acquisition being breached, which would result in our algorithm not being *deadlock-free*.

Unfortunately, such reordering of nodes on the same branch is possible in our case, due to the *remove(x)* mechanism which might replace a *"to-be-removed"* node with its successor – which could change their order in the branch (assuming the successor resided lower in height in the branch). Therefore, throughout the algorithm, after most node-lock acquisitions (we will soon explain what the exception cases are), we validate that the following "*to-be-acquired*" node-lock is correspondent with our **branch-specific total order** of node-lock acquisition (i.e., we validate that it's lower in height in the branch than the last acquired node-lock). Notice, in cases that we acquire a node-lock over a parent and then acquire a node-lock over its child, the lock order validation is done using the *child* fields of the parent.

We have mentioned that we don't validate the **branch-specific total order** of node-lock acquisition after every node-lock acquisition. These cases are very specific to our algorithm, and therefore to understand this explanation we will point to different places in our code.

The first case is when we call *removeBinaryNode()* from within a *remove(n_1)* call – which is done in case we want to remove a node that has two children. In such case, the last acquired node-lock is that of the n_1, while the *"to-be-acquired"* node-lock is that of the parent of the successor of n_1 (since we want to replace n_1 with its successor) – we'll call the successor *succ* and its parent *succ_pred*. Moreover, n_1*'s* right subtree isn't empty, which forces the fact that its successor is in its right subtree. Therefore, *succ_pred* isn't higher in height than n_1. Now, in order for *succ_pred* to move higher than n_1 in their branch and cause a reordering, it would take a different binary node n_2 higher than n_1, to find *succ_pred* as its successor (since in any other case *succ_pred* won't move higher than n_1). Similarly, to n_1, *succ_pred* is in n_2's right subtree, which forces that $n_2.key < succ.key < succ\_pred.key$ – and therefore *succ_pred* isn't n_2's successor, contradiction – therefore, we don't need to validate the **branch-specific total order** of node-lock acquisition.

The second case is similar to the last case. Using the same terms from the last case, it's when we call *removeWithNonLeafSuccessor()* from within a *removeBinaryNode()* call – which is done when we want to replace n_1 with *succ* but *succ* isn't a leaf. In such case, the last acquired node-lock is that of *succ*, while the *"to-be-acquired"* node-lock is that of the parent of the successor of *succ*. Now, *succ's* left subtree must always be empty (else n_1's successor won't be *succ*), therefore we get that *succ's* right subtree isn't empty, since *succ* isn't a leaf. From here, the same proof applies from the last case – we just treat *succ* as the previous case's n_1. Therefore, we don't need to validate the **branch-specific total order** of node-lock acquisition.