Submitters: Adam & Shachaf

**1 The Concurrent Binary Search Tree Algorithm**

**1.1 Abstract**

This algorithm is constituted by the standard ADT of a set. That includes:

– The *add(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 are 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 , , and that is the successor of . Moreover, assume that the path between the node holding and the node holding is greater than one edge (i.e., there’s at least one key in the tree which is in-between and ’s path). Consequently, assume there is some operation named pending which is traversing through the tree to find if exists in the tree. Moreover, assume that *op*’s *traversal* has yet to reach the node holding – and that it has already traversed through the node holding (i.e., the traversal is currently at a node holding a key in-between and ). Subsequently, assume another thread chose to run *remove*(). We get that this thread will replace the node holding with the node holding , since has a successor and it’s Assume it has done so. Subsequently, assume *op’s traversal* continues running, and that it finds that doesn’t exist in the tree (since the node holding has moved to the location of the node that held , and since the *traversal* has already passed the location of the node that held ). We therefore get a violation, since 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. **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). Notice that we rerun this *traversal* only in case of the key not being found.

// TODO: add pseudo-code?

**1.1** **The** *add(x)* **method**

**1.2** **The** *remove(x)* **method**