1 Algorithm Description

1.1 Introduction

A Binary Search Tree (BST) implements the dictionary abstract data type. In a BST, all the values in a node's left subtree are less than the node value and all the values in the node's right subtree are greater than the node value. Duplicates are not allowed. A BST supports three main operations, viz.: search, insert and delete. Search(x) determines if x is present in the tree. Insert(x) adds key x to the tree if it is not already present. Delete(x) removes the key x from the tree if it is present.

1.2 Related Work

Several algorithms have been proposed for non-blocking binary search trees. Ellen[1] and Aravind[2] have proposed a lock-free external BST. An external BST has keys only in the leaf nodes and the internal nodes serve as routing nodes. Howley[3] has proposed a lock-free internal BST. As the tree grows in size, search time dominates and the internal BST tend to perform better than external BSTs as the average path length of an internal BST is half that of an external BST. On the other hand the algorithms described by Ellen[1] and Howley[2] operate on the node level while the algorithm described by Aravind[2] operates on edge level which tend to provide more concurrency when there is high contention. So the algorithm described by Aravind[2] performs well for smaller trees with high contention. We have designed our algorithm to get the benefits of both the worlds. Our algorithm like Howley's[3] is an internal BST and like Aravind's[2] operate on edge level.

1.3 Overview of the algorithm

Every operation in our algorithm begins with a seek phase. The operation traverses the search tree from the root node until it finds the target key or it reaches a leaf node. We refer to the path traversed by the operation in the seek phase as the access path. Seek returns references to the last two nodes in the access path. For con-venience, we refer to them as parent and node. For insert operation, seek also returns a reference to the injection point where the new node will be inserted. The operation then compares the target key with the value stored in node. Depending on the result of the comparison and the type of the operation, the operation either terminates or moves to the next phase. We now describe the next steps for each of the type of operation one-by-one.

For a *search* operation, if the two keys match, then the operation returns *true* (key was found); otherwise it returns *false* (key was not found).

For an *insert* operation, if the two keys match, then the operation returns *false* (key already exists in the tree); otherwise it moves to the execution phase (key does not exist in the tree). A new node is created and it contains the key being inserted. Finally the insert operation switches the pointer at the node that is pointing to the injection point to point to the new node.

For a *delete* operation, if the two keys do not match, then the operation returns *false* (key does not exist in the tree); otherwise it moves to the execution phase (key does exist in the tree). Delete can be of two types, viz.: simple and complex. Deletion of a node with less than two children is a simple delete and deletion of a node with exactly two children is a complex delete.

In a simple delete, the pointer at the parent that is pointing to the node being deleted is switched to point to the non-null child of the node being deleted.

Complex delete has three steps: (i) identifying the successor node, which is the smallest node in the right subtree. (ii) copying the successor's key to the node being deleted and then removal of the successor by a simple delete and (iii) creating a fresh copy of the node and installing it by switching the pointer from the parent to the node being deleted to point to the fresh copy of the node.

1.4 Details of the Algorithm

A tree node in our algorithm consists of three fields: (i) *markAndKey* which contains the key stored in the node, (ii) *child* array, which con-

tains the addresses of the left and right children and (iii) *readyToReplace*, which is a boolean flag used by complex delete operation to indicate if a node can be replaced with a fresh copy of it. This algorithm like the algorithm described by Aravind[2], operates on edge level. A delete operation obtains ownership of the edges it needs to work on by marking them. To enable marking we steal two bits from the child addresses of a node,(i) *deleteFlag*, to denote if the node is undergoing deletion and (ii) *promoteFlag*, to denote if the node is undergoing promotion. The difference between *deleteFlag* and *promoteFlag* is that *deleteFlag* will be set on the node undergoing deletion while *promoteFlag* will be set on its successor node. A node is said to be marked if any of the *deleteFlag* or *promoteFlag* is set on its child addresses. We also steal another bit (*nullFlag*) from the child address to denote if the corresponding child is null. This serves as a unique null pointer which prevents ABA problem. As complex delete replaces a key in a node being deleted, a flag is required to identify if the key in a node has changed. So we steal a bit from the key field and use it as a mark bit. If the mark bit is set, it denotes that the key in the node has changed.

We next describe the details of the seek phase, which is executed by all operations (search as well as modify) after which we describe the details of the execution phases of insert and delete operations.

1.5 The Seek Phase

The seek phase has two steps: traverse step and validation step. In the traverse step an operation traverses the tree along a simple path from the root node to a leaf node, which is referred to as the access path. At each internal node, if the target key is smaller than the node's key it follows the left pointer, if the target key is greater than the node's key it follows the right pointer. If the target key matches the node's key or a leaf node is reached, the traverse step is completed. Also, the seek keeps track of the last unmarked edge in the access path which will be used during helping(explained later).

In the validation step the access path is validated. This is required as the key in any node along the access path can change due to a complex delete. In a complex delete we always replace the key in a node undergoing deletion with the smallest key in the right subtree. Hence the value of the key in a node can only increase. This affects the range of keys that can be possibly contained in each of its subtrees. The range in the left subtree expands and the range in the right subtree shrinks. Expanding the range will not affect the seek as the key being searched will still be contained in the subtree. But shrinking the range may require the seek to restart. Like in Howley's[3] algorithm, we keep track of the last edge in which a right path was taken. Once the traverse step is completed, we validate if the key in the last right edge has changed. If yes, we restart the seek, else the seek terminates.

An additional validation is required in the seek phase to prevent the below scenario. Consider a tree shown in fig(i) which is being modified by two threads A and B. Assume that thread B starts with delete(x). Meanwhile thread A starts seek(z) and gets stalled at node Y after the right turn it took at node X. Now thread B completes delete(x) and the tree looks like in fig(ii). Now assume thread B does delete(w), which is a complex delete and the key z is promoted up the tree and the node Z is deleted by a simple delete operation. Now Thread A resumes search(z) at node Y and finds a *null* node at the left child of node Y. It then checks if the key at the last right turn it took at node X (which is no longer a part of the tree) has changed after it had read it. Since it has not changed, search(z) terminates and can incorrectly report that z is not part of the tree.

To avoid this scenario seek keeps track of two seek records, viz.: current seek record and previous seek record. During the validation step, we first check if the key has not changed. If the key has changed we restart the seek phase. If the key has not changed we check if the last right node is unmarked. If it is unmarked then the seek completes and returns the current seek record to the operation which invoked the seek. If the last right node is marked, then

seek returns the previous seek record if the last right turn node matches in both the previous and current seek record. Else the seek restarts. This ensures that the seek is linearized correctly.

(i) initial state   (ii) simple delete(x)   (iii) complex delete(w)
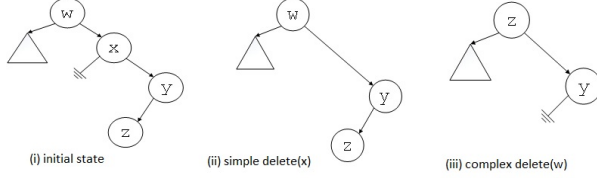
1.6 Execution Phase of an Insert Operation

For inserts the algorithms proposed by Ellen[1] and Howley[3] obtain ownership of a node and the algorithm proposed by Aravind[2] obtains ownership of an edge. Our algorithm does not obtain ownership of a node or an edge for an insert operation. Hence in our algorithm an insert operation will not block any other update (insert or delete) operation.

Let key denote the key to be inserted into the tree and R denote the seek record returned by the seek phase. Also let $nKey$ denote the key stored in $R{\rightarrow}node$. If key matches with $nKey$ then key is already present in the tree and insert returns *false*. If they do not match a node *newNode* is created and initialized with key *key*. Finally the insert operation tries to replace the edge $(R{\rightarrow}node, R{\rightarrow}injectionPoint)$ with the edge $(R{\rightarrow}node, R{\rightarrow}newNode)$ using a CAS instruction on the appropriate child field of $R{\rightarrow}node$. If the CAS instruction succeeds, then the insert opera-

tion has completed. Otherwise, the insert operation performs helping if needed and then retries by re-executing the seek phase.

To determine if it needs to perform helping, the insert operation reads the address stored in the child field of $R{\rightarrow}node$ on which the CAS failed. If the address is not marked then the injection point has changed due to a successful insert by another thread. So the insert operation of the current thread retries by re-executing the seek phase. If the address is marked then the insert operation has failed due to a concurrent delete operation. If the *deleteFlag* (or *promoteFlag*) is set then it implies that a concurrent delete operation is trying to remove (or promote) $R{\rightarrow}node$. In this case, the insert operation performs helping along the last unmarked edge $(R{\rightarrow}lastUParent, R{\rightarrow}lastUNode)$.

To summarize, the execution of an insert operation consists of an alternating sequence of seek and execution phases until the operation terminates. The pseudocode of the execution phase of an insert operation is given in lines 36-47. The steps for creating and installing a new node are given in lines 36-43. In case the CAS instruction fails, the steps for helping a delete operation are given in lines 45-47. Helping, if needed, is performed by invoking a deepHelp routine, which is given in lines 278-300.

1.7 Execution Phase of a Delete Operation

Delete operation begins with a seek