Efficient Lock-free Binary Search Trees*

Bapi Chatterjee, Nhan Nguyen and Philippas Tsigas

Department of Computer Science and Engineering
Chalmers University of Technology,
Gothenburg, Sweden
{bapic,nhann,tsigas}@chalmers.se

Abstract

In this paper we present a novel algorithm for concurrent lock-free internal binary search trees (BST) and implement a Set abstract data type (ADT) based on that. We show that in the presented lock-free BST algorithm the amortized step complexity of each set operation - ADD, REMOVE and CONTAINS - is $O(H(n) + c)$, where, $H(n)$ is the height of BST with n number of nodes and c is the contention during the execution. Our algorithm adapts to contention measures according to read-write load. If the situation is read-heavy, the operations avoid helping pending concurrent REMOVE operations during traversal, and, adapt to interval contention. However, for write-heavy situations we let an operation help pending REMOVE, even though it is not obstructed, and so adapt to tighter point contention. It uses single-word compare-and-swap (CAS) operations. We show that our algorithm has improved disjoint-access-parallelism compared to similar existing algorithms. We prove that the presented algorithm is linearizable. To the best of our knowledge this is the first algorithm for any concurrent tree data structure in which the modify operations are performed with an additive term of contention measure.

1 Introduction

With the wide and ever-growing availability of multi-core processors it is evermore compelling to design and develop more efficient concurrent data structures. The non-blocking concurrent data structures are more attractive than their blocking counterparts because of obvious reasons of being immune to deadlocks due to various fault-causing factors beyond the control of the data structure designer. Lock-free data structures provide guarantee of deadlock/livelock freedom with fault tolerance and are usually faster than wait-free ones which provide additional guarantee of starvation freedom at the cost of increased programming complexity.

In literature, there are lock-free as well as wait-free singly linked-lists [11, 24], lock-free doubly linked-list [23], lock-free hash-tables [17] and lock-free skip-lists [11, 22]. However, not many performance-efficient non-blocking concurrent search trees are available. A multi-word compare-and-swap (MCAS) based lock-free BST implementation was presented by Fraser in [12]. However, MCAS is not a native atomic primitive provided by available multi-core chips and is very costly to be implemented using single-word CAS. Bronson et al. proposed an optimistic lock-based partially-external BST with relaxed balance [4]. Ellen et al. presented lock-free external binary search tree [10] based on co-operative helping technique presented by Barnes [3]. Though their work did not include an analysis of complexity or any empirical evaluation of the algorithm, the contention window of update operations in the data structure is large. Also, because it is an external binary search tree, DELETE are simpler at the cost of extra memory to maintain internal nodes without actual values. Howley et al. presented a non-blocking internal binary search tree [15] based on similar technique. A software transactional memory based approach was presented by Crain et al. [7] to design a concurrent red-black tree. While this approach seems to outperform some coarse-grained locking methods, they are easily vanquished by a carefully tailored locking scheme as in [4]. Recently, two lock-free external BSTs [19, 5] and a lock-based internal BST [8] have been proposed. All of these works lack comprehensive theoretical complexity analysis.

In an internal BST, ADD operations start from the root and finish at a leaf node, where the new element is inserted. To REMOVE a node which has both the children present, its successor or predecessor is shifted to take

---

its place. A common predicament for the existing lock-free BST algorithms is that if multiple modify operations contend at a leaf node, and, if a Remove operation among them succeeds then all other operations have to restart from the root. It results in the complexity of a modify operation to be $O(cH(n))$ where $H(n)$ is the height of the BST on n nodes and c is the measure of contention. It may grow dramatically with the growth in the size of the tree and the contention. In addition to that, Contains operations have to be aware of ongoing Remove of nodes with both children present, otherwise, it may return invalid results, and, hence in the existing implementations of lock-free internal BST [15], they also may have to restart from the root on realizing that the return may be invalid if the other subtree is not scanned. The external or partially-external BSTs remain immune to the latter problem at a cost of extra memory for the routing internal nodes. Our algorithm solves both these problems elegantly. The Contains operations in our BST enjoy oblivion of any kind of modify operation as long as we do not put them to help a concurrent Remove, which may be needed only in write-heavy situations. Also, the modify operations after helping a concurrent modify operation restart not from the root rather from a level in the vicinity of failure. It ensures that all the operations in our algorithm run in $O(H(n) + c)$. This is our main contribution.

We always strive to exploit maximum possible disjoint-access-parallelism [16] in a concurrent data structure. The lock-free methods for BST [10, 15], in order to use single-word `CAS` for modifying the outgoing links atomically, and yet maintain correctness, store a flag as an operation field or some version indicator in the node itself, and hence a modify operation "holds" a node. This way of holding a node, specifically for a Remove, can reduce the progress of two operations which may remain non-conflicting if modifying two outgoing links concurrently. In [19], a flag is stored in a link instead of a node in an external BST. We chase this problem of just holding a link for an internal BST. We find that it is indeed possible that a Remove operation, instead of holding the node, just holds the links connected to and from a node in a determined order so that maximum possible progress of two concurrent operations, working at two disjoint memory words corresponding to two links can be ensured. We take the business of "storing a flag" to the link level from the node level which significantly improves the disjoint-access-parallelism. This is our next contribution.

Helping mechanism which ensures non-blocking progress may prove counterproductive to the performance if not used judiciously. However, at times, if the proportion of Remove operations increases, which may need help to finish their pending steps, it is better to help them, so that the traversal path does not contain large number of "under removal" nodes. Keeping that in view, we take helping to a level of adaptability to the read-write load: we provide choice over whether an operation, during its traversal, helps an ongoing Remove operation. We believe that this adaptive conservative helping in internal BSTs may be very useful in some situations. This is a useful contribution of this work.

Our algorithm requires only single-word atomic `CAS` primitives along with single-word atomic read and write which now exist in practically all the widely available multi-core processors in the market. Based on our design, we implement a Set ADT. We prove that our algorithm is linearizable [14]. We also present complexity analysis of our implementation which is lacking in existing lock-free BST algorithms. This is another contribution in this paper.

The body of our paper will further consist of the following sections. In section 2, we present the basic tree terminologies. In section 3, the proposed algorithm is described. Section 4 presents a discussion on the correctness and the progress of our concurrent implementation along with an amortized analysis of its step complexity. The paper is concluded in section 5.

2 Preliminaries

A *binary tree* is an ordered tree in which each *node* x has a *left child* and a *right child* denoted as $left(x)$ and $right(x)$ respectively, either or both of which may be *external*. When both the children are external the node is called a *leaf*, with one external child a *unary* node and with no external child a *binary* node, and, all these non-external nodes are called *internal* nodes. We denote the parent of a node x by $p(x)$ and there is a unique node called *root* s.t. $p(root) = null$. Each parent is connected with its children via pointers as links (we shall be often using the term pointer and link interchangeably when the context will be understood). We are primarily interested in implementing an ordered *Set* ADT - *binary search tree* using a binary tree in which each node is associated with a unique key k selected from a totally ordered universe. A node with a key k is denoted as $x(k)$ and x if the context is otherwise understood. Determined by the total order of the keys, each node x has a *predecessor* and a *successor*, denoted as $pre(x)$ and $suc(x)$, respectively. We denote height of x by $ht(x)$, which is defined as the distance of the deepest leaf in the subtree rooted at x from x.

We focus on *internal BSTs*, in which, all the internal nodes are *data-nodes* and the external nodes are usually denoted by null. There is a *symmetric order* of arranging the data - all the nodes in the *left subtree* of $x(k)$ have keys less than k and all the nodes in its *right subtree* have keys greater than k, and so no two nodes can have the same key. To query if the BST Contains a data with key k, at every *search-step* we utilize this order to look for the desired node either in the left or in the right subtree of the *current node* if not matched at it, unless we reach an external
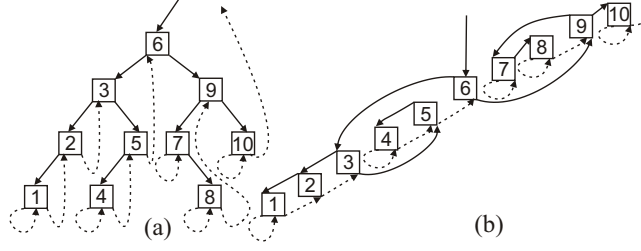
Figure 1: (a) Threaded BST (b) Equivalent ordered list.

node. On reaching an external node we return false, else, if the key matches at a node then we return true or address of the node if needed. To ADD data, we query by its key k. On reaching an external node we replace this node with a new leaf node $x(k)$. To REMOVE a data-node corresponding to key k we check whether $x(k)$ is in the BST. If the BST does not contain $x(k)$, false is returned. On finding a node with key k, we perform delete as following. If it is a leaf then we just replace it with an external node. In case of a unary node its only child is connected to its parent. For a binary node x, it is first replaced with $pre(x)$ or $suc(x)$, which may happen to be a leaf or a unary node, and then the replacer is removed.

In an alternate form - an *external BST*, all the internal nodes are *routing-nodes* and the external nodes are data-nodes. In this paper we focus on internal BSTs, and hence forward, by a BST we shall mean an internal BST.

3 Our Algorithm

3.1 The Efficient Lock-free BST Design

To implement a lock-free BST, we represent it in a threaded format [21]. In this format, if the left or the right child pointers at x is null and so corresponds to an external node, it is instead connected to $pre(x)$ or $suc(x)$, respectively. Some indicator is stored to indicate whether a child-link is used for such a connection. This is called *threading* of the child-links. In our design, we use the null child pointers at the leaf and unary nodes as following - right child pointer, if null, is threaded and is used to point to the successor node, whereas, a similar left child pointer is threaded to point to the node itself, see Fig. 1(a). In this representation a binary tree can be viewed as an ordered list with exactly two outgoing and two incoming pointers per node, as shown in Fig. 1(b). Also, among two incoming pointers, exactly one is threaded and the other is not. Further, if $x(k_i)$ and $x(k_j)$ are two nodes in the BST and there is no node $x(k)$ such that $k_i \leq k \leq k_j$ then the interval $[k_i, \ k_j]$ is called *associated* with the threaded link incoming at k_j.

We exploit this symmetry of the equal number of incoming and outgoing pointers. A usual traversal in the BST following its symmetric order for a predecessor query, is equivalent to a traversal over a *subsequence* of the ordered-list produced by an in-order traversal of the BST, which is exactly the one shown in Fig. 1(b). This is made possible by the threaded right-links at leaf or unary nodes. Though in this representation, there are two pointers in both incoming and outgoing directions at each node, a single pointer needs to be modified to ADD a node in the list. To REMOVE a node we may have to modify up to four pointers. Therefore, ADD can be as simple as that in a lock-free single linked-list [11], and REMOVE is no more complex than that in a lock-free double linked-list [23]. A traversal in a lock-free list may enjoy oblivion from a concurrent REMOVE of a node. Also in our design of internal BST, a traversal can remain undeterred by any ongoing modification, unlike that in existing lock-free implementations of internal BSTs [15, 12].

In all the existing designs of lock-free BSTs, when an operation fails at a link connecting to an external node because of a concurrent modify operation, it retries from the scratch i.e. it restarts the operation from the root of the tree, after helping the obstructing concurrent operation. We aim to avoid the "retry from scratch" behavior and rather restart from a node at the vicinity of the link where the operation fails, after the required helping. To achieve that, we need to get hold of the appropriate node(s) to restart the failed operations. We use a *backlink* per node which is guaranteed to point to a node *present* in the tree from where the failure spot is always a single link away. It should be noted that a backlink is not used for a tree traversal, see Fig. 2(a).

In designing an internal BST in a concurrent setup, the most difficult part is to perform an error-free REMOVE of a binary node. To remove a binary node we replace it with its predecessor, and hence, the incoming and outgoing links of the predecessor also need to be modified in addition to the incoming and outgoing links of the node itself. According to the number of links needed to be modified in order to remove a node, unlike traditional categorization of nodes of a BST into leaf, unary and binary, we categorize them into three categories as shown in Fig. 2(b). The categorization characteristic is the origin of the threaded incoming link into the node, hereafter we call this link the *order-link*. Nodes belonging to category 1 are those whose order-link emanates from themselves; to category 2, it
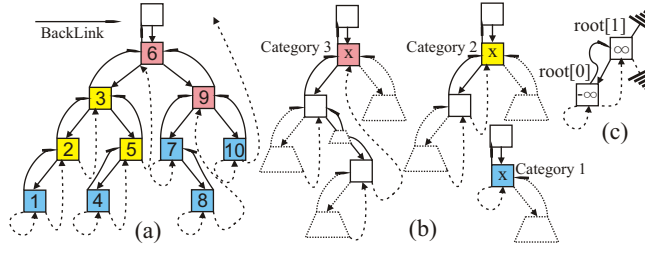
Figure 2: (a) Threaded BST with backlinks (b) Categorization of nodes for REMOVE (c) An empty tree.
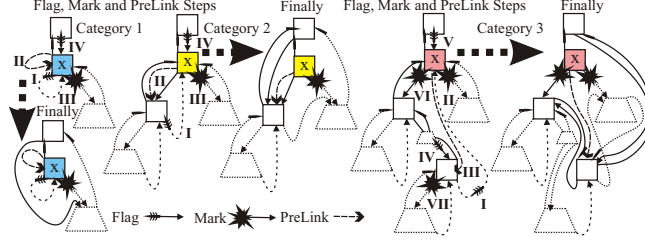

Figure 3: Remove steps of nodes

emanates from the left-child of the node, and to category 3 are those whose incoming order-link emanates from a "distant" node in its left-subtree. We name the node where the order-link emanates from, an *order-node*.

To remove a node of category 1, only the incoming parent-link needs to be modified to connect to the node pointed by the right-link. For a category 2, the parent-link is updated to connect to the node pointed by the left-link and the order-link is modified to point to the node which the right-link was pointing to. In order to remove a category 3 node, its predecessor replaces it and the incoming and outgoing links of the predecessor are updated to take the values of that of the removed node. Parent-link of the predecessor is connected to the node which its left-link was pointing to before it got shifted. Also, when a link is updated the thread indicator value of the link, which it updates to, is copied to it. Please note that in this categorization, a unary or a binary node whose left child is left-unary (i.e. whose right child is null) or a leaf, gets classified in to category 2. Whereas, category 1 includes conventional leaf and right-unary nodes (whose left child is null).

Having categorized the nodes as above we describe the modification of links associated with a node undergoing REMOVE. Storing an indicator bit in a pointer has been used in many previous papers [13, 11, 23, 19]. We use a similar technique as in [11]. Before swapping the pointers associated with a node we flag the incoming pointers to the node and its possible predecessor and mark the outgoing pointers from the same. Note that because a backlink is neither used for traversal nor for injecting a modify operation, we do not need to mark/flag it. Flagging ensures that an operation does not have to travel a long *chain* of backlinks. Once a link is flagged or marked it can not be a *point of injection* of a *new* ADD or REMOVE. However, in the conflict between a REMOVE of a category 3 node, and therefore shifting its predecessor, and a concurrent REMOVE of the predecessor node itself, we give priority to the former. Also to guarantee a single pointer travelling for recovery from failure due to a concurrent modify operation, we use a *prelink* at each node. It connects a node to its order-node before any of the outgoing pointers are marked. The flagging and marking are performed in a definite order to avoid a malformed structure of the BST and to ensure required priorities between conflicting operations. The flag-mark order of links for a category 3 node is as following - (I) flag the incoming order-link, (II) set the prelink, (III) mark the outgoing right-link, (IV) flag the parent-link of the predecessor incoming to that, (V) flag the incoming parent-link, (VI) mark the outgoing left-link and finally (VII) mark the outgoing left-link of the predecessor. For a node belonging to category 1 or 2, because there is no node between its order-node and itself, steps (IV), (VI) and (VII) do not happen. Having performed the flagging and marking we update the flagged links according to the category of the node as described before. See Fig. 3.

Because we follow orderly the modifications of the links, it never allows a node to be missed by a traversal in the BST unless both its incoming links are pointed away. However, because a node may shift "upward" to replace its successor, the interval associated with its successor, may shift "rightward" i.e. to the right subtree of the node after the successor is removed. Therefore, in order to terminate a traversal at the correct location, we have the stopping criterion given in Condition 1. It follows from the fact that the threaded left-link of a node is connected to the node itself and a traversal in the BST uses the order in the equivalent list.

CONDITION 1. *Let k be the search key and k_{curr} be the key of the current node in the search path. If $(k = k_{curr})$*

4

This stopping criterion not only solves the problem of synchronization between a concurrent REMOVE and a traversal for a predecessor query but also enables to achieve bound on the length of the traversal path. We shall explain that in section 4. A similar stopping criterion is used in [8] for conventional doubly-threaded BSTs.

We can observe that in our BST design, two modify operations that need to change two disjoint memory-words have significantly improved conditions for progress. For example, to REMOVE a category 2 node the left-link of its predecessor is never marked or flagged, therefore when such a node goes under REMOVE, a concurrent injection of ADD or REMOVE at the predecessor is possible. Also, because of the orderly flagging and marking of the links in more than one atomic steps, even after REMOVE of the node has commenced, a link at it that comes late in the order of flagging-marking, can be modified for injection of an ADD or a REMOVE of another node. These progress conditions are not possible in the existing algorithms that use "node holding" [10, 15]. It shows that our algorithm has improved disjoint-access-parallelism.

3.2 The Implementation

```
1  struct Node {
2    KType k;
     // Three booleans overlap with three unused bits of the pointer ref. left=child[0], right=child[1].
3    {NPtr ref, bool flag, bool mark, bool thread} child[2];
4    NPtr backLink;
5    NPtr preLink;
6  } *NPtr;
     // Global variable with the fixed value of members.
7  Node root[2] = {{−∞, {(&root[0], 0, 0, 1), (&root[1], 0, 0, 1)}, &root[1], null},
     {∞, {(&root[0], 0, 0, 0), (null, 0, 0, 1)}, null, null}};
```

We consider our concurrent setup as a shared memory machine in which processes are fully asynchronous and arbitrary failure halting is allowed. The read and write of a single memory-word is guaranteed to be atomic. The system provides atomic single-word CAS primitives. CAS(R, *old*, *new*) returns true, if ($old = R$), after updating R to *new*, else it returns false. We steal three bits from a pointer, for (a) thread indicator, (b) mark-bit and (c) flag-bit, see line 3. This is easily possible in many high level languages, including C/C++. We assume our algorithm be implemented with a safe lock-free memory reclamation scheme like hazard-pointer [18].

A typical node x in our BST implementation is represented by a structure consisting of five memory-words corresponding to - (a) a key k, (b) array *child*[2] containing two child pointers : $child[0] := left(x)$ and $child[1] := right(x)$, (c) a *backLink* and (d) a *preLink*. The bit sequence corresponding to boolean variables overlapping three stolen bits from a link is represented as (f, m, t). One or more bits together can be set or unset using a single atomic CAS operation over the pointer containing it. The node structure is shown in lines 1 to 6. We use two dummy nodes as global variables represented by a two member array of node-pointers *root*. The keys $-\infty$ and ∞ are stored in the two members of *root* and they can never be deleted. Node *root*[0] is left-child and predecessor of the node *root*[1], see line 7 and figure 2 (c).

3.2.1 Locating a Node

The set operations - CONTAINS, ADD and REMOVE, need to perform a predecessor query using a given key k to locate an interval $[k_i, k_j]$, s.t. either $\{k_i \leq k < k_j\}$ or $\{k_i \geq k > k_j\}$, where $x(k_i)$ and $x(k_j)$ are two nodes in the BST. The function LOCATE is used for that, which starts from a specified set of two consecutive nodes {prev, curr} and follows the symmetric order of the internal BST to terminate at such a location $\{x(k_{prev}), x(k_{curr})\}$. The return value of LOCATE can be 0, 1 or 2 depending on whether the key k is less than, greater than or equal to the key k_{curr} at the termination point, line 10. If $k_{curr} \neq k$ then the desired interval is associated with the threaded outgoing link from $x(k_{curr})$ in the direction indicated by return value of LOCATE, 0 denotes left and 1 denotes right. The termination criterion for LOCATE implements Condition 1.

As we mentioned before, we can make a traversal eagerly help pending REMOVE operations, even though it is not obstructed by them, in the situations in which proportion of REMOVE increases. If that is done then a traversal cleans a node whose marked right-link it encounters during execution by calling the function CLEANMARK. This functionality can be enabled using a boolean variable as an input argument to every Set operation which is further passed to the LOCATE that it performs. This is not done if the implementation does not demand such eager helping. See line 14 to 20. To perform a CONTAINS operation of key k, we start from the location $\{x(\infty), x(-\infty)\}$ represented by the global variables. Having performed LOCATE, the return value itself indicates whether the key k was present in the BST at the point of termination of LOCATE. If LOCATE returns 2 then CONTAINS returns true otherwise false, line 26 to 30.

```
     // Locate returns 2 if key is found otherwise 0 or 1 if the last visited link is left or right respectively.
 8   int Locate(NPtr& prev, NPtr& curr, KType k)
 9   while true do
         // Function cmp returns 2 := equal, 1 := greater than and 0 := less than
10       dir = cmp(k, curr→e);
11       if (dir = 2) then return dir;
12       else
13           (R, ∗, m, t) = curr→child[dir];
             /* If eager helping is required then help cleaning the current node if its right child pointer has been marked; this
             cleaning can not fail. */
14           if ((m = 1) and (dir = 1)) then
15               newPrev = prev→backLink;
16               CleanMarked(prev, curr, dir);
17               prev = newPrev;
18               pDir = cmp(k, prev→k);
19               curr = (prev→child[pDir])·ref;
20               continue;

21           if t then
                 // Check the stopping criterion.
22               nextE = R→k;
23               if ((dir = 0) or (k < nextE)) then
24                   return dir;
25               else  prev = curr; curr = R;
```

```
26   bool Contains(KType k)
     // Initialize the location variables with the global variables.
27   prev = &root[1]; curr = &root[0];
28   dir = Locate(prev, curr, k);
29   if (dir = 2) then return true;
30   else  return false;
```

3.2.2 Remove operation

```
31   bool Remove(KType k)
     // Initialize the location variables as before.
32   prev = &root[1]; curr = &root[0];
33   dir = Locate(prev, curr, k− ϵ);                                              // locate order-link
34   (next, f, ∗, t) = curr→child[dir];
35   if (k ≠ next→k) then  return false;
36   else
         // flag the order-link
37       result = TryFlag(curr, next, prev, true);
38       if (prev→child[dir].ref = curr) then
39           CleanFlag(curr, next, prev, true);

40   return result;
```

To Remove a node corresponding to key k, starting from $\{x(\infty), x(-\infty)\}$ we locate the link corresponding to the interval containing the key $(k - \epsilon)$ line 33. If the Locate terminates at $\{x(k_{prev}), x(k_{curr})\}$ then $suc(x(k_{curr}))$ is the desired node to remove if k matches with its key. $suc(x(k_{curr})$ is the node pointed by the threaded right-link of $x(k_{curr})$ and this link is indeed the order-link of $x(k)$. If $x(k)$ is located then we try to flag its order-link using TryFlag, line 37, in order to perform the step (I) of Remove.

TryFlag, line 41 to 58, returns true only if the operation performing it could successfully flag the desired link at line 45. If the CAS step to atomically flag the link fails then it checks the reason of failure. If it fails because some other thread already had successfully flagged the link, false is returned, line 50. If it fails because the link was marked then it first helps the operation that marked the link, line 51. It could also fail to flag a threaded link because of an Add of a new node. In both these cases it moves a step back and locates the node whose parent-link was desired to be flagged. The address for starting the recovery is saved in the variable back. If the node is not located then it implies that some other thread already removed the target node and so TryFlag just returns false, line 57.

Having performed the TryFlag, Remove checks whether the target node is still there. If the target node is found then Remove goes to clean the flag at the order-link of the target node, using the function CleanFlag, line 39. The function CleanFlag is also used to clean the flag of the parent-link of a node, and therefore a boolean variable is passed to it to inform about the thread-bit of the flagged link. If the link is an order-link then, for all categories of nodes, the next step is to set the prelink and then mark the right-link, see lines 72 to 87. If the CAS to mark the right-link fails because of its flagging, and if the right-link is threaded then it indicates that the node itself is being shifted to replace its successor by a concurrent Remove operation. We give priority to the shifting operation and therefore before proceeding it helps. As before, we need to recover from failures and so a node back stores the

```
41  bool TryFlag(NPtr& prev, NPtr& curr, NPtr& back, bool isThread)
42  while true do
43        pDir = cmp(curr→k, prev→k) & 1;                    // If cmp returns 2 then curr is the left link of prev; so pDir is changed to 0.
          // Try atomically flagging the parent link.
44        t = isThread;
45        result = CAS(prev→child[pDir], (curr, 0, 0, t), (curr, 1, 0, t));
46        if result then  return true;
47        else
              /* The CAS fails, check if the link has been marked, flagged or the curr node got deleted.  If flagged, return false; if
              marked, first clean it; else just proceed */
48            (newR, f, m, t) = prev→child[pDir];
49            if (newR = curr) then
50                if f then  return false;
51                else if m then
52                    CleanMarked(prev, pDir);

53                prev = back;
54                pDir = cmp(curr→k, prev→k);
55                newCurr = (prev→child[newPDir])·ref; Locate(prev, newCurr, curr→k);
56                if (newCurr ≠ curr) then
57                    return false;

58            back = prev→backLink;
```

```
59  void TryMark(NPtr& curr, int dir)
60  while true do
61        back = curr→backLink;
62        (next, f, m, t) = curr→child[dir];
63        if (m == 1) then  break;
64        else if (f == 1) then
65            if (t == 0)  then
66                CleanFlag(curr, next, back, false); continue;

67            else if ((t == 1) and (dir == 1)) then
68                CleanFlag(curr, next, back, true); continue;

69        result = CAS(curr→child[dir], (next, 0, 0, t), (next, 0, 1, t));          // Try atomically marking the child link.
70        if result then  break;
```

address to restart the recovery from. Here one has to be careful that if back was pointed at the successor node under remove then we need to change it before going to help, see line 77. Having marked the right-link, Remove proceeds to take further steps in the function CleanMark. Also, before marking the right-link, the prelink gets pointed to the correct order-node, line 85.

If CleanFlag is performed on a link that is not threaded then it is helping an operation that has flagged the parent of a node or of its predecessor. In that case it is determined whether the link is the parent-link of a node under Remove or that of its predecessor by checking the flag, mark and thread bits of the right-link. The right-link of a node being removed is marked, whereas that for the predecessor of a node being removed is always flagged and threaded. Accordingly, either step (V) (lines 107 to 118) or step (VI) (lines 94) are performed. To perform the step (VI) the outgoing left-link of a category 3 node is marked, and for that the function TryMark is used. TryMark atomically inserts the mark-bit at a link specified by its direction outgoing from a node, see lines 59 to 70. Note that, if the CAS to atomically mark a link fails due to a concurrent flagging of the link then TryMark helps the concurrent operation that flagged the link, except in the case when the link is a threaded left-link. This is because a flagged and threaded left-link of a node indicates that the node is a category 1 node under remove and we give priority to the operation that must have already flagged its right link to shift it.

Because the last step for category 1 and category 2 nodes is flagging their parent-link before the pointers are appropriately swapped, CleanFlag also includes the final swapping of pointers for such nodes. Having determined that a node belongs to category 1 or 2 by checking the equality of left-node and order-node at line 97, it performs the pointer swapping for category 1 nodes in lines 100 and 101. Pointer swapping steps to clean a category 2 node is shown between lines 103 and 106.

The CleanMark function is used to help an operation that puts a mark-bit at a link. Depending on whether the link is a left- or a right-link, it takes different steps. If the link happens to be a right-link then, for category 1 and 2 nodes, the final step of flagging the parent is executed; otherwise for a category 3 node the parent of its predecessor is flagged. To determine the category of a node, its order-node is checked if it coincides with its left-child. If it is a category 3 node then CleanMark attempts to flag the parent of the order-node. On failing because a concurrent operation had marked the link, it helps that operation. However, after helping it may be that the node under Remove changes from a category 3 to a category 2 node, so its status is checked again, see lines 123 to 140.

A mark-bit at a left-link indicates that it is either coming out from a category 3 node or its predecessor. In the

```
71  void CleanFlag(NPtr& prev, NPtr& curr, NPtr& back, bool isThread)
72  if (isThread ) then
           // Cleaning a flagged order-link
73    │    while true do
           │      // To mark the right-child link
74    │    │    (next, f, m, t) = curr→child[1];
75    │    │    if (m) then break;
76    │    │    else if (f) then
           │    │      // Help cleaning if right-child is flagged
77    │    │    │    if (back = next) then
           │    │    │      // If back is getting deleted then move back.
78    │    │    │    │    back = back→backLink;
79    │    │    │    backNode = curr→backLink;
80    │    │    │    CleanFlag(curr, next, backNode, t);
81    │    │    │    if (back = next) then
82    │    │    │    │    pDir = cmp(prev→k, backNode→k);
83    │    │    │    │    prev = back→child[pDir];
84    │    │    else
85    │    │    │    if (curr→preLink ≠ prev) then curr→preLink = prev;
86    │    │    │    result = CAS(curr→child[1], (next, 0, 0, t), (next, 0, 1, t)) ;              // Try marking the child link.
87    │    │    │    if result then break;
88    │    CleanMark(curr, 1);
89  else                                                                                         // Cleaning a flagged parent-link
90    │    (right, rF, rM, rT) = curr→child[1];
91    │    if (rM) then                                                                           // The node is to be deleted itself
92    │    │    (left, lF, lM, lT) = curr→child[0];
93    │    │    preNode = curr→preLink;
94    │    │    if (left ≠ preNode) then // A category 3 node
95    │    │    │    TryMark(curr, 0);
96    │    │    │    CleanMark(curr, 0);
97    │    │    else                                                                              // This is a category 1 or 2 node
98    │    │    │    pDir = cmp(curr→k, prev→k);
99    │    │    │    if (left = curr) then                                                        // A category 1 node
100   │    │    │    │    CAS(prev→child[pDir], (curr , f, 0, 0), (right, 0, 0, rT));
101   │    │    │    │    if (!rT) then CAS(right→backLink, (curr, 0, 0, 0), (prev, 0, 0, 0));
102   │    │    │    else                                                                         // A category 2 node
103   │    │    │    │    CAS(preNode→child[1], (curr, 1, 0, 1), (right, 0, 0, rT));
104   │    │    │    │    if (!rT) then CAS(right→backLink, (curr, 0, 0, 0), (prev, 0, 0, 0));
105   │    │    │    │    CAS(prev→child[pDir], (curr, 1, 0, 0), (preNode, 0, 0, rT));
106   │    │    │    │    CAS(preNode→backLink, (curr, 0, 0, 0), (prev, 0, 0, 0));
107   │    else if (rt and rF) then
           │      // The node is moving to replace its successor
108   │    │    delNode = right;
109   │    │    while true do
110   │    │    │    parent = delNode→backLink;
111   │    │    │    pDir = cmp(curr→k, prev→k);
112   │    │    │    (*, pF, pM, pT) = parent→child[pDir];
113   │    │    │    if (pM) then CleanMark(parent, pDir);
114   │    │    │    else if (pF) then break;
115   │    │    │    else if (CAS(parent→child[pDir], (curr, 0, 0, 0), (curr, 1, 0, 0))) then
116   │    │    │    │    break;
117   │    │    backNode = parent→backLink;
118   │    │    CleanFlag(parent, curr, backNode, true);
```

first case, the final step i.e. step (VII) to mark the left-link of predecessor of a category 3 node is performed, line 142. In the second case the pointer swapping steps of a category 3 nodes are performed, see lines 147 to 160.

Return value of Remove is that of the absolutely first TryFlag that it performs if the node was located.

3.2.3 Add operation

To Add a new data-node with key k in the BST, we Locate the target interval $[k_i, k_j]$, associated with a threaded link, containing key k. If Locate returns 2 then key is present in the BST and therefore Add returns false. If it returns 0 or 1 then we create a new node containing the key k. Its left-link is threaded and connected to itself and right-link takes the value of the link to which it needs to be connected, line 171. Note that when a new node is added, both its children links are threaded. Also, its backlink is pointed to the node $x(k_i)$. The link which it needs to connect to is modified in one atomic step to point to the new node using a CAS. If the CAS succeeds then true is returned. On failure, it is checked whether the target link was flagged, marked or another Add operation succeeded to insert a new node after we read the link. In case a new node is inserted, we start locating for a new proper link starting with the location comprising nodes at the two ends of the changed link. On failure due to marking or flagging of the current link, the concurrent Remove operation is helped. Recovery from failure due to a flagging or marking by a concurrent Remove operation makes it to go one link back following the backlink of prev. After

```
119  void CLEANMARK(NPtr& curr, int markDir)
120  (left, lF, lM, lT) = curr→child[0];
121  (right, rF, rM, rT) = curr→child[1];
122  if (markDir == 1) then
         /* The node is getting deleted itself.  if it is category 1 or 2 node, flag the incoming parent link; if it is a category 3
         node, flag the incoming parent link of its predecessor.  */
123      while true do
124          preNode = delNode→preLink;
125          if (preNode == left) then                                              // Category 1 or 2 node.
126              parent = delNode→backLink;
127              back = parent→backLink;
128              TRYFLAG(parent, curr, back, true);
129              if (parent→child[pDir].ref == curr) then
130                  CLEANFLAG(parent, curr, back, true);
131                  break;
132          else
                 // Category 3 node.
133              preParent = preNode→backLink;
134              (∗, pPF, pPM, pPT) = preParent→child[1];
135              backNode = preParent→backLink;
136              if (pM) then  CLEANMARK(preParent, 1);
137              else if (pF) then
138                  CLEANFLAG(preParent, preNode, backNode, true);break;
139              else if (CAS(parent→child[pDir], (curr, 0, 0, 0), (curr, 1, 0, 0))) then
140                  CLEANFLAG(preParent, preNode, backNode, true);break;
141  else
         // The node is getting deleted (†) or moved to replace its successor (‡).
142      if (rM) then
             // (†) clean its left marked link.
143          preNode = curr→preLink;
144          TRYMARK(preNode, 0);
145          CLEANMARK(preNode, 0);
146      else if (rt and rF) then
             // (‡) change links accordingly
147          delNode = right;
148          delNodePa = delNode→backLink;
149          preParent = curr→backLink;
150          pDir =cmp(delNode→k, delNodePa→k);
151          (delNodeL, ∗, ∗, dlT) = delNode→child[0];
152          (delNodeR, ∗, ∗, drT) = delNode→child[1];
153          CAS(preParent→child[1], (curr, f, 0, 0), (left, lF, 0, lT));
154          if (!lT) then CAS(left→backLink, (curr, 0, 0, 0), (preParent, 0, 0, 0));
155          CAS(curr→child[0], (left, 0, 1, lT), (delNodeL, 0, 0, 0));
156          CAS(delNodeL→backLink, (delNode, 0, 0, 0), (curr, 0, 0, 0));
157          CAS(curr→child[1], (right, 1, 0, 1), (delNodeR, 0, 0, drT));
158          if (!drT) then CAS(delNodeR→backLink, (delNode, 0, 0, 0), (curr, 0, 0, 0));
159          CAS(delNodePa→child[pDir], (delNode, 1, 0, 0), (curr, 0, 0, 0));
160          CAS(curr→backLink, (preParent, 0, 0, 0), (delNodePa, 0, 0, 0));
```

locating the new proper link, ADD is retried.

4 Correctness and Complexity

In this section we first show that executions in our algorithm produce a correct BST with linearizable operations. Then we prove the lock-free progress property and finally we discuss its amortized step complexity.

4.1 Correctness

Here we present a proof-sketch of correctness, due to space constraints. A detailed proof can be found in the technical report [6]. First we give classification of nodes in a BST Υ implemented by our algorithm.

DEFINITION 1. *A node $x \in \Upsilon$ is called logically removed if its right-link is marked and $\exists \, y \in \Upsilon$ s.t. either $left(y) = x$ or $right(y) = x$.*

DEFINITION 2. *A node $x \in \Upsilon$ is called physically removed if $\nexists \, y \in \Upsilon$ s.t. either $left(y) = x$ or $right(y) = x$ or $y \rightarrow backLink = x$.*

DEFINITION 3. *A node $x \in \Upsilon$ is called regular if it is neither logically removed nor physically removed.*

A node ever inserted in Υ has to fit in one of the above categories. At a point in the history of executions, the BST formed by our algorithm contains elements stored in regular nodes or logically removed nodes. Before we show that the return values of the set operations are consistent with this definition according to their linearizability, we have

```
161  bool ADD(KType k)
162  prev = &root[1]; curr = &root[0];
     /* Initializing a new node with supplied key and left-link threaded and pointing to itself. */
163  node = new Node(k);
164  node→child[0] = (node, 0, 0, 1);

165  while true do
166      dir = LOCATE(prev, curr, k);
167      if (dir = 2) then // key exists in the BST
168          return false;
169      else
170          (R, *, *, *) = curr→child[dir];
             /* The located link is threaded.  Set the right-link of the adding node to copy this value */
171          node→child[1] = (R, 0, 0, 1);
172          node→backLink = curr;
173          result = CAS(curr→child[dir], (R, 0, 0, 1), (node, 0, 0, 0));        // Try inserting the new node.

174          if result then return true;
175          else
                 /* If the CAS fails, check if the link has been marked, flagged or a new node has been inserted.  If marked or
                    flagged, first help cleaning. */
176              (newR, f, m, t) = curr→child[dir];
177              if (newR = R) then
178                  newCurr = prev;
179                  if m then CLEANMARKED(curr, dir);
180                  else if f then
181                      CLEANFLAGGED(curr, R, prev, true);
182              curr = newCurr;
183              prev = newCurr→backLink;
```

to show that the operations work as intended and the structure formed by the nodes operated with these operations maintains a valid BST. We present some invariants maintained by the operations and the nodes in lemmas 1 to 11.

LEMMA 1. *If a* LOCATE(prev, curr, k) *returns dir and terminates at* $[x(k_{prev}), x(k_{curr})]$ *then*

(a) *Either* $k_{curr} \leq k$ *or* $k_{curr} \geq k$.

(b) *If* $k_{curr} \neq k$ *then the link* $x(k_{curr})$→$child[dir]$ *is threaded.*

(c) $x(k_{curr})$ *is not physically deleted.*

LEMMA 2. *A* CONTAINS *operation returns true if and only if the key is located at a non-physically removed node.*

LEMMA 3. *An* ADD *always happens at an unmarked and unflagged threaded link.*

LEMMA 4. *A* REMOVE *always starts by flagging the incoming order-link to a node.*

LEMMA 5. *When a node is inserted, both its left- and right- links both threaded.*

LEMMA 6. *Before a node is logically removed its incoming order-link is flagged and its prelink points to its correct order-node.*

LEMMA 7. *Backlink of a node always points to a non physically removed node.*

LEMMA 8. *An unthreaded left-link can not be both flagged and marked.*

LEMMA 9. *A right-link can not be both flagged and marked.*

LEMMA 10. *A link once marked never changes again.*

LEMMA 11. *If a node gets logically removed then it will be eventually physically removed.*

Lemma 1 follows from the lines 11 and 21. CONTAINS returns true only if LOCATE returns 2 and that happens only if curr is non-physically removed at line 10 during its execution, this proves lemma 2. In the case of a key match, ADD returns false and otherwise it tries adding the key using an atomic CAS. If the CAS fails, it always uses LOCATE to find the desired link before retrying the CAS to add the new node. From this observation and using 1(b), lemma 3 follows. By lemma 1 if $x(k)$ is present in the tree then LOCATE(prev, curr, $(k-\epsilon)$) will always terminate at a location such that curr is order-node of $x(k)$ and that establishes lemma 4. Lemma 5 follows from lines 164 and 171. Line 153 ensures that even if the function CLEANFLAG helps a pending REMOVE, before it could successfully mark

the right-link at line 86 the flag that was put on the order link at line 37 is copied to the changed order-link. Also, the line 85 inside the while loop ensures that prelink is always set to the order-node. That proves the correctness of lemma 6. When a node is added its backlink is pointed to curr at line 172. Before a REMOVE operation returns it ensures that the backlinks of the predecessor, left-child and right-child, if present for the node under REMOVE, are appropriately updated at lines 101, 104, 106, 154, 156, 158 and 160. Hence, by induction, lemma 7 is proved. In our algorithm we always use an atomic CAS to put a flag or mark-bit in a pointer. Whenever a CAS fails we check the possible reason. The function TRYMARK helps cleaning the flag in all cases except when the link has direction 0 and it is threaded, line 67. These observations prove lemmas 8 and 9. Lemma 10 follows from lemmas 8 and 9. Lemma 10 proves that once the right-link of a node is marked, it can not be reversed and if the thread invoking REMOVE to mark it becomes faulty then eventually another thread invoking a possible ADD or REMOVE which gets obstructed, will help to complete the physical removal. Having proved the lemmas listed above, it is trivial to observe that whenever a pointer is dereferenced it is not null. And, by the initialization of the global variable to ∞ and $-\infty$, at line 7, the two starting nodes are never deleted.

Hence, we state proposition 1 whose proof will follow by the above stated lemmas and the fact that a thread always takes a correct "turn" during traversal according to the symmetric order of the internal BST.

PROPOSITION **1.** *The union of the regular and logically removed nodes operated under the set operations in the algorithm Efficient Lock Free BST maintain a valid internal Binary Search Tree.*

An execution history in our implementation may consist of ADD, REMOVE and CONTAINS operations. We present the linearization point of the execution of these operations. Proving that a history consisting of concurrent executions of these operations is legal will be ordering these linearization points. The linearization points of the operations are as following:

ADD - For a successful ADD operation, execution of the CAS at line 173 will be the linearization point. For an unsuccessful one the linearization point will be at line 10 where a key in the tree is found matched.

REMOVE - For a successful REMOVE operation the linearization point will be the successful CAS that swaps the flagged parent link. For an unsuccessful one there may be two cases - (a) if the node is not located then it is treated as an unsuccessful CONTAINS and its linearization point will be accordingly (b) if the node is located but its order-link got flagged by another concurrent REMOVE then its linearization point is just after the linearization point of that REMOVE.

CONTAINS - Our algorithm varies according to the read-write load situation. In case we go for eager helping by a thread performing LOCATE, a successful CONTAINS shall always return a regular node. However, if we opt otherwise then a successful CONTAINS returns any non-physically removed node. In both situations a successful CONTAINS will be linearized at line 10. An unsuccessful one, if the node never existed in the BST, is linearized at the start point. And, if the node existed in the BST when the CONTAINS was invoked but got removed during its traversal by a concurrent REMOVE then the linearization point will be just after the successful CAS operation that physically removed the node from the BST.

Following the linearization points as described above we have proposition 2:

PROPOSITION **2.** *The Set operations in the algorithm Efficient Lock Free BST are linearizable.*

4.2 Lock-Freedom

The lemmas 6, 8, 9 and 10 imply the following lemma.

LEMMA **12.** *If* REMOVE(x) *and* REMOVE(y) *work concurrently on nodes x and y then without loss of generality*

(a) *If x is the left-child of y and both x and y are logically deleted then* REMOVE(x) *finishes before* REMOVE(y).

(b) *If x is the right-child of y and both x and y are logically deleted then* REMOVE(y) *finishes before* REMOVE(x).

(c) *If x is the predecessor of y and the order-links of both x and y have been successfully flagged then* REMOVE(y) *finishes before* REMOVE(x).

(d) *If x is the predecessor of y and x has been logically deleted then* REMOVE(x) *finishes before the order-link of y could be successfully flagged.*

(e) *If x is the left-child of the predecessor of y and the incoming parent-link of x has been successfully flagged then* REMOVE(x) *finishes before* REMOVE(y).

(f) *If x is the left-child of the predecessor of y and the left-link of the predecessor of y has been successfully marked then* REMOVE(y) *finishes before* REMOVE(x).

(g) *In all other cases* REMOVE(x) *and* REMOVE(y) *do not obstruct each other.*

By the description of our algorithm, a non-faulty thread performing CONTAINS will always return unless its search path keeps on getting longer forever. If that happens, an infinite number of ADD operations would have successfully completed adding new nodes making the implementation lock-free. So, it will suffice to prove that the modify operations are lock-free. Considering a thread t performing a pending operation op on a BST Υ and takes infinite steps, and, no other modify operation completes after that. Now, if no modify operation completes then the tree remains unchanged forcing t to retract every time it wants to inject its own modification on Υ. This is possible only if every time op finds its injection point flagged or marked. This implies that a REMOVE operation is pending. It is easy to observe in the function ADD that if it gets obstructed by a concurrent REMOVE then before retrying after recovery from failure it helps the pending REMOVE by taking all the remaining steps of that. Also from lemma 12, whenever two REMOVE operations obstruct each other, one finishes before the other. It implies that whenever two modify operations obstruct each other one finishes before the other and so the tree changes. It is contrary to our assumption. Hence, by contradiction we show that no non-faulty thread shall remain taking infinite steps if no other non-faulty thread is making progress. This proves the following proposition 3.

PROPOSITION **3.** *Lock-freedom is guaranteed in the algorithm Efficient Lock Free BST.*

4.3 Complexity

Having proved that our algorithm guarantees lock-freedom, though we can not compute worst-case complexity of an operation, we can definitely derive their amortized complexity. We derive the amortized step complexity of our implementation by the accounting method along the similar lines as in [11, 20].

In our algorithm, an ADD operation does not have to hold any pointer and so does not obstruct an operation. We observe that after flagging the order-link of a node, a REMOVE operation op_r takes only a constant number of steps to flag, mark and swap pointers connected to the node and to its predecessor, if any, in addition to setting the prelink of the node under REMOVE. So, if a modify operation op gets obstructed by op_r then it would have to perform only a constant number of extra steps in order to help op_r provided that during the help it is not obstructed further by another REMOVE operation op_r'. This observation implies following lemma

LEMMA **13.** *An obstructing operation op makes an obstructed operation op$'$ take only a constant number of extra steps in order to finish the execution of op$'$.*

Now for an execution E, let n be the number of nodes at the beginning of E. Let $\mathcal{O}$ be the set of operations and let $\mathcal{S}$ be the set of steps taken by all $op \in \mathcal{O}$. Considering the invocation point $t_i(op)$ of op to be the time it reads the root node, and response point $t_r(op)$ to be the time it reads or writes at the last link before it leaves the tree, its *interval contention* c_I is defined as the total number of operations op' whose execution overlaps the interval $[t_i(op),\ t_r(op)]$ [1]. We define a function $f : \mathcal{S} \mapsto \mathcal{O}$ such that:

(a) Had there been no contention, all the essential steps s, representing read, write and `CAS` taken by an operation op is mapped to op by f.

(b) In case of contention, any failed `CAS` by an operation op is mapped to the operation op' whose successful `CAS` causes the failure.

(c) If an extra read is performed by a traversal due to an added node to the set of existing nodes by a concurrent ADD operation op then it is mapped to op.

(d) Any read, write or `CAS` step s taken by an operation op after the first failed `CAS` and before retrying at the same link i.e. during helping is mapped to an operation op' that performed the successful `CAS` in order to make op help it. This includes resetting of prelink, if needed.

Now we bound the length of the traversal path for a predecessor query in our implementation. Note that, all the set operations have to perform a predecessor query by key k to LOCATE an interval $[k_i,\ k_j]$ s.t. either $\{k_i \leq k < k_j\}$ $\{k_i \geq k > k_j\}$ and $x(k_i)$ and $x(k_j)$ are two nodes in the BST. Let us define the *access-node* of an interval as the order-node of the node that the interval associates with. We define *distance* of an interval from a traversal op as the number of links that op traverses from its current location to read the access-node of the interval. Suppose that at $t_i(op)$ there are n nodes in the BST Υ. Clearly, distance of the interval associated with any node for op at $t_i(op)$ is $O(H(n))$. It is also easy to observe that unless op accesses a node x, distance of any node in the subtree rooted at it is $O(ht(x))$. When op travels in the left subtree of a category 3 node x and if it gets removed, the interval associated with it gets associated with the leftmost node in the right subtree of the node that replaces it and so the distance of that interval from op changes. There is no such change for a category 1 or category 2 node. However, once an interval gets associated with the leftmost node in the right-subtree of x, which is obviously a category 1 node, its distance can not become more than $O(ht(x))$ from a traversal that has accessed x. These observations show that the

path length of a traversal in our lock-free BST is bounded by $2H(n)$, if no node is added in the traversal path. In case a node is added, the extra read cost is mapped to the concurrent ADD.

Having identified the operations to map for a step, it is easy to observe that an operation op' to which a step s by an operation op is mapped, always has its execution interval overlapping $[t_i(op), t_r(op)]$. So, because (a) the traversal path is bounded by $O(H(n))$, (b) a constant number of steps are needed for a modify operation after locating its target (node or link) and (c) by lemma 13, to an operation op only a constant number of extra steps can be mapped by any concurrent operation op' counted in c_I, we can conclude that the amortized step complexity of a set operation in our lock-free BST algorithm is $O(H(n) + c_I)$.

As we described in the section 3, we give choice of eager helping to operations according to the read-write load. Now if that happens, we do not need to count all the operations whose executions overlap the interval $[t_i(op), t_r(op)]$. We can then use a tighter notion of *point contention* c_P [2], which counts the maximum number of operations that execute concurrently at any point in $[t_i(op), t_r(op)]$. In that case, given the above discussion, along the similar lines as presented in [11], we can show that for any execution E, the average amortized step complexity of a set operation op in our algorithm will be

$$\hat{t}_{op \in E} \in O\left(\frac{\sum_{op \in E}(H(n(op)) + c_P(op))}{|\{op \in E\}|}\right)$$

where $n(op)$ is the number of nodes in the BST at the point of invocation of op and $c_P(op)$ is its point contention during E. That concludes the amortized analysis of our algorithm.

It is straightforward to observe that the number of memory-words used by a BST with n nodes in our design is $5n$.

5 Conclusion and Future Work

In this paper we proposed a novel algorithm for the implementation of a lock-free internal BST. Using amortized analysis we proved that all the operations in our implementation run in time $O(H(n) + c)$. We solved the existing problem of "retry from scratch" for modify operations after failure caused by a concurrent modify operation, which resulted into an amortized step complexity of $O(cH(n))$. This improvement takes care of an algorithmic design issue for which the step complexity of modify operations increases dramatically with the increase in the contention and the size of the data structure. This is an important improvement over the existing algorithms. Our algorithm also comes with improved disjoint-access-parallelism compared to similar lock-free BST algorithms. We also proposed a conservative helping technique which adapts to read-write load on the implementation. We proved the correctness showing linearizability and lock-freedom of the proposed algorithm.

We plan to thoroughly evaluate our algorithm experimentally vis-a-vis existing concurrent set implementations.

6 Acknowledgement

References

[1] Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[2] H. Attiya and A. Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):444–468, 2003.

[3] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the 5th ACM SPAA*, pages 261–270, 1993.

[4] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM PPoPP*, pages 257–268, 2010.

[5] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM PPoPP*, pages 329–342, 2014.

[6] B. Chatterjee, N. Nguyen, and P. Tsigas. Efficient lock-free binary search trees. Technical Report 2014:05, ISNN 1652-926X, Department of Computer Science and Engineering, Chalmers University of Technology, 2014.

[7] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *Proceedings of the 17th ACM PPoPP*, pages 161–170, 2012.

[8] D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th ACM PPoPP*, pages 343–356, 2014.

[9] F. Ellen, P. Fatourou, J. Helga, and E. Rupert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2013 ACM PODC*, 2014.

[10] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM PODC*, pages 131–140, 2010.

[11] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd ACM PODC*, pages 50–59, 2004.

[12] K. Fraser. *Practical lock-freedom*. PhD thesis, Cambridge University, Computer Laboratory, 2004.

[13] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing*, pages 300–314. Springer, 2001.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[15] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *Proceedinbgs of the 24th Annual ACM SPAA*, pages 161–171, 2012.

[16] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th ACM PODC*, pages 151–160, 1994.

[17] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th ACM SPAA*, pages 73–82, 2002.

[18] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504, 2004.

[19] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM PPoPP*, pages 317–328, 2014.

[20] R. Oshman and N. Shavit. The skiptrie: low-depth concurrent search without rebalancing. In *Proceedings of the 2013 ACM PODC*, pages 23–32, 2013.

[21] A. J. Perlis and C. Thornton. Symbol manipulation by threaded lists. *Communications of the ACM*, 3(4):195–204, 1960.

[22] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

[23] H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In *Principles of Distributed Systems*, pages 240–255. Springer, 2005.

[24] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-free linked-lists. In R. Baldoni, P. Flocchini, and R. Binoy, editors, *Principles of Distributed Systems*, volume 7702 of *LNCS*, pages 330–344. Springer Berlin Heidelberg, 2012.