# Non-blocking Binary Search Trees

Faith Ellen
University of Toronto, Canada

Panagiota Fatourou
University of Crete and
FORTH ICS, Greece

Eric Ruppert
York University, Canada

Franck van Breugel
York University, Canada

## ABSTRACT

This paper describes the first complete implementation of a non-blocking binary search tree in an asynchronous shared-memory system using single-word compare-and-swap operations. The implementation is linearizable and tolerates any number of crash failures. INSERT and DELETE operations that modify different parts of the tree do not interfere with one another, so they can run completely concurrently. FIND operations only perform reads of shared memory.

## Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*Distributed data structures*

## General Terms

Algorithms, Theory

## Keywords

Binary search tree, shared memory, non-blocking, CAS

## 1. INTRODUCTION

Although there are extensive, highly optimized libraries of sequential data structures, libraries of non-blocking distributed data structures are much less comprehensive. Over the past fifteen years, some progress has been made to provide non-blocking implementations of arrays and linked lists (as well as data structures that can be built from them, like stacks, queues, hash tables, and skip lists) in shared-memory distributed systems [4, 6, 10, 14, 21, 23, 25]. The binary search tree (BST) [16] is one of the most fundamental sequential data structures, but comparatively little has been done towards providing non-blocking implementations of it. In the documentation for the ConcurrentSkipListMap class of the Java standard library [18], Lea wrote, "you might wonder why this [non-blocking dictionary implementation using skiplists] doesn't use some kind of search tree instead, which would support somewhat faster search operations. The reason is that there are no known efficient lock-free insertion and deletion algorithms for search trees."

We give the first complete, non-blocking, linearizable BST implementation using only reads, writes, and single-word compare-and-swap (CAS) operations. It does not use large words, so it can be run directly on existing hardware. Updates to different parts of the tree do not interfere with one another, so they can run concurrently. Searches only read shared memory and follow tree edges from the root to a leaf, so they do not interfere with updates, either.

At a high level, our implementation is easy to understand, partly due to its modularity. We give an overview in Section 3. A more detailed description of our implementation, together with pseudocode, is given in Section 4. In the full version [5], we present a complete proof of correctness. It is surprisingly intricate; complex interactions can arise between concurrent operations being performed in the same part of the tree. This makes it difficult to guarantee that the coordination between processes actually does prevent concurrent updates on the same part of the tree from interfering with one another. We also must ensure that searches do not go down a wrong path and miss the element for which they are searching, when updates are happening concurrently. A sketch of the proof is provided in Section 5.

## 2. RELATED WORK

There are implementations of BSTs using locks that support concurrent accesses. For example, Guibas and Sedgewick [9] gave a balanced BST by uncoupling rebalancing operations from updates, as did Kung and Lehman [17], who also proved their implementation correct. Nurmi and Soisalon-Soininen [22] introduced chromatic trees, a leaf-oriented version of red-black trees with relaxed balance conditions and uncoupled update and rebalancing operations. Boyar, Fagerberg, and Larsen [3] modified some of the rebalancing operations of chromatic trees, improving their performance, and gave a proof of correctness of their locking scheme. However, in all of these implementations, a process changing the data structure must lock a number of nearby nodes. This can block searches and other updates from proceeding until the process removes the locks.

The co-operative technique described by Barnes [1] is a method for converting locked-based implementations into non-blocking ones. His idea is to replace locks owned by processes with locks owned by operations. When acquiring a lock on a part of the data structure, for example a node in a tree, an operation writes a description of the work that


*PODC'10,* July 25–28, 2010, Zurich, Switzerland.

it needs to do while holding the lock. Other processes that encounter a locked cell can then help the operation that locked it, so that the lock can eventually be released, even if the process that acquired the lock crashes. Like most general transformations, it can have large overhead when applied to particular implementations. If it were applied to the lock-based tree implementations mentioned above, a process may have to repeatedly help many other operations progress down the tree, which could result in a very long delay until any operation completes.

Valois [24, 25] briefly sketched a possible non-blocking implementation of a node-oriented BST using registers and CAS objects, based on his non-blocking implementation of a linked list, but a full description of this implementation has not yet appeared. His idea is to have an auxiliary node between a tree node and each of its children, to avoid interference between update operations. However, as in his linked list implementation, this approach can give rise to long chains of consecutive auxiliary nodes, which degrade performance, when nodes are deleted. Our implementation is also conceptually simpler than Valois's, requiring one tree pointer to be changed for each update (rather than four).

In his Ph.D. thesis [8], Fraser wrote, "CAS is too difficult to apply directly" to the implementation of BSTs. Instead, he gave a non-blocking implementation of a node-oriented BST using multi-word CAS operations that can atomically operate on eight words spread across five nodes. He provided some justification for the correctness of his implementation, but did not provide a proof of correctness. He described how to build multi-word CAS operations from single-word CAS, but this construction involves substantial overhead.

Bender *et al.* [2] described a non-blocking implementation of a cache-oblivious B-tree from LL/SC operations, but a full version of this implementation has not yet appeared.

Universal constructions can be used to provide wait-free, non-blocking implementations of any data structure, including BSTs. However, because of their generality, they are usually less efficient than implementations tailor-made for a specific data structure. Some universal constructions [12] put all operations into a queue and the operations are applied sequentially, in the order they appear in the queue. This precludes concurrency. In other universal constructions [11, 13] a process copies the data structure (or the parts of it that will change and any parts that directly or indirectly point to them), applies its operation to the copy, and then tries to update the relevant part of the shared data structure to point to its copy. In a BST, the root points indirectly to every node, so no concurrency is possible using this approach, even for updates on separate parts of the tree.

Similarly, shared BST implementations can be obtained using software transactional memory [7, 8, 15]. However, current implementations of transactional memory either satisfy weaker progress guarantees (like obstruction-freedom) or incur high overhead when built from single-word CAS.

# 3. IMPLEMENTATION OVERVIEW

A BST implements the *dictionary* abstract data type. A dictionary maintains a set of keys drawn from a totally ordered universe and provides three operations: INSERT($k$), which adds key $k$ to the set, DELETE($k$) which removes key $k$ from the set, and FIND($k$), which determines whether $k$ is in the set. An *update* operation is either an INSERT or a DELETE. We assume duplicate keys are not permitted in the dictionary, so an insertion of a duplicate key should return FALSE without changing the dictionary. Similarly, an attempt to delete a non-existent key should return FALSE. Our implementation can also store auxiliary data with each key, if this is required by the dictionary application.

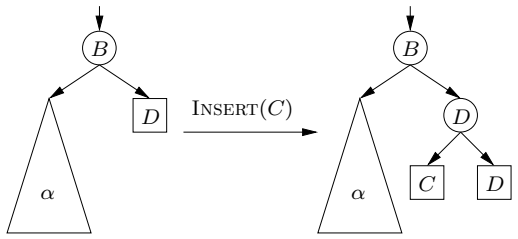


**Figure 1: Insertion in a leaf-oriented BST in a single-process system.**

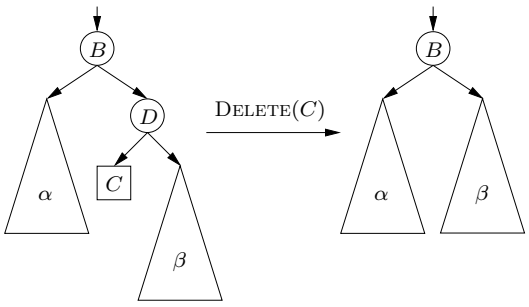


**Figure 2: Deletion in a leaf-oriented BST in a single-process system.**

Our BST implementation is *non-blocking*: starting from any configuration of any infinite asynchronous execution, with any number of crash failures, some operation always completes. It is also *linearizable*. This means that, for every execution, one can assign a *linearization point* to each completed operation and some of the uncompleted operations so that the linearization point of each operation occurs after the operation starts and before it ends, and the results of these operations are the same as if they had been performed sequentially, in the order of their linearization points.

In our implementation, nodes maintain child pointers but not parent pointers. We use a *leaf-oriented* BST, in which every internal node has exactly two children, and all keys currently in the dictionary are stored in the leaves of the tree. (Any auxiliary data can also be stored in the leaves along with the associated keys.) Internal nodes of the tree are used to direct a FIND operation along the path to the correct leaf. The keys stored in internal nodes may or may not be in the dictionary. A leaf-oriented BST also maintains the following BST property: for each internal node $x$, all descendants of the left child of $x$ have keys that are strictly less than the key of $x$ and all descendants of the right child of $x$ have keys that are greater than *or equal* to the key of $x$. As shown in Figure 1 and 2, an insertion replaces a leaf by a subtree of three nodes and a deletion removes a leaf and its parent by making the leaf's sibling a child of the leaf's former grandparent. (In the figures, leaves are square and internal nodes are round.) For both types of updates, only a single child pointer near a leaf of the tree must be changed.

However, simply using a CAS on the one child pointer that an update must change would lead to problems if there

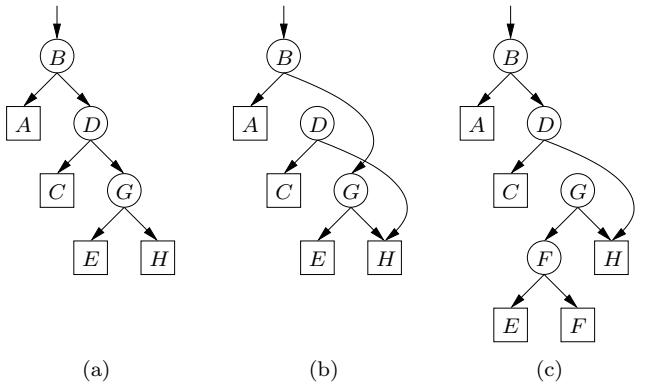


**Figure 3: Problems can occur if updates only** CAS **one child pointer.**

are concurrent updates. Consider the initial tree shown in Figure 3(a). If a DELETE($C$) and a concurrent DELETE($E$) perform their CAS steps right after each other, the resulting tree, shown in Figure 3(b), still contains $E$, which is incorrect. Similarly, if a DELETE($E$) and a concurrent INSERT($F$) perform their CAS steps right after each other, then the resulting tree, shown in Figure 3(c), does not contain $F$ because it is unreachable from the root. Harris [10] avoided analogous problems in his linked list implementation by setting a "marked" bit in the successor pointer of a node before deleting that node from the list. Once the successor pointer is marked, it cannot be changed. We use a similar approach: when deleting a leaf, we mark the parent of the leaf before splicing that parent out of the tree. Once a node is marked, we ensure that its child pointers cannot change.

Ensuring that the child pointers of a marked node do not change is more difficult than in Harris's linked list implementation, because the two child pointers are stored in two different words, so we cannot atomically mark both of them. Instead, we mark a node using a separate *state* field of the node. The *state* field is changed by CAS steps, and is initially set to the value CLEAN. To mark the node, we set its *state* to MARK. We also use this field to *flag* the node, to indicate that an update is trying to change a child pointer of the node. Before an INSERT or DELETE changes either of the node's child pointers, it must change the *state* of the node to IFLAG or DFLAG, respectively. After the child has successfully been changed, the *state* is changed from IFLAG or DFLAG back to CLEAN. This prevents both of the problems shown in Figure 3. (The use of flagging is motivated partly by Fomitchev and Ruppert's linked list implementation [6], although they used flagging purely to improve performance.)

Thus, the key steps in update operations can be described as follows (using the examples of Figure 1 and 2). The INSERT($C$) operation shown in Figure 1 is done using three CAS steps: (1) flag node $D$'s parent, node $B$, (2) change the appropriate child pointer of node $B$, and (3) unflag node $B$. We refer to these three types of CAS steps as *iflag, ichild* and *iunflag* CAS steps, respectively. The DELETE($C$) operation shown in Figure 2 is accomplished using four CAS steps: (1) flag $C$'s grandparent, node $B$, (2) mark $C$'s parent, node $D$, (3) change the appropriate child pointer of $B$, and (4) unflag node $B$. We call these four types of CAS steps *dflag, mark, dchild* and *dunflag* CAS steps, respectively. We refer to ichild and dchild CAS steps collectively as *child* CAS steps. We refer to iflag and dflag CAS steps collectively as *flag* CAS steps. We refer to iunflag and dunflag CAS steps collectively as *unflag* CAS steps.

In some sense, setting the *state* of a node to MARK, IFLAG, or DFLAG is analogous to locking the child pointers of the node: An operation must successfully acquire the lock (by setting the flag) before it can change a child pointer. Marking a node locks the node's child pointers forever, ensuring that they never change again. Viewed in this way, a FIND does not acquire any locks, an INSERT is guaranteed to complete when it acquires a lock on a single node at the site of the insertion, and a DELETE is guaranteed to complete after acquiring locks on just two nodes at the site of the deletion. Since each operation only requires locks on one or two nodes near a leaf of the tree, our locking scheme will not cause serious contention issues, and concurrent updates will not interfere with one another if they are on different parts of the tree. In contrast, the lock-based algorithms described in Section 2 require locking all nodes along a path from the root to a leaf (although not all simultaneously).

To achieve a non-blocking implementation, we use a strategy similar to Barnes's technique, described in Section 2. When an operation flags a node $x$ to indicate that it wishes to change a child pointer of $x$, it also stores a pointer to an Info record (described below) that contains enough information for other processes to help complete the operation, so that the node can be unflagged. Thus, an operation that acquires a lock always leaves a key to the lock "under the doormat" so that another process blocked by a locked door can unlock it after doing a bit of work. We prove this helping mechanism suffices to achieve a non-blocking implementation. The pointer to the Info record is stored in the same memory word as the *state*. (In typical 32-bit word architectures, if items stored in memory are word-aligned, the two lowest-order bits of a pointer can be used to store the *state*.)

Helping can often contribute to poor performance because several processes try to perform the same piece of work. Thus, we choose a conservative helping strategy: a process $P$ helps another process's operation only if the other operation is preventing $P$'s own progress. Since FIND operations do not have to make any changes to the tree (and therefore cannot be blocked from doing so), they never help any other operation. If $P$ is performing an update operation that must flag or mark a node that is already flagged or marked, it helps complete the operation that flagged or marked the node. Then $P$ retries its own update operation.

Once an INSERT operation has successfully performed its first key step, the iflag CAS, there is nothing that can block it from completing the other two key steps, the ichild and iunflag CAS steps. Similarly, once a DELETE operation has successfully performed its first two key steps, the dflag and mark CAS steps, there is nothing that can block it from completing the other two key steps, the dchild and dunflag CAS steps. However, if a DELETE operation successfully performs its first key step, the dflag CAS, and then fails when attempting its mark CAS, it is impossible to complete the operation as planned. This could happen, for example, if a concurrent INSERT has replaced the leaf to be deleted by three new nodes, in which case the flag is no longer on the node whose child pointer must be changed to accomplish the deletion. Thus, if a mark CAS fails, the DELETE operation uses a CAS to remove its flag and restarts the deletion from scratch. The CAS that removes the flag in this case is called

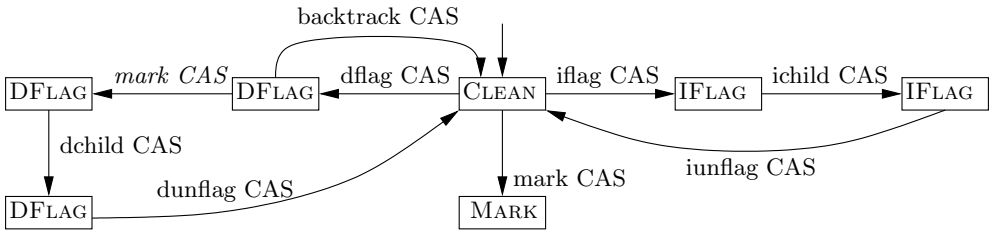


**Figure 4: The effects of successful CAS operations.**

a *backtrack* CAS to distinguish it from a dunflag CAS. (See Figure 4 for a summary of the types of CAS steps.)

The proof of correctness includes several major parts. We show that no value is ever stored in the same CAS object by two different successful CAS steps. This implies that, if a process reads a CAS object twice and sees the same value both times, the CAS object did not change between the two reads. We show that helping is carefully coordinated so that each of the CAS steps required to perform an operation is performed by at most one process (and they are done in the correct order). For example, we show that if a process $P$ performs a successful iflag CAS as part of an INSERT operation, at most one process (either $P$ or a process helping to perform the operation) successfully performs the following ichild CAS, and, if it is successfully performed, then at most one process performs the following iunflag CAS. Similarly, for a DELETE operation, after a dflag CAS succeeds, then either the remaining three CAS steps required to complete the DELETE are successfully performed at most once each and in the correct order, or the DFLAG is removed by a backtrack CAS. In the latter case, the tree is unchanged, and the deletion is retried. We use all of these facts to prove that the nodes together with their child pointers always form a BST with distinct keys in the leaves. This allows us to choose very natural linearization points for successful update operations: each successful INSERT and DELETE operation is linearized at its successful child CAS.

# 4. DETAILED IMPLEMENTATION

## 4.1 Representation in Memory

Our BST is represented using registers, which support read and write, and compare-and-swap (CAS) objects, which support read and CAS operations. If $R$ is a CAS object, then CAS($R$, *old*, *new*) changes the value of $R$ to *new* if the object's value was *old*, in which case we say the CAS was *successful*. A CAS always returns the value the object had prior to the operation. (If a CAS object $R$ does not support reads, a CAS$(R, v, v)$, for any value $v$, will read $R$.) The data types we use are defined in Figure 7.

A *leaf node* has no children. It has a single field, *key*, and we say that it has type Leaf. (If auxiliary data is to be stored in the dictionary, the Leaf node can have additional fields stored in registers.) An *internal node* has two children. It has five fields, *key*, *left*, *right*, *state*, and *info*, and we say it has type Internal. We also say that leaf and internal nodes have type Node. The *key* field of a leaf or internal node is stored in a register, which is initialized with a value when the node is created, and is never changed thereafter. The *left* and *right* child pointers of an internal node are represented by CAS objects. They always point to other nodes. The *state* field has one of four possible values, CLEAN, MARK, IFLAG, or DFLAG, and is initially CLEAN. It is used to coordinate actions between updates acting on the same part of the tree, as described in Section 3. An internal node is called *clean, marked* or *flagged* depending on its *state*. Finally, an internal node has a pointer, *info*, to an Info record. This field is initialized to a null pointer, $\perp$. The *state* and *info* fields are stored together in a CAS object. Thus, an internal node uses four words of memory. The word containing the *state* and *info* fields is called the *update* field of the node.

When an update operation $U$ flags or marks a node, $U$ stores enough information so that another process that encounters the flagged or marked node can complete $U$'s update. We use two types of records, called *IInfo* records and *DInfo* records (referred to collectively as *Info* records) to store this information. When a node is flagged or marked, a pointer to an Info record containing the necessary information is simultaneously stored in the *info* field of the node. To complete an INSERT, a process must have a pointer to the leaf that is to be replaced, that leaf's parent and the newly created subtree that will be used to replace the leaf. This information is stored in an IInfo record, in fields named $l$, $p$ and *newInternal*, respectively. To complete a DELETE, a process must have a pointer to the leaf to be deleted, its parent, its grandparent, and a copy of the word containing the *state* and *info* fields of the parent. This information is stored in a DInfo record in fields named $l$, $p$, $gp$ and *pupdate*, respectively. The fields of Info records are stored in registers.

The data structure is illustrated in Figure 5. The figure shows a part of the tree containing three leaves with keys $A$, $C$ and $E$ and two internal nodes with keys $B$ and $D$. Two update operations are in progress. A DELETE($C$) operation successfully flagged the internal node with key $B$. Then, an INSERT($F$) operation successfully flagged the internal node with key $D$. The Info records for these two updates are shown to the right of the tree. The INSERT is now guaranteed to succeed: either the process performing the INSERT or some other process that is helping it will change the right child of the internal node with key $D$ to point to the newly created subtree (shown at the right) that the IInfo record points to. The DELETE operation is doomed to fail: When the DELETE created the DInfo record, it stored the values it saw in node $D$'s *state* and *info* fields in the *pupdate* field of the DInfo record. When the DELETE (or a process helping it) performs the mark CAS on node $D$, it will use, as the old value, the *pupdate* value stored in the DInfo record. The *state* and *info* fields of the node with key $D$ have since changed, so the mark CAS will fail. This is a desirable outcome (and is the reason insertions flag nodes before changing them): if the DELETE did succeed in changing the right child of the internal node with key $B$ to the leaf with key $E$ (thereby removing the internal node with key $D$ from the

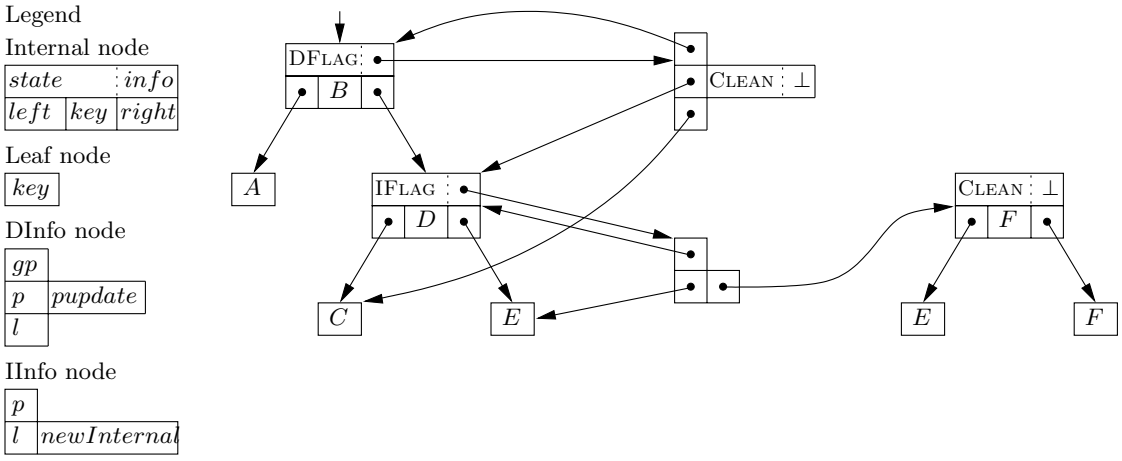


**Figure 5: An example of the data structure. A DELETE($E$) and an INSERT($F$) are in progress. Fields separated by dotted lines are stored in a single word.**

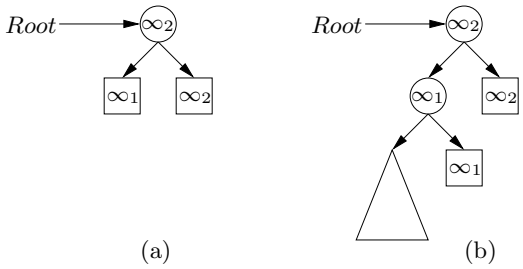


**Figure 6: Trees showing leaves with dummy keys when the dictionary is (a) empty and (b) non-empty.**

```
1   type Update {                    ▷ stored in one CAS word
2       {Clean, DFlag, IFlag, Mark} state
3       Info *info
4   }
5   type Internal {                  ▷ subtype of Node
6       Key ∪ {∞1, ∞2} key
7       Update update
8       Node *left, *right
9   }
10  type Leaf {                      ▷ subtype of Node
11      Key ∪ {∞1, ∞2} key
12  }
13  type IInfo {                     ▷ subtype of Info
14      Internal *p, *newInternal
15      Leaf *l
16  }
17  type DInfo {                     ▷ subtype of Info
18      Internal *gp, *p
19      Leaf *l
20      Update pupdate
21  }
    ▷ Initialization:
22  shared Internal *Root := pointer to new Internal node
        with key field ∞2, update field ⟨Clean, ⊥⟩, and
        pointers to new Leaf nodes with keys ∞1 and
        ∞2, respectively, as left and right fields.
```

**Figure 7: Type definitions and initialization.**

tree), the newly inserted key $F$ would disappear from the tree. Instead, the DFLAG stored in the internal node with key $B$ will eventually be removed by a backtrack CAS, and the DELETE will try deleting key $C$ again.

As shown in Figure 1 and 2, the basic modifications to the tree require changing the child pointers of either a parent or grandparent of a leaf. This would require numerous special cases in the pseudocode to handle situations where the tree has fewer than three nodes. To avoid these special cases, we append two special values, $\infty_1 < \infty_2$, to the universe Key of keys (where every real key is less than $\infty_1$) and initialize the tree so that it contains two dummy keys $\infty_1$ and $\infty_2$, as shown in Figure 6(a). (Thus, the *key* field of a node actually stores an element of Key$\cup\{\infty_1, \infty_2\}$.) Deletion of the leaves with dummy keys is not permitted, so the tree will always contain at least two leaves and one internal node. When the dictionary is non-empty, dictionary elements will be stored in leaves of the subtree shown in Figure 6(b). The shared variable *Root* is a pointer to the root of the tree, and this pointer is never changed. All other named variables used in the pseudocode are local variables (although they may contain pointers to shared memory).

For simplicity we assume nodes and Info records are always allocated new memory locations. In practice, however, memory management is an important issue: it would be more practical to reallocate the memory locations that are no longer in use. Such a scheme should not introduce any problems, as long as a memory location is not reallocated while any process could reach that location by following a chain of pointers. Such safe garbage collection schemes are often provided (for example, by the Java environment) and some options for memory-management schemes that might be used with our algorithm are discussed briefly in Section 6.

## 4.2 The Algorithms

Pseudocode for the BST operations are given in Figure 7, 8 and 9. Comments are preceded by ▷. In variable declarations, T *$x$ declares $x$ to be a pointer to an instance of

```
23 SEARCH(Key k) : ⟨Internal*, Internal*, Leaf*, Update, Update⟩ {
       ▷ Used by INSERT, DELETE and FIND to traverse a branch of the BST; satisfies following postconditions:
       ▷ (1) l points to a Leaf node and p points to an Internal node
       ▷ (2) Either p → left has contained l (if k < p → key) or p → right has contained l (if k ≥ p → key)
       ▷ (3) p → update has contained pupdate
       ▷ (4) if l → key ≠ ∞1, then the following three statements hold:
       ▷    (4a) gp points to an Internal node
       ▷    (4b) either gp → left has contained p (if k < gp → key) or gp → right has contained p (if k ≥ gp → key)
       ▷    (4c) gp → update has contained gpupdate
24     Internal *gp, *p
25     Node *l := Root
26     Update gpupdate, pupdate                                   ▷ Each stores a copy of an update field
27     while l points to an internal node {
28         gp := p                                                ▷ Remember parent of p
29         p := l                                                 ▷ Remember parent of l
30         gpupdate := pupdate                                    ▷ Remember update field of gp
31         pupdate := p → update                                  ▷ Remember update field of p
32         if k < l → key then l := p → left else l := p → right  ▷ Move down to appropriate child
33     }
34     return ⟨gp, p, l, pupdate, gpupdate⟩
35 }
36 FIND(Key k) : Leaf* {
37     Leaf *l
38     ⟨−, −, l, −, −⟩ := SEARCH(k)
39     if l → key = k then return l
40     else return ⊥
41 }
42 INSERT(Key k) : boolean {
43     Internal *p, *newInternal
44     Leaf *l, *newSibling
45     Leaf *new := pointer to a new Leaf node whose key field is k
46     Update pupdate, result
47     IInfo *op
48     while TRUE {
49         ⟨−, p, l, pupdate, −⟩ := SEARCH(k)
50         if l → key = k then return FALSE                       ▷ Cannot insert duplicate key
51         if pupdate.state ≠ CLEAN then HELP(pupdate)             ▷ Help the other operation
52         else {
53             newSibling := pointer to a new Leaf whose key is l → key
54             newInternal := pointer to a new Internal node with key field max(k, l → key),
                   update field ⟨CLEAN, ⊥⟩, and with two child fields equal to new and newSibling
                   (the one with the smaller key is the left child)
55             op := pointer to a new IInfo record containing ⟨p, l, newInternal⟩
56             result := CAS(p → update, pupdate, ⟨IFLAG, op⟩)     ▷ iflag CAS
57             if result = pupdate then {                          ▷ The iflag CAS was successful
58                 HELPINSERT(op)                                  ▷ Finish the insertion
59                 return TRUE
60             }
61             else HELP(result)          ▷ The iflag CAS failed; help the operation that caused failure
62         }
63     }
64 }
65 HELPINSERT(IInfo *op) {
       ▷ Precondition: op points to an IInfo record (i.e., it is not ⊥)
66     CAS-CHILD(op → p, op → l, op → newInternal)                 ▷ ichild CAS
67     CAS(op → p → update, ⟨IFLAG, op⟩, ⟨CLEAN, op⟩)              ▷ iunflag CAS
68 }
```

**Figure 8: Pseudocode for SEARCH, FIND and INSERT.**

```
69  DELETE(Key k) : boolean {
70      Internal *gp, *p
71      Leaf *l
72      Update pupdate, gpupdate, result
73      DInfo *op

74      while TRUE {
75          ⟨gp, p, l, pupdate, gpupdate⟩ := SEARCH(k)
76          if l → key ≠ k then return FALSE                          ▷ Key k is not in the tree
77          if gpupdate.state ≠ CLEAN then HELP(gpupdate)
78          else if pupdate.state ≠ CLEAN then HELP(pupdate)
79          else {                                                    ▷ Try to flag gp
80              op := pointer to a new DInfo record containing ⟨gp, p, l, pupdate⟩
81              result := CAS(gp → update, gpupdate, ⟨DFLAG, op⟩)     ▷ dflag CAS
82              if result = gpupdate then {                           ▷ CAS successful
83                  if HELPDELETE(op) then return TRUE                ▷ Either finish deletion or unflag
84              }
85              else HELP(result)          ▷ The dflag CAS failed; help the operation that caused the failure
86          }
87      }
88  }

89  HELPDELETE(DInfo *op) : boolean {
        ▷ Precondition: op points to a DInfo record (i.e., it is not ⊥)
90      Update result                                                 ▷ Stores result of mark CAS

91      result := CAS(op → p → update, op → pupdate, ⟨MARK, op⟩)      ▷ mark CAS
92      if result = op → pupdate or result = ⟨MARK, op⟩ then {        ▷ op → p is successfully marked
93          HELPMARKED(op)                                            ▷ Complete the deletion
94          return TRUE                                               ▷ Tell DELETE routine it is done
95      }
96      else {                                                        ▷ The mark CAS failed
97          HELP(result)                                              ▷ Help operation that caused failure
98          CAS(op → gp → update, ⟨DFLAG, op⟩, ⟨CLEAN, op⟩)           ▷ backtrack CAS
99          return FALSE                                              ▷ Tell DELETE routine to try again
100     }
101 }

102 HELPMARKED(DInfo *op) {
        ▷ Precondition: op points to a DInfo record (i.e., it is not ⊥)
103     Node *other

        ▷ Set other to point to the sibling of the node to which op → l points
104     if op → p → right = op → l then other := op → p → left else other := op → p → right
        ▷ Splice the node to which op → p points out of the tree, replacing it by other
105     CAS-CHILD(op → gp, op → p, other)                             ▷ dchild CAS
106     CAS(op → gp → update, ⟨DFLAG, op⟩, ⟨CLEAN, op⟩)               ▷ dunflag CAS
107 }

108 HELP(Update u) {                                                  ▷ General-purpose helping routine
        ▷ Precondition: u has been stored in the update field of some internal node
109     if u.state = IFLAG then HELPINSERT(u.info)
110     else if u.state = MARK then HELPMARKED(u.info)
111     else if u.state = DFLAG then HELPDELETE(u.info)
112 }

113 CAS-CHILD(Internal *parent, Node *old, Node *new) {
        ▷ Precondition: parent points to an Internal node and new points to a Node (i.e., neither is ⊥)
        ▷ This routine tries to change one of the child fields of the node that parent points to from old to new.
114     if new → key < parent → key then
115         CAS(parent → left, old, new)
116     else
117         CAS(parent → right, old, new)
118 }
```

**Figure 9: Pseudocode for DELETE and some auxiliary routines.**

type T. Similarly, in describing the output type of a function, T* is a pointer to an instance of type T. If $x$ is a pointer, $x \rightarrow y$ refers to field $y$ of the record to which $x$ points.

The SEARCH($k$) routine traverses a branch of the tree from the root to a leaf, towards the key $k$. It behaves exactly as in a sequential implementation of a leaf-oriented BST, choosing which direction to go at each node by comparing the key stored there to $k$, continuing until it reaches a leaf. The FIND($k$) routine simply checks whether the leaf $l$ returned by SEARCH contains the key $k$. The INSERT($k$) and DELETE($k$) routines also call SEARCH to find the location in the tree where they should apply their update. In addition to the leaf $l$ reached, SEARCH returns some auxiliary information that is used by INSERT and DELETE: SEARCH returns the node $p$ that it saw as the parent of $l$, and the node $gp$ that it saw as the parent of $p$, and the values it read from the *state* and *info* fields of $p$ and $gp$. (Note, however, that $p$ and $gp$ may not be the parent and grandparent of $l$ when SEARCH terminates, due to concurrent updates changing the tree.)

Both INSERT and DELETE make repeated attempts until they succeed. INSERT first performs a SEARCH for the leaf that it must replace. If this leaf already contains the key to be inserted, INSERT returns FALSE (line 50), since the dictionary is not allowed to contain multiple copies of a key. If INSERT finds that some other operation has already flagged or marked the parent, it helps that operation complete (line 51) and then starts over with a new attempt. Otherwise, it creates the two new Leaf nodes and a new Internal node to be added to the tree, as well as a new IInfo record containing the necessary information. It then tries to flag the parent with an iflag CAS (line 56). If this fails, line 61 of INSERT helps the operation that has flagged or marked the parent, if any, and begins a new attempt. If the iflag CAS succeeds, the rest of the insertion is done by HELPINSERT, which simply attempts the ichild CAS (line 66) and the iunflag CAS (line 67), using the information stored in the IInfo record. The ichild CAS is actually carried out by CAS-CHILD, which determines whether to change the left or right child of the parent, depending on the key values, so the actual ichild CAS is on either line 115 or 117. After calling HELPINSERT, INSERT returns TRUE (line 59).

The structure of DELETE is very similar to the structure of INSERT. It first calls SEARCH to find the leaf to be deleted (and its parent and grandparent). If it fails to find the key, DELETE returns FALSE (line 76). If DELETE finds that some other operation has already flagged or marked the parent or grandparent, it helps that operation complete (line 77 and 78) and then begins over with a new attempt. Otherwise, it creates a new DInfo record containing the necessary information and attempts to flag the grandparent with a dflag CAS (line 81). If this fails, the DELETE helps the operation that has flagged or marked the grandparent (line 85), if any, and then begins a new attempt. If the dflag CAS is successful, the remainder of the deletion is carried out by HELPDELETE. However, it is possible that the attempted deletion will fail to complete even after the grandparent is flagged (if some other operation has changed the parent's *state* and *info* fields before it can be marked), so HELPDELETE returns a boolean value that describes whether the deletion was successfully completed. If it is successful, DELETE returns TRUE (line 83); otherwise, it tries again.

HELPDELETE first attempts to mark the parent of the leaf to be deleted with a mark CAS (line 91). If the mark CAS is successful, the remainder of the deletion is carried out by HELPMARKED, which simply performs the dchild CAS (line 105) and the dunflag CAS (line 106) using information stored in the DInfo record. However, if the mark CAS is unsuccessful, then HELPDELETE helps the operation that flagged the parent node and performs a backtrack CAS (line 98) to unflag the grandparent.

As mentioned in Section 4.1, each time a node is flagged, its *info* field contains a pointer to a new Info record, so it will always be different from any value previously stored there. When the *state* field is subsequently changed back to CLEAN by a dunflag, iunflag or backtrack CAS, we leave the pointer to the Info record in the *info* field so that the CAS object that contains these two fields has a value that is different from anything that was previously stored there. (If this causes complications for automatic garbage collection because of cycles of pointers between tree nodes and Info records, clean *update* fields could instead have a counter attached to them to serve the same purpose. Alternatively, pointers from the Info records to the nodes of the tree can be set to $\perp$ when the Info record is no longer needed, thus breaking cycles. This would complicate the pseudocode, because one would have to check that the Info record's pointer fields are non-$\perp$ before using them.)

# 5. CORRECTNESS

We provide only an outline of the proof of correctness. The full (and lengthy) proof appears in [5].

It is fairly straightforward to prove that calls to various subroutines satisfy their simple preconditions, which are given in the pseudocode. (These preconditions are mostly required to ensure that when we access a field using $x \rightarrow y$, $x$ is a non-$\perp$ pointer.) They are proved together with some very simple data structure invariants, for example, the keys of a node and its two children are in the right order, and the tree always keeps the nodes with dummy infinity keys at the top, as shown in Figure 6. These facts are sufficient to prove that every terminating SEARCH satisfies its postconditions.

A large part of the proof is devoted to showing that the CAS steps proceed in an orderly way. For example, we show that a successful mark CAS is performed on the correct node before the corresponding dchild CAS is performed. The sequence of changes that a node can go through are illustrated in Figure 4. The labels in the boxes show the values of the *state* field of the node. Each of the transitions corresponds to a successful CAS step on the node, except for the italicized *mark CAS*, which denotes a mark CAS on the appropriate *child* of the node. The italicized mark CAS changes the *state* field of the child from CLEAN to MARK. There are three circuits in Figure 4 starting from the CLEAN state. Each time a node travels once around one of these circuits, a new Info record is created for the flag CAS that begins the traversal. All subsequent CAS steps in that traversal of the circuit use information from that Info record. (We say that all of these CAS steps *belong to* the Info record.) A traversal of the three-step circuit on the right, which we call the *insertion circuit*, corresponds to successfully completing an insertion. A traversal of the four-step circuit on the left corresponds to successfully completing a deletion. A traversal of the two-step circuit on the left corresponds to an unsuccessful attempt of a deletion in which the mark CAS fails. We call these the *deletion circuits*.

It is fairly easy to see, by examining each of the CAS steps

in the code, that the *state* field of a node $v$ can change from CLEAN to DFLAG or IFLAG and then back to CLEAN again or from CLEAN to MARK (in which case it can never change again). Moreover, it can be shown that, each time the *state* and *info* fields of a node are changed, the pair of fields have a value they have never had before: When the *state* changes from CLEAN to DFLAG or IFLAG, the *info* field points to a newly created Info record. When the *state* changes back to CLEAN, the pointer in the *info* field is not changed (and that pointer has never previously been stored alongside a CLEAN *state*). If the *state* changes to MARK, the *info* field stores the corresponding DInfo pointer, which has never previously been stored alongside a MARK *state*.

To prove that Figure 4 reflects *all* possible changes to the tree requires more work. A traversal of the insertion circuit is initiated by an iflag CAS on a node $v$'s *update* field. This iflag CAS belongs to some IInfo record $f$. We prove there is at most one successful ichild CAS belonging to $f$. This requires a more technically involved argument to prove that, whenever a child pointer is changed, it points to a node to which it never previously pointed. We also show that, if there is an iunflag CAS (belonging to $f$) which resets the state of $v$ to CLEAN, it can happen only after the unique successful ichild CAS belonging to $f$ has been performed.

Each traversal of a deletion circuit begins with a dflag CAS on a node $v$'s *update* field. This dflag CAS belongs to some DInfo record $f$. We prove that only the first mark CAS belonging to $f$ succeeds (using the fact that the *update* field of $v$ is never changed to a value it had previously). A process does a backtrack CAS belonging to $f$ only after performing an unsuccessful mark CAS belonging to $f$ and observing that every prior mark CAS belonging to $f$ also failed (line 92). Hence, if some mark CAS belonging to $f$ succeeds, no process will perform a backtrack CAS belonging to $f$. This is why a backtrack CAS can only occur from one place in Figure 4. Thus, once the mark CAS occurs, the deletion will not have to retry. We also prove that only the first dchild CAS belonging to $f$ succeeds and that a dchild CAS belonging to $f$ succeeds only after the mark CAS belonging to $f$. Finally, a dunflag CAS belonging to $f$ can succeed only after a successful dchild CAS belonging to $f$.

All of this information about the ordering of CAS steps essentially tells us that the *state* field is correctly acting as a lock for the child pointers of the node. This allows us to prove that the effects of the child CAS steps are exactly as shown in Figure 1 and 2: an ichild CAS replaces a leaf by three new nodes that have never been in the tree before, and a dchild CAS replaces a node by one of its children (as part of a deletion of the node's other child, which is a leaf).

Using this information, we prove some invariants about the tree. Because a dchild CAS happens only after the corresponding mark CAS, an internal node is removed from the tree only after it is marked. Thus, every unmarked node is still in the tree. Because we know the effects of ichild and dchild CAS steps, we can prove that no node ever acquires a new ancestor in the tree after it is first added to the tree. (However, it may lose ancestors that are spliced out of the tree by dchild CAS steps and it may gain new descendants.) This allows us to prove a useful lemma: if a node $x$ is on the search path for key $k$ at some time, then $x$ remains on the search path for $k$ as long as $x$ is in the tree. (This is because $x$ can never obtain a new ancestor that could redirect the search path for $k$ in a different direction, away from $x$.)

Finally, we can define the linearization points for all FIND, INSERT and DELETE operations. For convenience, we also define linearization points for calls to SEARCH. While a process is performing SEARCH($k$), a node to which it is pointing may get removed from the tree. However, we show that each node the SEARCH visits *was* in the tree, on the search path for $k$, at some time during the SEARCH. We prove this by induction. It is true for the root node (which never changes). Suppose the SEARCH advances from one node $x$ to its child $y$. Either $y$ was already a child of $x$ when $x$ was on the search path for $k$ (in which case $y$ was also on the search path for $k$ at that time), or $y$ became a child at some later time. In the latter case, $x$ was flagged when $y$ became its child. At that time, $x$ must have been unmarked and, therefore, in the tree, on the search path for $k$. So $y$ was on the search path for $k$ just after it became the child of $x$. Thus, if SEARCH($k$) reaches a leaf and terminates, we can linearize it at a point when that leaf was on the search path for $k$.

We linearize each FIND operation at the same point as the SEARCH that it calls. The INSERT and DELETE operations that return FALSE are linearized at the same point as the SEARCH they perform. We prove that every INSERT and DELETE operation which returns TRUE must have a successful child CAS and we linearize the operation at that child CAS, since that is when the tree changes to reflect the update. Using this linearization, we show each operation returns the same result as it would if the operations were done sequentially in their linearization order and the keys in the leaves of the tree are always the contents of the dictionary.

The last part of the proof shows the implementation is non-blocking. To derive a contradiction, assume there is an execution where some set of operations continue taking steps forever, but no operation completes. We argue that no iflag, mark, child or unflag CAS steps can succeed, because then an operation would terminate. Thus, eventually, only dflag and backtrack CAS steps succeed and the tree stabilizes. We argue that processes must continue performing CAS steps, and (eventually) they fail only because of other successful CAS steps. Hence, there must be infinitely many successful dflag and backtrack CAS steps. After a successful dflag CAS on a node $v$, a backtrack CAS is performed on $v$ only if the mark CAS on $v$'s child fails. Thus, the deletion operating on a lowest node in the tree cannot backtrack, and therefore terminates, contradicting the assumption.

# 6. FUTURE WORK

Theoretical analysis and experimental work is needed to optimize our implementation and compare its performance to other dictionary implementations. There are results (in the sequential setting) proving that the expected time for operations on randomly constructed BSTs is logarithmic in the number of keys [19]. Such bounds for random *concurrent* updates are not as well-studied. In sequential systems, there are many techniques for maintaining a balanced BST that guarantee logarithmic height. One important goal is to extend our implementation to provide a similar guarantee, possibly by adapting some techniques for balancing lock-based concurrent BSTs, mentioned in Section 2.

We can also study ways to improve the amortized step complexity per operation. For example, after a process helps another operation to complete, it restarts its own operation from scratch. There may be more efficient ways to resume the operation by adding parent pointers to marked nodes us-

ing a strategy similar to the one used for linked lists [6], but this could add significant complications to the algorithm.

An adversarial scheduler can prevent a FIND from completing in the following run. Starting from an empty tree, one process inserts keys 1, 2 and 3 and then starts a FIND(2) that reaches the internal node with key 2. A second process then deletes 1, re-inserts 1, deletes 3 and re-inserts 3. Then, the first process advances two steps down the tree, again reaching an internal node with key 2. This can be repeated *ad infinitum*. A natural question is whether FIND can be made wait-free without a significant reduction in efficiency.

Effective management of memory is important for achieving reasonable space bounds. Hazard pointers [20] may be applicable to a slightly modified version of our implementation, where a SEARCH helps DELETE operations to perform their dchild CAS steps to remove from the tree marked nodes that the SEARCH encounters. More specifically, retirement of tree nodes and Info records could be performed when an unflag (or backtrack) CAS takes place. SEARCH would maintain a hazard pointer to each of the nodes pointed to by $gp$, $p$, $l$ and $l$'s sibling, as it traverses its search path. Each time an operation $O$ helps another operation $O'$, $O$ first ensures that hazard pointers are set to point to the Info record $f$ of $O'$, and to the nodes pointed to by $f.gp$, $f.p$, $f.l$ and $f.l$'s sibling. This may require storing more information in Info records. For example, it might be helpful to store an additional bit indicating whether the Info record is retired or not. This bit can be updated to TRUE with an additional CAS immediately after an unflag or backtrack CAS. The implementation of hazard pointers ensures that memory is not de-allocated as long as one or more hazard pointers point to it, even if it has been retired. Other techniques for garbage collection may also be applicable.

Many sequential tree-based data structures lack efficient non-blocking implementations in which non-interfering operations can be applied concurrently, using standard primitives such as (single-word) CAS. The techniques introduced here may prove useful for these problems, too.

**Acknowledgements** We thank Michalis Alvanos and the anonymous reviewers for their comments. Much of this research was done while Eric Ruppert was visiting FORTH ICS. Financial support was provided by NSERC and the European Commission via SARC integrated project 027648 (FP6) and ENCORE STREP project 248647 (FP7).

# 7. REFERENCES

[1] G. Barnes. A method for implementing lock-free data structures. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[2] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. 17th ACM Symp. on Parallel Algorithms and Architectures*, pages 228–237, 2005.

[3] J. Boyar, R. Fagerberg, and K. S. Larsen. Amortization results for chromatic search trees, with an application to priority queues. *Journal of Computer and System Sciences*, 55(3):504–521, 1997.

[4] P. Chuong, F. Ellen, and V. Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proc. 22nd ACM Symp. on Parallel Algorithms and Architectures*, 2010.

[5] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. Technical Report CSE-2010-04, York University, 2010.

[6] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proc. 23rd ACM Symp. on Principles of Distributed Computing*, pages 50–59, 2004.

[7] K. Fraser and T. L. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):article 5, May 2007.

[8] K. A. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, Dec. 2003. Available as Univ. of Cambridge Tech. Rep. UCAM-CL-TR-579.

[9] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symp. on Foundations of Comp. Sci.*, pages 8–21, 1978.

[10] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. 15th International Conference on Distributed Computing*, volume 2180 of *LNCS*, pages 300–314, 2001.

[11] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proc. 2nd ACM Symp. on Principles and Practice of Par. Programming*, 1990.

[12] M. Herlihy. Wait-free synchronization. *ACM Trans. on Prog. Languages and Systems*, 13(1):124–149, 1991.

[13] M. Herlihy. A methodology for implementing highly concurrent objects. *ACM Trans. on Prog. Languages and Systems*, 15(5):745–770, 1993.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[15] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proc. 22nd ACM Symp. on Principles of Distributed Computing*, 2003.

[16] T. N. Hibbard. Some combinatorial properties of certain trees with applications to searching and sorting. *J. ACM*, 9(1):13–28, 1962.

[17] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980.

[18] D. Lea. *Java Concurrent Skip List Map*. Available from http://www.java2s.com/Code/Java/Collections-Data-Structure/ConcurrentSkipListMap.htm.

[19] H. H. Mahmoud. *Evolution of random search trees*. Wiley-Interscience, 1992.

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

[21] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proc. 14th ACM Symp. on Parallel Algorithms and Architectures*, 2002.

[22] O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica*, 33(6):547–557, 1996.

[23] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *Proc. 19th ACM Symp. on Applied Computing*, pages 1438–1445, 2004.

[24] J. D. Valois. *Lock-free data structures*. PhD thesis, Computer Science Department, Rensselaer Polytechnic Institute, 1995.

[25] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 214–222, 1995.