Non-blocking Binary Search Trees

Faith Ellen, Panagiota Fatourou, Eric Ruppert and Franck van Breugel

Technical Report CSE-2010-04

May 2010

Department of Computer Science and Engineering
4700 Keele Street, Toronto, Ontario M3J 1P3 Canada

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

May 2010

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.

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, 5, 9, 14, 20, 22, 24]. The binary search tree (BST) [15] 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 [17], 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. A proof of correctness is given in Section 5. 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.

2 Related Work

There are implementations of BSTs using locks that support concurrent accesses. For example, Guibas and Sedgewick [8] gave a balanced BST by uncoupling rebalancing operations from updates, as did Kung and

Lehman [16], who also proved their implementation correct. Nurmi and Soisalon-Soininen [21] 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 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 [23, 24] 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 [7], 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 [6, 7, 10]. 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

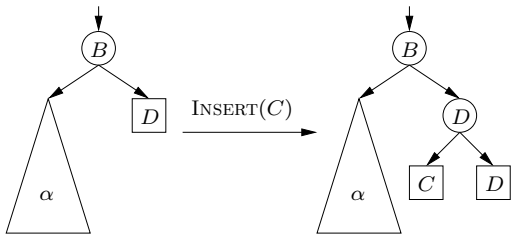

Figure 1: Insertion in a leaf-oriented BST in a single-process system.

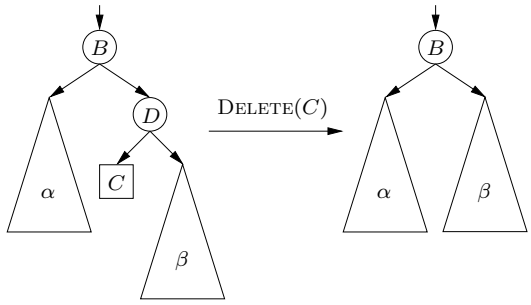

Figure 2: Deletion in a leaf-oriented BST in a single-process system.

each key, if this is required by the dictionary application.

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 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 [9] 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

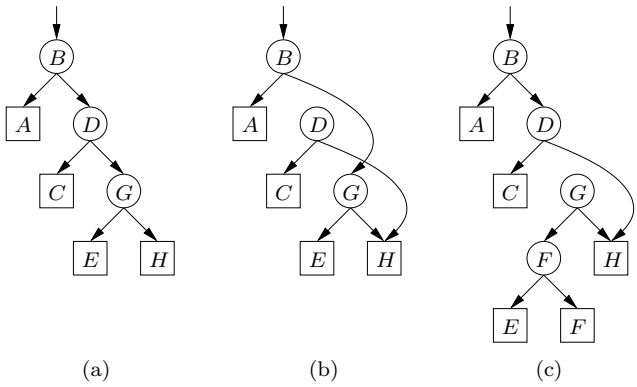

Figure 3: Problems can occur if updates only CAS one child pointer.

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 [5], 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

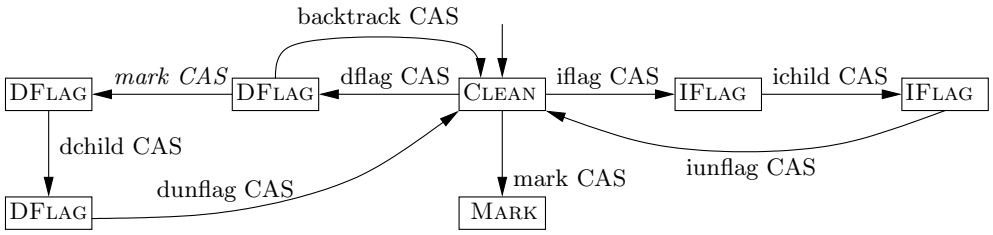

Figure 4: The effects of successful CAS operations.

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 a *backtrack* CAS to distinguish it from a dunflag CAS. (See Figure 4 for a summary of the types of CAS steps. In the figure, the italicized mark CAS is performed on the child of the node whose state is DFLAG.)

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

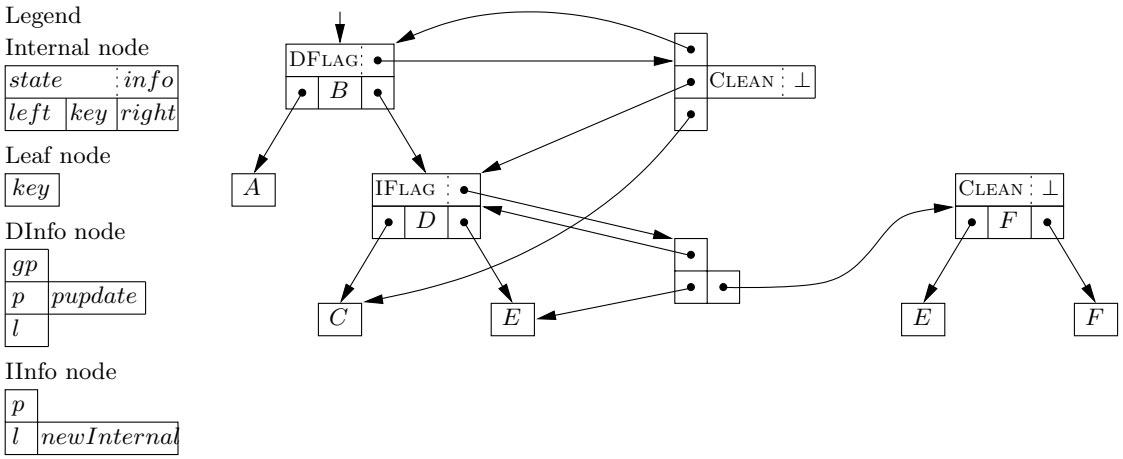

Figure 5: An example of the data structure. A DELETE(E) and an INSERT(F) are in progress. Fields separated by dotted lines are stored in a single word.

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*

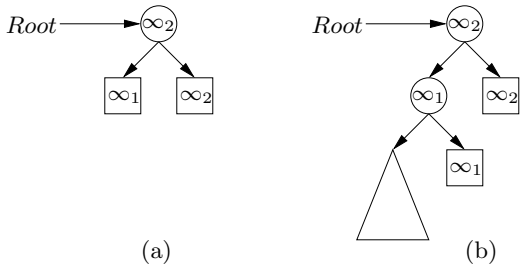

Figure 6: Trees showing leaves with dummy keys when the dictionary is (a) empty and (b) non-empty.

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 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 ∞_1) and initialize the tree so that it contains two dummy keys ∞_1 and ∞_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 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, $x \to y$ 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.)

```
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.

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

```
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.

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 begin with an outline of the proof of correctness, followed by the lengthy detailed proof. In the following proof, we use a fairly standard definition of an execution. A *configuration* is an instantaneous snapshot of the system describing the state of all local and shared variables as well as the programme counter and invocation stack of each process. A *step* can be either a shared-memory access by a process (including the result of the access) or a local step that simply updates a process's own local variables. For local steps, a step corresponds to executing one line of code. In particular, an invocation of a subroutine and the return from a subroutine are each considered to be a step. We assume each step is atomic, so an *execution* consists of an alternating sequence of configurations and steps, starting with the initial configuration where the shared memory is initialized with three nodes as shown in Figure 6(a). An execution is legal if every process follows its algorithm in the subsequence consisting of the steps that it performs, and if every shared object behaves according to its sequential specification in the subsequence of steps that access it.

5.1 Outline of Correctness Proof

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.

We begin by proving some basic invariants and simple pre- and postconditions in Section 5.2. Next, we prove some basic facts about how *update* fields of internal nodes are changed by various types of CAS steps in Section 5.3. The proofs in these two sections are relatively straightforward. (The reader may want to read only the statements of the lemmas in those sections when first reading this proof.) The more interesting proofs begin in Section 5.4, which prove that CAS steps proceed in an orderly way. (Essentially, we show that they occur as shown in Figure 4.) Section 5.4 concludes with Corollary 15 and 16, which say the effects of child CAS steps are exactly as shown in Figure 1 and 2. Next, in Section 5.5, we establish some invariants of the tree. In Section 5.6 we use these tree properties to define the linearization points for Find operations and update operations that return False. Each update operation that returns True is linearized at its successful child CAS step. We prove that each operation returns the same result as it would if the operations were done in their linearization order. Finally, in Section 5.7, we prove the algorithm is non-blocking.

5.2 Basic Invariants and Preconditions

We now begin the detailed proof by first proving some fairly simple properties. We first observe from the code that some fields of objects never change.

Observation 1 *The key field of a Node never changes. No field of an Info record ever changes. The Root pointer never changes.*

Proving that calls to the various subroutines satisfy their preconditions is a bit tricky: many invocations (indirectly) use results obtained from earlier calls to the Search routine. Thus, we must establish that each call to Search satisfies its postconditions. However, proving these postconditions requires some basic invariants about the data structure. The proofs of those invariants, in turn, are dependent on how the data structure is updated. In particular, they depend on knowing that the routines satisfy their preconditions. Thus, we need to combine, in an inductive argument, the proof that preconditions are satisfied with the proofs of the data structure invariants and of Search's postconditions. Thus, the following lemma serves to bootstrap the rest of the proof of correctness. The first five items in the lemma describe subroutine preconditions. The next two involve the postconditions of Search. The remaining items are some simple data structure invariants.

Lemma 2 *The following are invariants of the algorithm.*

1. *Each call to* HelpInsert *satisfies its preconditions.*
2. *Each call to* HelpDelete *satisfies its preconditions.*
3. *Each call to* HelpMarked *satisfies its preconditions.*
4. *Each call to* Help *satisfies its preconditions.*
5. *Each call to* CAS-Child *satisfies its preconditions.*
6. *Any* Search *that has executed line 25 has a non-$\perp$ value in its local variable l.*

7. *Each call to* SEARCH *that terminates satisfies its postconditions.*

8. *For each internal node x, neither of x's child pointers are $\perp$. Furthermore, x's left child has a key that is smaller than x's key and x's right child has a key that is greater than or equal to x's key.*

9. *The info field of each internal node "matches" its state field. More precisely:*

   (a) *if $v.state$ is* IFLAG*, then $v.info$ points to an IInfo record, and*

   (b) *if $v.state$ is* DFLAG *or* MARK*, then $v.info$ points to a DInfo record.*

10. *The top part of the tree is always as shown in Figure 6. More precisely:*

    (a) *$Root \rightarrow left \rightarrow key = \infty_1$, and*

    (b) *if $Root \rightarrow left$ points to an internal node, then $Root \rightarrow left \rightarrow right$ points to a leaf with key ∞_1.*

11. *For any IInfo record f, $f.p$ and $f.newInternal$ are pointers to internal nodes, and $f.l$ is a pointer to a leaf node.*

12. *For any DInfo record f, $f.gp$ and $f.p$ are pointers to internal nodes, $f.l$ is a pointer to a leaf node and $f.pupdate$ is a value that has previously appeared in $f.p \rightarrow update$.*

Proof: Claim 8 and 10 are true initially, according to the initialization of the data structure. All other claims are vacuously true for an execution of 0 steps. Assume that the invariants hold throughout some prefix α of the execution. Let $\alpha' = \alpha \cdot s \cdot C$ where s is a single step and C is the configuration that results from performing s after the final configuration of α. We show that each of the 12 parts of the lemma hold throughout the prefix α'.

1. We show that if s is an invocation of HELPINSERT (either on line 58 or 109), its argument is a pointer to an IInfo record.

   If s is an execution of line 58, its argument is a pointer to a newly created IInfo record.

   If s is an execution of line 109 in the HELP routine, the precondition of HELP (part 4 of the induction hypothesis) ensures that the argument of HELPINSERT was taken from the *update.info* field of some internal node. Furthermore, the *state* of the update field is IFLAG, so the claim follows from part 9 of the induction hypothesis.

2. We show that if s is an invocation of HELPDELETE (either on line 83 or 111), its argument is a pointer to a DInfo record. (This proof is similar to the preceding paragraph.)

   If s is an execution of line 83, its argument is a pointer to a newly created DInfo record.

   If s is an execution of line 111 in the HELP routine, the precondition of HELP (part 4 of the induction hypothesis) ensures that the argument of HELPDELETE was taken from the *update.info* field of some internal node. Furthermore, the *state* of the update field is DFLAG, so the claim follows from part 9 of the induction hypothesis.

3. We show that if s is an invocation of HELPMARKED (either on line 93 or 110), its argument is a pointer to a DInfo record.

   If s is an execution of line 93 in the HELPDELETE routine, the precondition of HELPDELETE (part 2 of the induction hypothesis) ensures that the argument of HELPMARKED is a pointer to a DInfo record.

   If s is an execution of line 110 in the HELP routine, the precondition of HELP (part 4 of the induction hypothesis) ensures that the argument of HELPINSERT was taken from the *update.info* field of some internal node. Furthermore, the *state* of the update field is MARK, so the claim follows from part 9 of the induction hypothesis.

4. We show that if s is an invocation of HELP, its argument has been stored in the *update* field of an internal node. There are six places in the code that call the HELP routine. We consider each of them in turn.

   If s is an execution of line 51 of INSERT or line 78 of DELETE, its argument is the *pupdate* value returned by an earlier SEARCH. By part 7 of the induction hypothesis, that SEARCH satisfied its postconditions. In particular, *pupdate* was read in the *update* field of a node, by Postcondition (3).

   If s is an execution of line 61 of INSERT, it uses the value read from $p \rightarrow update$ at line 56.

   If s is an execution of line 77 of DELETE, its argument is the *gpupdate* value returned by an earlier SEARCH. By part 7 of the induction hypothesis, that SEARCH satisfied its postconditions. Furthermore, line 77 can be executed only if $l \rightarrow key = k \neq \infty_1$, so Postcondition (4c) ensures that *gpupdate* was read in the *update* field of a node.

   If s is an execution of line 85 of DELETE, it uses the value read from $gp \rightarrow update$ at line 81. (As argued in the previous paragraph, postcondition (4a) applies to the SEARCH on line 75, so gp points to an internal node and the access to $gp \rightarrow update$ makes sense.)

   If s is an execution of line 97 of HELPDELETE, it uses the value read from $op \rightarrow p \rightarrow update$ at line 91. (The access to $op \rightarrow p \rightarrow update$ makes sense because $op \rightarrow p$ is an internal node, by part 12 of the induction hypothesis.)

5. We show that if s is an invocation of CAS-CHILD (either on line 66 or 105), its first argument is a pointer to an internal node and its last argument is a pointer to a node.

   If s is an execution of line 66 of HELPINSERT, the precondition of HELPINSERT (which was satisfied, by part 1 of the induction hypothesis) ensures that *op* points to some IInfo record f. By part 11 of the induction hypothesis, the arguments to the CAS-CHILD, $f.p$ and $f.newInternal$ both point to internal nodes.

   If s is an execution of line 105 of HELPMARKED, the precondition of HELPMARKED (which was satisfied, by part 3 of the induction hypothesis) ensures that *op* points to some DInfo record f. By part 12 of the induction hypothesis, $f.p$ points to an internal node. By part 8 of the induction hypothesis, that internal node has two non-$\perp$ child pointers. Thus line 104 of HELPMARKED stores a non-$\perp$ pointer in the local variable *other*. Also, by part 12 of the induction hypothesis, $f.gp$ points to an internal node. Thus, the precondition for CAS-CHILD is satisfied when it is called in step s.

6. We check that if s is an execution of line 25 or 32, it maintains claim 6. (No other lines can affect this claim.) Line 25 sets l to *Root*, which is non-$\perp$, by Observation 1, so it preserves claim 6. By part 8 of the induction hypothesis, executing line 32 also maintains claim 6.

7. We must check that if a SEARCH terminates at s, it satisfies its postconditions. Let $\langle gp_{final}, p_{final}, l_{final}, pupdate_{final}, gpupdate_{final} \rangle$ be the final values of the local variables when the SEARCH terminates.

   By part 6 of the induction hypothesis, the exit condition of the SEARCH's while loop ensures that l_{final} points to a leaf. *Root* always points to an internal node, by Observation 1. Thus, the SEARCH must execute the loop at least once. At the beginning of each loop iteration that the SEARCH performs, l points to an internal node; otherwise the exit condition would have been satisfied. Thus, each iteration of the loop stores a pointer to an internal node in p. This establishes postcondition (1).

   Postcondition (2) is satisfied, because the last time the SEARCH executes line 32, the value l_{final} is read from the appropriate child of the node that p_{final} points to.

   Postcondition (3) is satisfied, because the last time the SEARCH executes line 31, the value $pupdate_{final}$ is read from $p_{final} \rightarrow update$.

   We now prove Postcondition (4). Assume $l_{final} \rightarrow key \neq \infty_1$. Since the root node contains key ∞_2 and $k < \infty_2$, the first time the SEARCH executes line 32, it changes l to point to the root's left child.

By part 10 of the induction hypothesis, that child's key is ∞_1. Since $l_{final} \rightarrow key \neq \infty_1$, the loop is executed at least twice.

In the second and all subsequent iterations of the loop, a pointer to an internal node is stored in the local variable gp (since a pointer to an internal node is stored in the local variable p in every iteration). So Postcondition (4a) is satisfied.

Postcondition (4b) is satisfied, because the second-last time the SEARCH executes line 32, the value p_{final} is read from the appropriate child of the node that gp_{final} points to.

Postcondition (4c) is satisfied, because the second-last time the SEARCH executes line 31, the value $gpupdate_{final}$ is read from $gp_{final} \rightarrow update$.

8. When a new internal node is created at line 54, its child pointers satisfy this claim. It remains to show that if s is an execution of line 115 or 117 of CAS-CHILD that succeeds in changing a child pointer, then s preserves claim 8. (No other line changes a child field of an internal node.) When the CAS-CHILD was invoked prior to s, it satisfied its preconditions by part 5 of the induction hypothesis. In particular, its *new* argument was non-$\perp$. Thus, the new value written by s is non-$\perp$. Furthermore, the test on line 114 ensures that the ordering property is maintained by s.

9. The only step that can write IFLAG in a node's *state* field is an iflag CAS (line 56), which also stores a pointer to a newly-created IInfo record in the node's *info* field. So part (a) is preserved by every step.

   Similarly, the only step that can write DFLAG in a node's *state* field is a dflag CAS (line 81), which also stores a pointer to a newly-created DInfo record in the node's *info* field. The only step that can write MARK in a node's *state* field is a mark CAS on line 91 of the HELPDELETE routine. According to part 2 of the inductive hypothesis, the argument of that call was a pointer to a DInfo record, and the mark CAS writes a pointer to that object in the node's *info* field. Thus part (b) is preserved by every step.

10. Since *Root* never changes and the *key* field of a node never changes (by Observation 1), it suffices to prove the following two things to show that s preserves this part of the invariant: s cannot change the field $Root \rightarrow left \rightarrow right$, and if s changes $Root \rightarrow left$, then s preserves (a) and (b). We prove these statements by considering two cases, corresponding to the two types of steps that can change child pointers.

    First, suppose s is an ichild CAS that changes $Root \rightarrow left$. Thus, s is inside a call to CAS-CHILD from line 66 of HELPINSERT. By part 1 of the induction hypothesis, the argument to that HELPINSERT is a pointer to some IInfo record f. Since s is a successful ichild CAS, s changes the root's left child pointer from $f.l$ to $f.newInternal$. By part 10 of the induction hypothesis, $f.l \rightarrow key = \infty_1$. Consider the time f was created (on line 55 of some call to INSERT(k)). At that time $f.newInternal \rightarrow key$ and $f.newInternal \rightarrow right \rightarrow key$ are both set to $\max(k, f.l \rightarrow key) = \max(k, \infty_1) = \infty_1$ (since $k \in$ Key). Furthermore, $f.newInternal \rightarrow right$ points to a leaf. Thus, after s, $Root \rightarrow left \rightarrow key = f.newInternal \rightarrow key = \infty_1$ and $Root \rightarrow left \rightarrow right$ is equal to $f.newInternal \rightarrow right$, which points to a leaf with key ∞_1. So, both (a) and (b) are preserved by s in this case.

    Now, suppose s is a dchild CAS that changes $Root \rightarrow left$. Thus, s is inside a call to CAS-CHILD from line 105 of HELPMARKED. By part 3 of the induction hypothesis, the argument to that HELPMARKED is a pointer to some DInfo record f. Consider the time f was created (on line 80 of some call to DELETE(k)). Since line 80 is executed, $f.l \rightarrow key = k \neq \infty_1$; otherwise the test in line 76 would have evaluated to TRUE. Since s is a successful dchild CAS on $Root \rightarrow left$, the value in $Root \rightarrow left$ just prior to s was $f.p$. By part 12 of the induction hypothesis, $f.p$ points to an interal node. Therefore, by part 10 of the induction hypothesis, $f.p \rightarrow right$ points to a leaf with key ∞_1. But $f.l \rightarrow key \neq \infty_1$, so $f.p \rightarrow right \neq f.l$. Thus, just before the dchild CAS, *other* is set to $f.p \rightarrow right$ on line 104. This means that s changes $Root \rightarrow left$ to $f.p \rightarrow right$, which we have already shown to be a pointer to a leaf with key ∞_1. Thus, just after s, (a) is satisfied, and so is (b), since $Root \rightarrow left$ now points to a leaf.

Next, to derive a contradiction, we assume s is an ichild CAS that changes the field $Root \rightarrow left \rightarrow right$. Thus, s is inside a call to CAS-CHILD from line 66 of HELPINSERT. By part 1 of the induction hypothesis, the argument to that HELPINSERT is a pointer to some IInfo record f. Step s changes $f.p \rightarrow right$ from $f.l$ to $f.newInternal$. Thus, just before s, $Root \rightarrow left = f.p$ and $Root \rightarrow left \rightarrow right = f.l$. Thus, by part 10 of the induction hypothesis, $f.p \rightarrow key$ and $f.l \rightarrow key$ are both ∞_1. Consider the time when f was created (on line 55 of some call to INSERT(k)). The values in $f.p$ and $f.l$ were obtained from a call to SEARCH on line 49. By part 7 of the induction hypothesis, that SEARCH satisfied its postconditions. In particular, postcondition (2) implies that $f.p \rightarrow left$ was equal to $f.l$ during the search (since $k < \infty_1 = f.p \rightarrow key$). This contradicts part 8 of the induction hypothesis since $f.l \rightarrow key = f.p \rightarrow key$. Thus, s cannot be an ichild CAS that changes the field $Root \rightarrow left \rightarrow right$.

Finally, to derive a contradiction, we assume s is a dchild CAS that changes the field $Root \rightarrow left \rightarrow right$. Thus, s is inside a call to CAS-CHILD from line 105 of HELPMARKED. By part 3 of the induction hypothesis, the argument to HELPMARKED is a pointer to some DInfo record f. Since step s is a successful CAS, $Root \rightarrow left \rightarrow right$ must be equal to $f.p$ just prior to s. However, $f.p$ points to an internal node, by part 12 of the induction hypothesis, and $Root \rightarrow left \rightarrow right$ points to a leaf, by part 10 of the induction hypothesis. This contradiction means that s cannot be a dchild CAS that changes the field $Root \rightarrow left \rightarrow right$.

11. Since fields of IInfo records are never changed, by Observation 1, we need only check that the claim is preserved if s creates a new IInfo record. This only occurs at line 55 of the INSERT routine. When an IInfo record f is created, $f.newInternal$ points to a newly created internal node, and $f.p$ and $f.l$ are values obtained from the SEARCH on line 49. By part 7 of the induction hypothesis, that SEARCH satisfied its postconditions. In particular, $f.p$ points to an internal node and $f.l$ points to a leaf node.

12. Since fields of DInfo records are never changed, by Observation 1, we need only check that the claim is preserved if s creates a new DInfo record. This only occurs at line 80 of the DELETE routine. When a DInfo record f is created, its fields are filled with values obtained from the SEARCH on line 75. By part 7 of the induction hypothesis, that SEARCH satisfied its postconditions. Furthermore, the test on line 76 must have failed, so $f.l \rightarrow key = k \neq \infty_1$ and Postcondition (4) is applicable. It follows from the postconditions of SEARCH that $f.gp$ and $f.p$ point to internal nodes, $f.l$ points to a leaf node and $f.pupdate$ is a value that has previously been read from $f.p \rightarrow update$.

■

We remark that the pre- and post-conditions described in the preceding lemma are sufficient to guarantee that whenever a line of the pseudocode accesses $X \rightarrow Y$, the pointer X points to an object of the appropriate type and is not $\perp$, so the access makes sense.

5.3 Behaviour of CAS Steps on *update* Fields

Next, we examine the order in which successful CAS steps can occur, and their effects on the *update* field of a node. As mentioned in Section 5.1, we can view each node v in the tree as an automaton and we wish to show that the sequences of changes that it can go through are as shown in Figure 4. The main goal of the next few lemmas is to show that this automaton accurately captures all the successful CAS steps that can occur. The first one is easily proved by observing the code.

Lemma 3 *The following statements are true for each internal node v.*

1. *When v is created, $v.update = \langle$CLEAN$, \perp\rangle$.*
2. *Flag and mark* CAS *steps on v's update field succeed only if v's state is* CLEAN.
3. *Iunflag* CAS *steps on v's update field succeed only if v's state is* IFLAG.

4. *Dunflag and backtrack* CAS *steps on v's update field succeed only if v's state is* DFlag.

5. *Once v is marked, its update field never changes.*

Proof: We prove each of the five statements in turn.

1. The internal node that is initially in the tree starts with a Clean *state*. Internal nodes are created only on line 54, which sets the *state* to Clean.

2. Prior to an iflag CAS that uses *pupdate* as the old value, it is checked that *pupdate.state* is Clean. Prior to a dflag CAS that uses *gpupdate* as the old value, it is checked that *gpupdate.state* is Clean. Now consider a successful mark CAS that is performed by a call to HelpDelete. Lemma 2(2) says that the argument to HelpDelete was a DInfo record f. In the execution of Delete that created f, the test on line 78 failed, meaning that *f.pupdate.state* was Clean. The mark CAS uses *f.pupdate* as the old value for the CAS, so it will succeed only if the node's *state* field was Clean.

3. An iunflag CAS specifies that the old value of the *state* field must be IFlag.

4. A dunflag CAS or backtrack CAS each specifies that the old value of the *state* field must be DFlag.

5. Follows from the previous three parts: none of the CAS steps on *v.update* can succeed if *v.state* = Mark.

■

We now prove that no ABA problem can occur for *update* fields of internal nodes. We use the notation $\&x$ to denote a pointer to the object x.

Lemma 4 *For each internal node v, no* CAS *ever changes v.update to a value that was previously stored there.*

Proof: When v is created, *v.update* is set to $\langle \text{Clean}, \perp \rangle$, by Lemma 3(1). We consider each of the types of CAS steps that can change *v.update* and argue that each one writes a value in *v.update* that has never appeared there before. A successful mark CAS is the first CAS that changes *v.state* to Mark, by Lemma 3(5). Each successful flag CAS on *v.update* sets its *v.info* subfield to point to a newly created Info record, so that address could never have appeared in *v.info* before. An iunflag CAS changes the update field from $\langle \text{IFlag}, \&f \rangle$ to $\langle \text{Clean}, \&f \rangle$ (where f is some Info record). Since $\&f$ never appeared in *v.info* prior to the iflag CAS that set *v.update* to $\langle \text{IFlag}, \&f \rangle$, *v.update* has never had the value $\langle \text{Clean}, \&f \rangle$ prior to this iunflag CAS. Identical reasoning applies to a dunflag or backtrack CAS that changes the state of *v.update* from $\langle \text{DFlag}, \&f \rangle$ to $\langle \text{Clean}, \&f \rangle$. ■

Each successful flag CAS stores a pointer to a newly created Info record. Thus, no two successful flag CAS steps use pointers to the same Info record. If a successful flag CAS stores a pointer to an Info record f, we say that the flag CAS *belongs to* f. Mark, child, unflag and backtrack CAS steps all use information from some Info record. (More precisely, a pointer to this Info record is used as the argument of the invocation of HelpDelete, HelpInsert or HelpMarked that performs the CAS.) We say that each of these CAS steps also *belong to* the Info record. Notice that dflag, mark, dchild and dunflag CAS steps can belong only to DInfo records, and iflag, ichild and iunflag CAS steps can belong only to IInfo records. When an Insert or Delete creates an Info record, the values that it stores in several fields of the Info record are taken from the results of an invocation of Search. We also say that this invocation of Search *belongs to* the Info record.

Lemma 5 *Only the first mark* CAS *belonging to a DInfo record can succeed.*

Proof: Let f be any DInfo record. Now suppose some mark CAS, *mcas*, that belongs to f succeeds. We wish to show it is the first mark CAS that belongs to f. To derive a contradiction, assume there is some mark CAS *mcas'* that belongs to f and precedes *mcas*. Both mark CAS steps attempt to change $f.p \rightarrow update$ from $f.pupdate$ to $\langle \text{MARK}, \&f \rangle$. According to the postcondition of the SEARCH belonging to f, $f.pupdate$ was read from $f.p \rightarrow update$ during the SEARCH. This happens before f is created, and therefore before *mcas'*. When *mcas* occurs (after *mcas'*), $f.p \rightarrow update$ is still equal to $f.pupdate$; otherwise *mcas* would fail. Thus, by Lemma 4, $f.p \rightarrow update$ must be equal to $f.pupdate$ when *mcas'* occurs. Hence, *mcas'* succeeds and changes $f.p \rightarrow update$ to $\langle \text{MARK}, \&f \rangle$. This field never changes again after *mcas'*, by Lemma 3, so *mcas* must fail, contrary to the definition of *mcas*. ■

Lemma 6 *If a successful mark* CAS *belongs to a DInfo record f, then no backtrack* CAS *belongs to f.*

Proof: Let f be a DInfo record. Assume a successful mark CAS, *mcas*, belongs to f. Consider any invocation H of HELPDELETE whose argument is a pointer to f. By Lemma 5, its mark CAS either succeeds (if it is the first mark CAS belonging to f) or some earlier mark CAS belonging to f has succeeded in which case H sets its *result* to $\langle \text{MARK}, \&f \rangle$ (by Lemma 3). Either way, the test on line 92 of the HELPDELETE evaluates to TRUE and H will not attempt a backtrack CAS. ■

5.4 Behaviour of Child CAS Steps

We now focus on how child CAS steps update child pointers. The following lemma describes how the old value used in a child CAS is obtained.

Lemma 7 *Let ccas be a child* CAS *belonging to some Info record f and let R be the register it attempts to change. Let r be the last or second-last execution of line 32 in the* SEARCH *belonging to f, depending on whether ccas is an ichild* CAS *or a dchild* CAS*, respectively. Then, r reads R and the value read by r is the old value used for ccas.*

Proof: We consider two cases.

Case 1: *ccas* is an ichild CAS. We first note that r is well-defined: the call to SEARCH belonging to f performs at least one iteration of the loop, since *Root* always points to an internal node. The IInfo record f was created by an invocation INSERT(k) for some k. The value read by r is returned by the SEARCH and saved in $f.l$. This value is used as the old value for *ccas*, proving the second part of the claim.

Step r reads $f.l$ from one of the child fields of $f.p$ and *ccas* is a CAS on one of the child fields of $f.p$. To complete the proof of the first part of the claim, we must show that they both are on the *same* child field of that node. Let $pkey = f.p \rightarrow key$. Whether *ccas* is applied to the left or right child depends on whether the key of $f.newInternal \rightarrow key$ is smaller than *pkey* or not.

If r reads $f.p \rightarrow left$, then $k < pkey$ and so is $f.l \rightarrow key$ (by Lemma 2(8)). Thus $f.newInternal \rightarrow key = \max(k, f.l \rightarrow key) < pkey$ and *ccas* is applied to $f.p \rightarrow left$, according to the test on line 114.

On the other hand, if r reads $f.p \rightarrow right$, then $k \geq pkey$, so $f.newInternal \rightarrow key = \max(k, f.l \rightarrow key) \geq pkey$ and *ccas* is applied to $f.p \rightarrow right$, according to the test on line 114.

Case 2: *ccas* is a dchild CAS. The DInfo record f was created by some invocation DELETE(k), where $k = f.l \rightarrow key$. We first check that r is well-defined. As in Case 1, the SEARCH belonging to f executed at least one iteration of its loop. Furthermore, it returned a leaf with key $k \neq \infty_1$, so the SEARCH must have performed at least two iterations of the loop before reaching a leaf (by Lemma 2(10)). The value read by r is returned by the SEARCH and saved in $f.p$. This value is used as the old value for *ccas*, proving the second part of the claim.

Step r reads $f.p$ from one of the child fields of $f.gp$ and *ccas* is a CAS on one of the child fields of $f.gp$. We must show that they both are on the *same* child field of that node. The step *ccas* is applied to the left *child* field iff $f.l \rightarrow key = k < f.gp \rightarrow key$, which is true iff r reads the left *child* field. ■

Let $ccas_1, ccas_2, \ldots$ be the sequence of successful child CAS steps in the execution, in the order that they occur. Let f_i be the Info record to which $ccas_i$ belongs. (If $ccas_i$ is an ichild CAS, then f_i is necessarily

an IInfo record. If $ccas_i$ is a dchild CAS, then f_i is necessarily a DInfo record.) Let $fcas_i$ be the (unique) successful flag CAS that belongs to f_i. Let $ucas_i$ be the CAS that next changes the *update* field flagged by $fcas_i$ after $fcas_i$, if such a change exists. (If it exists, $ucas_i$ must be an unflag or backtrack CAS, by Lemma 3.)

For each successful child CAS step, we define R_i and r_i as in the preceding lemma: R_i is the register changed by $ccas_i$ and r_i is the last or second-last execution of line 32 in the SEARCH belonging to f_i, depending on whether $ccas_i$ is an ichild CAS or a dchild CAS, respectively. (As argued in the proof of the lemma, r_i is well-defined.) By definition of r_i, r_i reads the value $f_i.l$ in R_i if $ccas_i$ is an ichild CAS, or the value $f_i.p$ in R_i if $ccas_i$ is a dchild CAS. We have the following corollary of Lemma 7.

Corollary 8 *For all i, r_i reads R_i and the value read by r_i is the old value used for $ccas_i$.*

The next few lemmas prove that child CAS steps are done in an orderly way. Intuitively, we prove each child CAS comes between the corresponding flag and unflag CAS, and after the corresponding mark CAS in the case of dchild CAS steps. Furthermore, at most one successful child CAS belongs to each Info record, the child CAS steps are done in a way that avoids the ABA problem, and they maintain the invariant that the data structure is a full binary tree.

Lemma 9 *Each mark* CAS *that belongs to an Info record f is preceded by a successful dflag* CAS *that belongs to f.*

Proof: Let *mcas* be any mark CAS belonging to f. It is performed by an invocation of HELPDELETE. If HELPDELETE was called by line 83, then the dflag CAS belonging to f on line 81 succeeded. Otherwise, HELPDELETE was called by line 111 of HELP after a dflag CAS belonging to f wrote $\langle \text{DFLAG}, \&f \rangle$ in the *update* field of some node (according to the precondition of HELP). Thus, in both cases, the mark CAS comes after the successful dflag CAS belonging to f. ■

Lemma 10 *Each dchild* CAS *that belongs to an Info record f is preceded by a successful mark* CAS *that belongs to f.*

Proof: Let *ccas* be any dchild CAS belonging to f. It is performed by HELPMARKED. If HELPMARKED is called by line 93 of HELPDELETE, the test on line 92 ensures that $f.p \rightarrow update$ was equal to $\langle \text{MARK}, \&f \rangle$ after line 91. Otherwise, HELPMARKED is called by line 110. In this case, $\langle \text{MARK}, \&f \rangle$ has been read in the *update* field of some node, according to the precondition of HELP. Thus, in both cases, a successful mark CAS belonging to f has occurred before the invocation of HELPMARKED. ■

Lemma 11 *If a dchild* CAS *belongs to a DInfo record f, then no backtrack* CAS *belongs to f.*

Proof: If a dchild CAS belongs to f, then a successful mark CAS belongs to f, by Lemma 10. Thus, there is no backtrack CAS that belongs to f, by Lemma 6. ■

Lemma 12 *Each child* CAS *that belongs to an Info record f is preceded by a successful flag* CAS *that belongs to f. (In particular, for all i, $ccas_i$ occurs after $fcas_i$.)*

Proof: Let *ccas* be a child CAS belonging to f.

If *ccas* is an ichild CAS, then it is performed by line 66 of HELPINSERT. This routine is called either by line 58 of INSERT after successfully performing the iflag CAS belonging to f or by a process executing line 109 of HELP. In the latter case, $\langle \text{IFLAG}, \&f \rangle$ has appeared in a node's *update* field, by the preconditions of HELP. Either way, some process has performed the successful iflag CAS belonging to f prior to *ccas*.

If *ccas* is a dchild CAS, it follows from Lemma 9 and Lemma 10 that *ccas* is after the successful flag CAS belonging to f. ■

Lemma 13 *For all i, if $ccas_i$ is a dchild* CAS *then there is exactly one successful mark* CAS *belonging to f_i. This mark* CAS *occurs between $fcas_i$ and $ccas_i$.*

Proof: Lemma 10 says that a successful mark CAS belonging to f_i occurs before $ccas_i$. By Lemma 5, there is exactly one successful mark CAS that belongs to f_i. Lemma 9 says that the unique successful mark CAS belonging to f_i comes after $fcas_i$. ■

Lemma 14 (The Child CAS Lemma) *For all $i \geq 1$ such that $ccas_i$ exists, the following statements are true.*

1. *$ccas_i$ is the first successful child* CAS *on R_i after r_i. It is also the first successful child* CAS *belonging to f_i.*
2. *If $ucas_i$ exists, it is an unflag* CAS *that belongs to f_i, and $ccas_i$ does not occur after $ucas_i$.*
3. *If $ccas_i$ is an ichild* CAS*, then $ccas_i$ writes a pointer to the root of a full binary tree consisting of three new nodes that have never appeared in the data structure before (i.e., the three nodes have never been reachable from the Root by following child pointers before).*
4. *If $ccas_i$ is a dchild* CAS*, then just prior to $ccas_i$, R_i contains $f_i.p$. If $ccas_i$ is an ichild* CAS*, then just prior to $ccas_i$, R_i contains $f_i.l$.*
5. *If $ccas_i$ is a dchild* CAS*, then $f_i.p$'s child pointers do not change between the last execution of line 32 within the* SEARCH *belonging to f_i and $ccas_i$.*
6. *After $ccas_i$, the data structure is a full binary tree. (That is, every internal node that can be reached by following child pointers from Root has exactly two children, there are no cycles in the child pointers among these nodes, and there is at most one path of child pointers to any node from the root.)*
7. *$ccas_i$ writes a value into R_i that has never been stored there before.*

Proof: We prove the lemma using strong induction on i. Let k be a positive integer such that $ccas_k$ exists. Assume the claims are true for $ccas_i$ when $1 \leq i < k$. We prove the claims are true for $ccas_k$.

1. Let o_k be the old value used for $ccas_k$. By Corollary 8, o_k is the value read from R_k by r_k. To derive a contradiction, assume that $ccas_k$ is *not* the first successful child CAS on R_k after r_k. Then, let $ccas_j$ be the last successful child CAS on R_k between r_k and $ccas_k$. The child CAS $ccas_j$ writes a different value than o_k (by Claim 7 of the induction hypothesis). The value in R_k does not change between $ccas_j$ and $ccas_k$ (by definition of $ccas_j$), so R_k contains a value different from o_k when $ccas_k$ is performed. Since $ccas_k$ uses o_k as the old value, it will not succeed, which contradicts the definition of $ccas_k$. Thus, $ccas_k$ is the first successful child CAS on R_k after r_k.

   Now, to derive a contradiction, assume $ccas_k$ is not the first successful child CAS that belongs to f_k. Thus, there is some $j < k$ such that $ccas_j$ also belongs to f_k. By Lemma 12, $ccas_j$ occurs after $fcas_j$ and $ccas_k$ occurs after $fcas_k$. There is at most one flag CAS for each Info record created, so $fcas_j = fcas_k$. So, $ccas_j$ occurs after $fcas_k$ (by Lemma 12), which occurs after r_k. Thus, $ccas_k$ is not the first successful child CAS on R_k after r_k, contradicting the previous paragraph. Hence, $ccas_k$ must be the first child CAS that belongs to f_k.

2. Suppose that $ucas_k$ exists. Recall that it is the first CAS step that succeeds in changing R_k after $fcas_k$. Thus it changes R_k from $\langle \text{IFLAG}, \&f_k \rangle$ or $\langle \text{DFLAG}, \&f_k \rangle$ to something else. By Lemma 3, the only types of CAS steps that can do this are unflag CAS or backtrack CAS steps. Since the old value used by $ucas_k$ is $\langle \text{IFLAG}, \&f_k \rangle$ or $\langle \text{DFLAG}, \&f_k \rangle$, $ucas_k$ must belong to f_k.

   We first show that $ucas_k$ cannot be a backtrack CAS belonging to f_k. If $ccas_k$ is a ichild CAS, there cannot be a backtrack CAS that belongs to the IInfo record f_k. If $ccas_k$ is a dchild CAS, then it was preceded by a mark CAS belonging to f_k, by Lemma 10. Thus, there are no backtrack CAS steps belonging to f_k, by Lemma 6.

Thus, we know that $ucas_k$ is an unflag CAS that belongs to f_k. It remains to show that $ccas_k$ occurs before $ucas_k$. To derive a contradiction, assume that $ccas_k$ occurs after $ucas_k$. Let q be the process that performs $ucas_k$. We consider two cases.

If $ucas_k$ is an iunflag CAS, let $ccas$ be the ichild CAS performed by q just before $ucas_k$. Both $ccas$ and $ucas_k$ belong to f_k. Since q used a pointer to f_k in the old value for $ucas_k$, q's invocation of HelpInsert that performs $ccas$ and $ucas_k$ comes after $fcas_k$ (and therefore after r_k). The read r_k returns the value $f_k.l$ and this is the value stored in R_k at all times between r_k and $ccas_k$, by Claim 1, proved above. Thus, $ccas$ must be a successful child CAS on R_k, since it uses $f_k.l$ as its old value and it occurs between r_k and $ccas_k$. But this contradicts Claim 1 for $ccas_k$, proved above.

If $ucas_k$ is a dunflag CAS, the proof is very similar to the preceding paragraph. Let $ccas$ be the dchild CAS performed by q just before $ucas_k$. Both $ccas$ and $ucas_k$ belong to f_k. Since q used a pointer to f_k in the old value for $ucas_k$, q's invocation of HelpMarked that performs $ccas$ and $ucas_k$ comes after $fcas_k$ (and therefore after r_k). The read r_k returns the value $f_k.p$ and this is the value stored in R_k at all times between r_k and $ccas_k$, by Claim 1, proved above. Thus, $ccas$ must be a successful child CAS on R_k, since it uses $f_k.p$ as its old value and it occurs between r_k and $ccas_k$. But this contradicts Claim 1 for $ccas_k$, proved above.

In both cases, we obtained a contradiction. Thus, $ccas_k$ cannot occur after $ucas_k$.

3. If $ccas_k$ is an ichild CAS, it writes the pointer $f_k.newInternal$. That pointer points to a node x created by the Insert that performed $fcas_k$. When x is created, its child pointers point to two newly created leaf nodes, y and z. None of x, y or z can be reachable from the *Root* until some child CAS changes a child pointer to point to one of them. We consider all successful child CAS steps prior to $ccas_k$ and prove that none of them wrote pointers to x, y or z.

   An ichild CAS can only write a pointer to an internal node, so it cannot write a pointer to y or z. Furthermore, only an ichild CAS belonging to f_k can write a pointer to x, and there is no such ichild CAS prior to f_k, by Claim 1, proved above.

   It remains to show that no dchild CAS can write a pointer to x, y or z before $ccas_k$. To derive a contradiction, assume some dchild CAS writes a pointer to x, y or z prior to $ccas_k$. Consider the first such dchild CAS, $ccas_j$. The value written by $ccas_j$ is read from a child pointer of $f_j.p$ in line 104. Thus, $ccas_j$ could not have written a pointer to x because no child pointer points to x before $ccas_j$ (by definition of $ccas_j$). Thus $ccas_j$ writes a pointer to y or z. Without loss of generality, assume it writes a pointer to y. Then, the process that performed $ccas_j$ had previously read a pointer to y in a child field of the node that $f_j.p$ points to (on line 104). Thus, $f_j.p$ could only point to x since, prior to $ccas_j$, the only child pointer that points to y is in the node x. So, one of the child fields in the node that $f_j.gp$ points to contained a pointer to x before the end of the Search belonging to f_j, according to postcondition (4b) of the Search. This is impossible, since no child pointer points to x before $ccas_k$. This completes the proof of Claim 3.

4. A dchild CAS $ccas_k$ uses $f_k.p$ as the old value. An ichild CAS $ccas_k$ uses $f_k.l$ as the old value. Since the CAS is successful, R_k must have contained that value just before $ccas_k$.

5. Let x be the node that $f_k.p$ points to. Let r be the last execution of line 32 of the Search that belongs to f_k. To derive a contradiction, assume that for some $j < k$, $ccas_j$ is a successful child CAS that changes a child pointer in node x between r and $ccas_k$.

   By Lemma 10, x is marked before $ccas_k$. By Lemma 12 and Claim 2 of the induction hypothesis, x is flagged when $ccas_j$ occurs. Since marked nodes never become unmarked and a node cannot be both marked and flagged, $ccas_j$ occurs before the successful mark CAS belonging to f_k. Since x's *state* must be Clean just before that mark CAS occurs, there must be a successful unflag CAS on x between $ccas_j$ and the mark CAS. The old value used by the mark CAS was read at line 31 before r, but the unflag CAS changes the update field between r and the mark CAS. This contradicts the fact

that the mark CAS succeeds (because Lemma 4 implies that no ABA problem can occur in an *update* field).

6. If $k = 1$, the data structure is a full binary tree prior to $ccas_k$ because of the way it is initialized. If $k > 1$, the induction hypothesis implies that the data structure is a full binary tree prior to $ccas_k$ because no changes to it have occured between $ccas_{k-1}$ and $ccas_k$. We must show that $ccas_k$ preserves this property.

   If $ccas_k$ is an ichild CAS, it replaces a subtree by a full binary tree of three new nodes, by Claim 3 proved above, so the full binary tree property is preserved.

   Now suppose $ccas_k$ is a dchild CAS. Then $ccas_k$ changes a child pointer of the node that $f_k.gp$ points to from $f_k.p$ to the pointer *other* read (in line 104) from a child pointer of the node $f_k.p$ points to. Since the child pointers of the node that $f_k.p$ points to cannot change between line 104 and the $ccas_k$ (by Claim 5, proved above), it follows that just prior to the dchild CAS, *other* points to a child of the node $f_k.p$ points to. Thus, the dchild CAS replaces a subtree by a subtree of that subtree, and the full binary tree property is preserved.

7. If $ccas_k$ is an ichild CAS, it updates the child pointer to point to a new node that has never been in the data structure before, by Claim 3 proved above.

   Now consider the case where $ccas_k$ is a dchild CAS. Let x be the node whose child pointer is changed by $ccas_k$. Without loss of generality, assume x's left child pointer is changed by $ccas_k$. Suppose $ccas_k$ changes the left child of x from node z to node y. To derive a contradiction, assume that y was the left child of x at some earlier time.

   Just prior to $ccas_k$, y was a child of z, as argued in the proof of Claim 6. By Claim 6 of the induction hypothesis, $y \neq z$. Thus there is some $j < k$ such that $ccas_j$ caused y to stop being the left child of x. (This child CAS exists because y was a child of x but is no longer a child of x just before $ccas_k$.)

   We prove that, just after $ccas_j$, y is not a descendant of x. If $ccas_j$ is an ichild CAS, this follows from Claim 3 of the induction hypothesis. If $ccas_j$ is a dchild CAS, then by Claim 5 of the induction hypothesis, $ccas_j$ replaces a pointer to y by a pointer to y's child, so y is no longer a descendant of x (since y cannot be a descendant of its own child by Claim 6 of the induction hypothesis).

   Claims 3 and 5 imply that y can never become a descendant of x again between $ccas_j$ and $ccas_k$, contradicting the fact that y is a grandchild of x just before $ccas_k$.

This completes the proof of the lemma. ■

The following two corollaries of Lemma 14 describe the effects of a successful ichild or dchild CAS. Essentially, they say that the effect of a child CAS is exactly as shown in Figure 1 and 2.

Corollary 15 *For all i, if $ccas_i$ is an ichild* CAS*, it changes a child pointer of $f_i.p$, replacing $f_i.l$, which is a pointer to a leaf, by a pointer to the root of a full binary tree consisting of three new nodes that have never appeared in binary tree before.*

Proof: Immediate from Lemma 14(3) and 14(4). ■

Corollary 16 *Consider any successful dchild* CAS*, $ccas_i$. One of the child pointers in the node that $f_i.p$ points to was $f_i.l$ when line 32 was last executed by the* SEARCH *belonging to f_i. Let ps be the other child pointer of $f_i.p$ at that time. Then, ps and $f_i.l$ are still the child pointers of $f_i.p$ just before $ccas_i$, and $ccas_i$ changes a child pointer of $f_i.gp$ from $f_i.p$ to ps.*

Proof: By Lemma 14(5), $f_i.p$'s child pointers do not change between the last execution of line 32 within the SEARCH belonging to f_i and $ccas_i$. So, just before the dchild CAS, line 104 sets the local variable *other* to ps, and the claim follows from the code of line 105. ■

5.5 Tree Properties

We have proved in Lemma 14(6) that the child pointers form a full binary tree. We can now prove some properties of that tree. An internal node is called *inactive* when it is first created. It becomes *active* when a successful ichild CAS writes a pointer to it for the first time. It remains active forever afterwards. An exception is the root node, which is active right from the beginning of the execution.

Lemma 17 *In every configuration, each active unmarked internal node is reachable from the root.*

Proof: Since marked nodes never become unmarked, we only have to show that every successful child CAS maintains this invariant.

First, consider a successful ichild CAS $ccas_i$ that changes a child pointer of $f_i.p$. We show that the only node that becomes unreachable from the root as a result of $ccas_i$ is a leaf. By Lemma 14(1), $ccas_i$ is the first change to R_i since it was read by r_i during a SEARCH. When that SEARCH executed r_i, it read the pointer to a leaf $f_i.l$. So, by Corollary 8, $f_i.l$ is the only node that becomes unreachable as a result of $ccas_i$.

Next, we show that the node y that becomes active as a result of $ccas_i$ is reachable from the root just after $ccas_i$. Lemma 12 and Lemma 14(2) implies that when $ccas_i$ occurs, $f_i.p$ is flagged, so it cannot be marked. Thus, $f_i.p$ is reachable from the root just prior to $ccas_i$, and $f_i.p$'s new child y is also reachable from the root after $ccas_i$.

Now suppose $ccas_i$ is a dchild CAS. By Corollary 16, $ccas_i$ changes the child of $f_i.gp$ from $f_i.p$ to $f_i.l$'s sibling. So the only nodes that become unreachable as a result of this CAS are $f_i.p$ and the subtree rooted at $f_i.l$. Since $f_i.l$ is a leaf, and $f_i.p$ is marked when cas_i occurs (by Lemma 10), the claim follows. ■

We say that node x is a *left-descendant* (*right-descendant*) of internal node y in configuration C if, in configuration C, x is in the tree and x is a descendant of y's left (right, respectively) child. The following lemma implies that nodes cannot acquire new ancestors.

Lemma 18 *If x is a left-descendant (right-descendant) of y at some configuration C, then x is a left-descendant (right-descendant, respectively) of y in all configurations between the first time x is in the tree and C.*

Proof: We write the proof for left-descendants; the proof for right-descendants is symmetric. Assume the claim is false to derive a contradiction. Then there is some successful child CAS, $ccas_i$, such that
(1) x is in the tree at some time before $ccas_i$,
(2) x is not a left-descendant of y in the configuration before $ccas_i$, and
(3) x is a left-descendant of y in the configuration after $ccas_i$.
If $ccas_i$ is an ichild CAS, this violates Lemma 14(3), because the nodes added to the tree by the ichild CAS must never have appeared in the tree before. If $ccas_i$ is a dchild CAS, this violates Corollary 16, because the dchild CAS removes two nodes from the tree and does not add any. ■

For any configuration C, let T_C be the binary tree formed by the child pointers in configuration C. We define the *search path* for key k in configuration C to be the unique path in T_C that would be followed by the ordinary sequential BST search procedure. (Although we have not yet proved that T_C is always a BST, we can still define this search path just by looking at what a BST search would do if it were run on T_C.)

Lemma 19 *If x is a node on the search path for key k in some configuration C and x is still in the tree in some later configuration C', then x is still on the search path for k in C'.*

Proof: Let y be any ancestor of x in $T_{C'}$. If x is a left-descendant of y in configuration C', then x is a left-descendant of y in configuration C (by Lemma 18), so $k < y.key$ (since x is on the search path for k in configuration C). By a symmetric argument, if x is a right-descendant of y in configuration C', then $k \geq y.key$. Thus, if a traversal of T'_C decides which direction to go at each ancestor y of x based on a comparison of that y's key with k, it will always choose to go towards the node x. Thus, x is on the search path for k in C'. ■

5.6 Proof of Linearizability

In this section, we prove that the implementation is linearizable. We shall define linearization points for each of the INSERT, DELETE and FIND operations. In addition, it will be convenient to define linearization points for calls to the SEARCH routine. The goal is to show that at any time T, the set of keys resulting from the sequence of update operations linearized before T is exactly the same as the set of keys that are currently stored in the leaves of the tree. Roughly speaking, we shall define a linearization point for each call to SEARCH(k) at which time the leaf returned by the SEARCH is on the search path for k in the tree. Thus, a FIND can be linearized at the same time as the SEARCH that it performs and each update operation that returns FALSE can be be linearized at the same time as the last SEARCH that it performs. Each update operation that returns TRUE will be linearized at the child CAS that changes the BST to reflect the update. (This child CAS may be performed by the process doing the update itself, or by another process that is helping the update.)

First, we define linearization points for the SEARCH routine. We wish to choose a linearization point for the SEARCH(k) so that the leaf the SEARCH returns is in the tree and on the search path for k at the time that it is linearized. It would be natural to try to linearize the SEARCH in the last iteration of its loop when it reads a pointer to that leaf. However, we have designed the SEARCH routine to ignore marks and flags. Thus, by the time the SEARCH reaches the leaf, that leaf may no longer be in the tree if, for example, the leaf's parent has been marked and removed by a concurrent DELETE operation. However, we can prove that the leaf *was* in the tree at some time during the execution of the SEARCH, using the next lemma.

We say that a SEARCH *enters* a node when a pointer to this node is assigned to l on line 25 or line 32.

Lemma 20 *Let $v_1, v_2 \ldots$, be the nodes entered by a* SEARCH(k) *(in the order they are entered). For all j, there is a configuration C_j such that*

1. *v_j is on the search path for k in configuration C_j,*
2. *C_j is before the* SEARCH *enters v_j, and*
3. *C_j is after the* SEARCH *is invoked.*

Proof: We prove the claim by induction on j.
Base Case ($j = 1$): Since *Root* never changes, v_1 is the root node. Let C_1 be the configuration immediately before the SEARCH enters v_1. Claim 1 is true because the root node is always on the search path for any key. Claim 2 and 3 follow from the definition of C_j.
Induction Step: Let $j > 1$. Assume the lemma holds for $j - 1$ and that the SEARCH enters a node v_j. We prove that the lemma holds for j. Let $enter_j$ be the step when the SEARCH enters v_j. This will be an execution of line 32, when a pointer to v_j is read in a child field of v_{j-1}.

In the special case where v_j is the one of the nodes initially in the tree, it must be the left child of the root, since all calls to SEARCH use keys smaller than ∞_1. In this case, we choose C_j to be the configuration right before $enter_j$. This choice clearly satisfies the claim.

Otherwise, there is some child CAS *ccas* that writes a pointer to v_j in a child field of v_{j-1} before $enter_j$. (In fact there is exactly one such child CAS, by Lemma 14(7).) Let C be configuration just after *ccas*. We define C_j to be either C_{j-1} or C, whichever is later.

By the induction hypothesis, C_{j-1} is after the SEARCH is invoked. Thus, C_j must also be after the SEARCH is invoked. By the induction hypothesis, C_{j-1} precedes $enter_{j-1}$, which precedes $enter_j$. Configuration C also precedes $enter_j$ (by definition), so C_j is before the SEARCH enters v_j. It remains to prove that v_j is on the search path for k in configuration C_j. We consider two cases.
Case 1: $C_j = C_{j-1}$. This means that *ccas* wrote a pointer to v_j in the child field of v_{j-1} before C_{j-1}, and the pointer was still there when $enter_j$ read that field after C_{j-1}. By Lemma 14(7), the child pointer must have contained the pointer to v_j at C_{j-1}. Thus, at C_{j-1}, v_{j-1} was on the search path for k (according to the induction hypothesis) and the child pointer of v_{j-1} that would be read by a search for k contained a pointer to v_j. Thus v_j is on the search path for k at $C_{j-1} = C_j$.

Case 2: $C_j = C$. The successful child CAS, *ccas* changes a child pointer of v_{j-1} immediately before C. By Lemma 12 and Lemma 14(2), v_{j-1} is flagged (and hence not marked) when *ccas* occurs. By Lemma 17, v_{j-1} is in the tree T_C. In some configuration C_{j-1} before C, v_{j-1} was on the search path for k, by the induction hypothesis. By Lemma 19, v_{j-1} is still on the search path for k in configuration C.

Step $enter_j$ reads the appropriate child of v_{j-1} (*i.e.*, the left child if $k < v_{j-1}.key$ and the right child otherwise), so v_j is the appropriate child of v_{j-1} in configuration C. Thus, v_j is on the search path for k in $C = C_j$. ■

For each SEARCH that terminates, we define its linearization point to be the configuration C_j corresponding to the last node the SEARCH enters, as defined in Lemma 20. Thus we have the following corollary.

Corollary 21 *Consider any execution of* SEARCH(k) *that terminates. The linearization point chosen for the* SEARCH *is during the* SEARCH. *The* SEARCH *returns a pointer l to a leaf, and that leaf is on the search path for k at the linearization point of the* SEARCH.

Recall the BST property states that, for each internal node x in the binary tree, the key of every left-descendant of x is smaller than x's key and the key of every right-descendant of x is greater than or equal to x's key. Now that we know that each SEARCH ends at the "correct" leaf of the BST, we can prove that the BST property is an invariant.

Lemma 22 *In every configuration, the tree of child pointers is a BST.*

Proof: The claim is true for the tree in the initial configuration. Since keys stored in nodes never change, we need only check that every child CAS preserves the invariant.

Assume the BST property holds prior to a dchild CAS. By Corollary 16, the dchild CAS replaces a subtree of the tree by a subtree of that subtree, so the BST property still holds after the dchild CAS.

Now consider an ichild CAS. Let C be the configuration just prior to the ichild CAS. Assume the BST property holds in C. We shall show that it also holds just after the ichild CAS. Let f be the IInfo record to which the ichild CAS belongs. Let INSERT(k) be the operation that created f. The ichild CAS changes a child pointer of $f.p$ from $f.l$ to $f.newInternal$, which points to an internal node with key $\max(k, f.l \rightarrow key)$. That internal node has two children with keys $f.l \rightarrow key$ and k (and they are in the right positions relative to their parent according to the way that they were constructed). Thus, we must just check that $f.l$ is on the search path for both k and $f.l \rightarrow key$ in C. The leaf $f.l$ is definitely on the search path for $f.l \rightarrow key$ in C, since the tree satisfies the BST property in C.

It remains to show that $f.l$ is on the search path for k in C. By Corollary 21, $f.l$ was on the search path for k at the linearization point of the SEARCH belonging to f. By Lemma 20, this linearization point occurs before C. Since $f.l$ is still in the tree at C, it follows from Lemma 19, that $f.l$ is still on the search path for k in C. ■

As mentioned above, we linearize each FIND operation that terminates at the same time as the linearization point for the SEARCH routine that it calls. Now we turn our attention to constructing linearization points for the update operations. We begin, in the next few lemmas, by showing that each update that returns TRUE has a unique successful child CAS and each update that returns FALSE has no successful child CAS.

Lemma 23 *Let f be any Info record. The first child* CAS *that belongs to f (if there is one) must succeed.*

Proof: Let *ccas* be the *first* child CAS that belongs to f. Let $fcas$ be the successful flag CAS for f, which exists and precedes *ccas*, according to Lemma 12.

Let x be the node whose child pointer *ccas* tries to change. Let r be the step in which the SEARCH belonging to f read the child pointer of x, as described in Lemma 7. Let r' be the previous step of the SEARCH, which read the update field of x at line 31. To show that *ccas* is successful, it suffices to prove that no successful child CAS on x occurs between r and *ccas* because the value read by r is used as the old value for the CAS step *ccas*.

Because $fcas$ succeeded, the value in $x.update$ just prior to $fcas$ must be the same as it was when it was read by r'. The step r' must have read CLEAN in $x.update.state$; otherwise, the test on line 51 (for INSERT) or 77 (for DELETE) would evaluate to TRUE, and f would not have been created. Thus, by Lemma 4, x's *state* is CLEAN at all times between r' and $fcas$.

We consider how $x.update$ can change after $fcas$. If f is a DInfo record, the next change to $x.update$ cannot be a backtrack CAS, because a successful mark CAS belonging to f was performed (by Lemma 10) and therefore there cannot be a backtrack CAS belonging to f (by Lemma 6). Thus, if $x.update$ is changed after $fcas$, the first change must be an unflag CAS belonging to f (by Lemma 3), which must occur after *ccas* since an unflag CAS is always immediately preceded in the code by a child CAS.

Thus, at all times between r and *ccas*, $x.update$ is either clean or flagged with a pointer to f. By Lemma 12 and Lemma 14(2), when any successful child CAS $ccas_i$ occurs on x, x is flagged with a pointer to f_i. Since there are no child CAS steps belonging to f before *ccas* (by definition of *ccas*), there cannot be any successful child CAS steps on x between r and *ccas*. Therefore, *ccas* will succeed. ■

Lemma 24 *If an* INSERT *or* DELETE *operation returns result* TRUE*, then during the operation, there is a successful child* CAS *that belongs to an Info record created by the operation.*

Proof: Consider any INSERT or DELETE operation that returns TRUE, and let f be the Info record that it creates during the last iteration of the while loop, just before it returns TRUE. Let $fcas$ be the flag CAS belonging to f that the operation performs. We first prove there is a child CAS that belongs to f by considering two cases.

If the operation is an INSERT, it can return TRUE (on line 59) only after executing HELPINSERT using a pointer to f on the preceding line. This HELPINSERT performs an ichild CAS belonging to f.

If the operation is a DELETE, it can return TRUE (on line 83) only if the call to HELPDELETE using a pointer to f returns TRUE. That call to HELPDELETE returns TRUE only after HELPMARKED is called, and HELPMARKED performs a child CAS belonging to f.

In both cases, there is some child CAS that belongs to f before the end of the INSERT or DELETE operation. The first child CAS that belongs to f, which must also be before the end of the operation, succeeds according to Lemma 23. That child CAS occurs after $fcas$, by Lemma 12. Thus the successful child CAS occurs during the INSERT or DELETE operation. ■

Lemma 25 *If a successful child* CAS *belongs to an Info record f created by an* INSERT *or* DELETE *operation, then f is created during the last iteration of the operation's while loop.*

Proof: First, consider an INSERT operation. Suppose there is a successful ichild CAS belonging to an IInfo record f created by the INSERT. By Lemma 12, there is a successful iflag CAS belonging to f. After this successful iflag CAS the INSERT routine returns TRUE on line 59 (unless it crashes), so the INSERT does perform any further iterations of its while loop.

Now, consider a DELETE operation. Suppose there is a successful dchild CAS belonging to a DInfo record f created by the DELETE. Let *mcas* be the *first* mark CAS belonging to f. We argue that *mcas* must succeed. By Lemma 9, a successful dflag CAS that belongs to f is performed before *mcas*. Prior to that dflag CAS, the SEARCH belonging to f reads the value $f.pupdate$ in $f.p \rightarrow update$. If *mcas* fails, the value of $f.p \rightarrow update$ was changed to some value other than $f.pupdate$ before *mcas*, so it will never be changed back to $f.pupdate$ (by Lemma 4). Thus, all the later mark CAS steps belonging to f will also fail, contradicting Lemma 10. So, *mcas* must succeed. So, $f.p \rightarrow update$ is equal to $\langle \text{DFLAG}, \&f \rangle$ at all times after *mcas*, by Lemma 3.

Now, after creating f, the DELETE routine calls HELPDELETE with a pointer to f. When HELPDELETE performs its mark CAS, it either succeeds (if it is the first mark CAS that belongs to f) or it sees that $p \rightarrow update$ is already equal to $\langle \text{DFLAG}, \&f \rangle$. Thus, the test on line 92 will evaluate to TRUE, and HELPDELETE will return TRUE (unless it crashes). Thus, the DELETE does not perform any further iterations of its while loop. ■

Corollary 26 *For each* INSERT *or* DELETE *operation, there is at most one successful child* CAS *belonging to the Info records created by the operation.*

Proof: By Lemma 25, only the last Info record created by the operation can have a successful child CAS that belongs to it. Only the first child CAS belonging to that Info record can succeed, by Lemma 14(1). ■

Corollary 27 *If an* INSERT *or* DELETE *returns* FALSE*, there is no successful child* CAS *that belongs to any Info record created by that operation.*

Proof: If the INSERT routine or DELETE routine returns FALSE, it does not create an Info record in the final iteration of the while loop (since it exits on line 50 or 76, respectively, before creating an Info record). Thus, by Lemma 25, there is no successful child CAS that belongs to any Info record created by the update operation. ■

We can now define the linearization points for update operations. If there is a successful child CAS that belongs to an Info record created by an update operation, the operation that created the Info record is linearized when that child CAS occurs. The choice of this linearization point is unique, by Corollary 26. This defines linearization points for all upates that return TRUE, by Lemma 24. By Lemma 27, this does not define a linearization point for any update operation that returns FALSE. If the update operation returns FALSE, we linearize it at the same time as the SEARCH that it performs in the last iteration of its while loop.

Lemma 28 *If an operation has a linearization point, then that linearization point is during the operation.*

Proof: For FIND operations and update operations that return FALSE, this follows from Corollary 21. For update operations that have a successful child CAS, this follows from Lemma 24. ■

The linearization points chosen for operations are either child CAS steps or configurations. The linearization ordering of all the operations is the order in which their linearization points occur in the execution (which, recall, is an alternating sequence of steps and configurations). The only operation that can be linearized at a child CAS is the update that created the Info record to which the child CAS belongs. However, there may be several operations that are linearized at a single configuration. (These will either be FIND operations or update operations that return FALSE.) To make the linearization order a total order on the operations, we can break the ties arbitrarily. It remains to show that all terminating operations output the same result as they would if they were performed in the linearization ordering.

We first consider the subsequence of update operations that are linearized at child CAS steps. For $i \geq 1$, let O_i be the update operation linearized at $ccas_i$. Let L_i be the set of keys in leaves of the tree in the configuration just after $ccas_i$. Let D_i be the set of keys that would be in a dictionary if operations $O_1, \ldots, O_i$ were performed sequentially, in that order, starting with an empty dictionary. (Let $D_0 = \emptyset$ and let L_0 be the set of keys in the leaves of the tree in the initial configuration, *i.e.*, $L_0 = \{\infty_1, \infty_2\}$.)

Lemma 29 *For all $i \geq 0$, the following statements are true.*

1. *If $i \geq 1$ and O_i is an* INSERT(k) *operation, then it returns* TRUE *and $k \notin D_{i-1}$.*
2. *If $i \geq 1$ and O_i is a* DELETE(k) *operation, then it returns* TRUE *and $k \in D_{i-1}$.*
3. *For all $i \geq 0$, $L_i = D_i \cup \{\infty_1, \infty_2\}$.*

Proof: We prove the lemma by induction on i.
Base case ($i = 0$): $L_0 = \{\infty_1, \infty_2\}$ and $D_0 = \emptyset$.
Inductive step: Let $i \geq 1$. Assume the claims hold for $i - 1$. We consider two cases.
Case 1: O_i is an INSERT(k) operation. Then O_i returns TRUE, since INSERT operations that return FALSE are not linearized at child CAS steps. Recall that f_i is the IInfo record that $ccas_i$ belongs to. The SEARCH that belongs to f_i returned a pointer $f_i.l$ with $f_i.l \rightarrow key \neq k$; otherwise the INSERT would have returned FALSE on line 50. By Corollary 21, $f_i.l$ is on the search path for k at the time the SEARCH is linearized.

Because $ccas_i$ succeeds, $f_i.p$ is flagged when it occurs (by Lemma 12 and Lemma 14(2)) and therefore $f_i.p$ is not marked. Thus, $f_i.p$ and $f_i.l$ are in the tree immediately before $ccas_i$ (by Lemma 17) . By Lemma

19, $f_i.l$ is still on the search path for k immediately before $ccas_i$. Since the tree is a BST (by Lemma 22), and there is a leaf $f_i.l$ on the search path for k in the tree that does not contain the key k, k must not appear in any leaf. Thus, $k \notin L_{i-1}$ and, by the induction hypothesis, $k \notin D_{i-1}$.

The effect of $ccas_i$ on the tree is to replace the leaf $f_i.l$ by a subtree containing two leaves whose keys are k and $f_i.l \to key$, by Corollary 15. Thus, $L_i = L_{i-1} \cup \{k\}$. So, we have $L_i = L_{i-1} \cup \{k\} = D_{i-1} \cup \{k\} = D_i$.

Case 2: O_i is a DELETE(k) operation. Then O_i returns TRUE, since DELETE operations that return FALSE are not linearized at child CAS steps. Recall that f_i is the DInfo record that $ccas_i$ belongs to. The SEARCH that belongs to f_i returned a pointer $f_i.l$ with $f_i.l \to key = k$; otherwise the DELETE would have returned FALSE on line 76.

Because $ccas_i$ succeeds, $f_i.gp$ is flagged when it occurs (by Lemma 12 and Lemma 14(2)) and therefore $f_i.gp$ is not marked. Thus, $f_i.gp$ is in the BST immediately before $ccas_i$ (by Lemma 17). By Corollary 16, $f_i.p$ is a child pointer of $f_i.gp$ and $f_i.l$ is a child pointer of $f_i.p$ immediately before $ccas_i$, so $f_i.l$ is also in the tree. Hence, $k = f_i.l \to key$ is in L_{i-1} and, by the induction hypothesis, $k \in D_{i-1}$.

The effect of $ccas_i$ is to replace $f_i.gp$'s pointer to $f_i.p$ by a pointer to $f_i.l$'s sibling, thus removing exactly one leaf $f_i.l$ from the tree (by Corollary 16. Thus, $L_i = L_{i-1} - \{k\}$. So, we have $L_i = L_{i-1} - \{k\} = D_{i-1} - \{k\} = D_i$. ■

Now we consider all the other operations and show that they also return the results they should, according to the linearization ordering.

Lemma 30 *Consider an operation O that is linearized after $ccas_i$ but before $ccas_{i+1}$ (if $ccas_{i+1}$ exists).*

1. *If O is a* FIND(k) *operation that returns $\perp$, then $k \notin D_i$.*
2. *If O is a* FIND(k) *operation that does not return $\perp$, then $k \in D_i$.*
3. *If O is an* INSERT(k) *operation, then it returns* FALSE, *and $k \in D_i$.*
4. *If O is a* DELETE(k) *operation, then it returns* FALSE, *and $k \notin D_i$.*

Proof: We prove each of the statements in turn.

1. If O is a FIND(k) operation that returns $\perp$, then its SEARCH returns a pointer to a leaf that is on the search path for k in the tree at its linearization point, by Corollary 21. Since the FIND returns $\perp$, that leaf contains a key different from k. Since the tree is a BST, k cannot be stored in any other leaf of the tree, so $k \notin L_i$. By Lemma 29, $k \notin D_i$.

2. If O is a FIND(k) operation that does not return $\perp$, then its SEARCH returns a pointer to a leaf that is in the tree at its linearization point, by Corollary 21. Thus, $k \in L_i = D_i$ by Lemma 29.

3. If O is an INSERT(k) operation, it returns FALSE since it is not linearized at a child CAS. Consider the SEARCH(k) routine that it called just before returning FALSE. That SEARCH returns a pointer to a leaf that is in the tree at its linearization point, by Corollary 21. Since the INSERT returns FALSE, that leaf has key k. Thus, $k \in L_i = D_i$ by Lemma 29.

4. If O is an DELETE(k) operation, it returns FALSE since it is not linearized at a child CAS. Consider the SEARCH(k) routine that it called just before returning FALSE. That SEARCH returns a pointer to a leaf that is on the search path for k in the tree at its linearization point, by Corollary 21. Since the DELETE returns FALSE, that leaf does not have key k. Since the tree is a BST, k cannot be stored in any other leaf of the tree, so $k \notin L_i$. By Lemma 29, $k \notin D_i$.

■

Lemma 31 *All* FIND *operations linearized before $ccas_1$ return $\perp$. All* DELETE *operations linearized before $ccas_1$ return* FALSE. *There are no* INSERT *operations linearized before $ccas_1$.*

Proof: Consider any DELETE(k) or FIND(k) operation linearized before $ccas_1$. It is linearized at the same time as its SEARCH is linearized. At that time the leaf returned by the SEARCH is on the search path for k in the tree, which is still as it was when it was initialized (since no successful child CAS steps have occurred yet), so the leaf returned by the SEARCH contains the key $\infty_1 \neq k$. Thus, the FIND would return $\perp$ and the DELETE would return FALSE.

To derive a contradiction, suppose an INSERT(k) is linearized before $ccas_1$. Then it must return FALSE (by the definition of the linearization points of INSERT operations). Thus, the SEARCH(k) called by the last iteration of the INSERT's while loop must have returned a leaf l containing the key k. The linearization point for the SEARCH is the same as for the INSERT, meaning that l was in the tree before $ccas_1$ (by Corollary 21), but this is impossible. ■

Theorem 32 *The implementation given in Figure 7, 8 and 9 is a linearizable implementation of a dictionary.*

Proof: By Lemma 29, 30 and 31 all terminating operations in the execution return the same response as they would if the operations were done in the linearization ordering. It follows from Lemma 28 that the linearization respects the real-time ordering of operations. ■

5.7 Progress

We first give simple proofs of three technical lemmas that will help ensure DELETE operations make progress. The first one shows that helping is guaranteed to remove a flag.

Lemma 33 *Suppose some process completes an execution of* HELP($\langle a, \&f \rangle$)*, where* $a \in \{\text{IFLAG}, \text{DFLAG}\}$*. Then there is a successful unflag or backtrack* CAS *belonging to* f*.*

Proof: We consider two cases. If a is IFLAG, then f is an IInfo record, and HELP calls HELPINSERT($\&f$), which performs a iunflag CAS belonging to f. The first such step succeeds.

If a is DFLAG, then f is a DInfo record, and HELP calls HELPDELETE($\&f$). If the test on line 92 evaluates to TRUE, then HELPMARKED performs a dunflag CAS belonging to f. If the test evaluates to FALSE, then a backtrack CAS belonging to f is performed. The first dunflag or backtrack CAS that belongs to f succeeds. ■

Lemma 34 *Let* f *be a DInfo record. If, in some configuration* C*, a node* x *has* $x.update = \langle \text{DFLAG}, \&f \rangle$ *and no dchild* CAS *belonging to* f *has yet occurred, then* $f.gp$ *points to* x *and* $f.p$ *points to a child of* x*.*

Proof: The only node that can be flagged with a pointer to f is the one that $f.gp$ points to, so $f.gp$ must point to x. Let r be the step in which the SEARCH belonging to f read $f.p$ in the child pointer of x, as described in Lemma 7. Let r' be the previous step of the SEARCH, which read the update field of x at line 31.

Let $fcas$ be the (unique) flag CAS belonging to f. Because $fcas$ succeeded, the value in $x.update$ just prior to $fcas$ must be the same as it was when it was read by r'. The step r' must have read CLEAN in $x.update.state$; otherwise, the test on line 51 (for INSERT) or 77 (for DELETE) would evaluate to TRUE, and f would not have been created. Thus, by Lemma 4, x's *state* is CLEAN at all times between r' and $fcas$.

Thus, at all times between r and C, $x.update$ is either clean or flagged with a pointer to f. By Lemma 12 and Lemma 14(2), when any successful child CAS $ccas_i$ occurs on x, x is flagged with a pointer to f_i. Since there are no child CAS steps belonging to f before C, there cannot be any successful child CAS steps on x between r and $ccas$. Thus, the node that $f.p$ points to is still a child of x in configuration C. ■

Lemma 35 *Let* f *be a DInfo record. At all times after some process performs a dchild* CAS *that belongs to* f*, the node that* $f.p$ *points to is not in the tree (i.e., it is not reachable from the root).*

Proof: Suppose some dchild CAS that belongs to f has been performed. The first dchild CAS that belongs to f succeeds, by Lemma 23. That dchild CAS changes one of $f.gp$'s child pointers from $f.p$ to one

of $f.p$'s children, according to Lemma 16. Since there is only one path from the root to $f.p$ prior to the dchild CAS, by Lemma 14(6), $f.p$ is no longer reachable from the root after the dchild CAS. It follows from Lemma 15 and 16 that no other child CAS can ever create a path leading from the root to $f.p$ after it is removed from the tree. ■

For any execution α, define the graph G_α to be the directed graph whose vertices are all internal nodes created during α. There is an edge from node x to node y in G_α if and only if y is a child of x at some time during α. (This graph may have infinitely many vertices for an infinite execution.)

Lemma 36 *For any execution α, G_α contains no cycles.*

Proof: It suffices to show, for all i, that there is no cycle in the subgraph G_i of G_α corresponding to the first i steps of the execution. We prove this by induction on i.
Base case ($i = 0$): G_0 contains a single node and no edges.
Inductive step: Let $i \geq 1$. Assume G_{i-1} contains no cycles. Unless the ith step is a child CAS, the set of edges in G_i is the same as in G_{i-1}, so it suffices to prove that G_i is acyclic when step i of the execution is a child CAS. By Corollary 15, an ichild CAS adds an edge to G_i from some node to a node that has no outgoing edges in G_i, so adding this edge cannot create a cycle in G_i. By Corollary 16, a dchild CAS belonging to f adds an edge to G_i from $f.gp$ to a child *other* of $f.p$ (unless *other* is a leaf, in which case no edges are added to G_i). Moreover, edges from $f.gp$ to $f.p$ and from $f.p$ to *other* already exist in G_{i-1}. So this change cannot introduce a cycle in G_i. ■

Theorem 37 *The implementation given in Figure 7, 8 and 9 is non-blocking.*

Proof: To derive a contradiction, assume there is some execution α that does not satisfy the nonblocking property. Thus there is a suffix of α in which no operations terminate and some operations take infinitely many steps. Let S be the set of operations that take infinitely many steps in α.

Claim 1: There are a finite number of successful child, mark, iflag, iunflag, and dunflag CAS steps in α.
Proof of Claim 1: There are a finite number of operations invoked in α. By Lemma 26, there is at most one successful child CAS per operation, so there are a finite number successful child CAS steps. Thus, there are a finite number of nodes ever added to the tree. By Lemma 3, there is at most one successful mark CAS per node in the tree. Any Insert that successfully performs its iflag CAS calls HelpInsert which terminates in two steps and then the Insert itself terminates. Thus each of the finitely many Insert operations in the execution has at most one successful iflag CAS. By Lemma 3 two successful iunflag CAS steps on a node's *update* field must have a successful iflag CAS between them. So there are a finite number of successful iflag and iunflag CAS steps. There is at most one successful dunflag CAS belonging to each DInfo record f. It is performed by HelpMarked, which is called only after there has been a successful mark CAS belonging to f. Thus, since there are a finite number of successful mark CAS steps, there are a finite number of successful dunflag CAS steps. This completes the proof of Claim 1.

Since there are a finite number of successful child CAS steps in α, the tree eventually stabilizes. Let T be this stable tree. Also, the graph G_α has a finite number of edges and has no cycles, by Lemma 36. Each iteration of line 32 in a Search corresponds to moving the pointer l from a node x to y, where (x, y) is an edge of G_α. Thus, any Search in α must terminate. Hence, there are no Find operations in S.

Calls to CAS-Child terminate in a constant number of steps. Therefore, calls to HelpInsert and HelpMarked also terminate in a constant number of steps.

Claim 2: Each call to Help and HelpDelete in α must terminate.
Proof of Claim 2: Help and HelpDelete do not contain any loops, and they only call each other or routines that are guaranteed to terminate. Thus, we must show that they do not call each other in a mutual recursion forever. To derive a contradiction, suppose this occurs. For all $i \geq 1$, let op_i be the argument to the

ith call to HelpDelete in this infinite recursion and let x_i be the node that $op_i \to p$ points to. Similarly, let u_i be the argument given to Help when the ith call to HelpDelete in this infinite recursion invokes it.

Let $i > 1$. We show (x_{i-1}, x_i) is an edge in G_α. When HelpDelete is called for the $(i-1)$th time, it performs an unsuccessful Mark CAS on $x_{i-1}.update$ and then calls Help using the value read from $x_{i-1}.update$ as the argument u_{i-1}. Then, since the mutual recursion continues, $u_{i-1}.state$ must be DFlag and $op_i = u_{i-1}.info$. Since $\langle \text{DFlag}, op_i \rangle$ appeared in $x_{i-1}.update$, $op_i \to gp$ must point to x_{i-1}, since it was placed there by a dflag CAS. By definition, $op_i \to p$ points to x_i. The pointers $op_i \to gp$ and $op_i \to p$ were written into op_i after they were returned by a Search, and the postconditions of that Search ensure that $op_i \to p$ was a child pointer read in the node that $op_i \to gp$ points to. Thus, x_i was a child of x_{i-1} at some time during that Search. So, (x_{i-1}, x_i) is an edge in G_α.

Thus, (x_{i-1}, x_i) is in G_α for all $i > 1$, yielding an infinite path in G_α, which is impossible (since G_α is acyclic and contains a finite number of edges). This completes the proof of Claim 2.

Thus, the only routines that might not terminate are Insert and Delete. If a call to either routine does not terminate, it must perform infinitely many iterations of the routine's while loop. Thus, each operation in S performs infinitely many calls to Search (with the same key as its argument). Let $O_1, \ldots, O_m$ be the operations in S. Since the tree eventually stabilizes, every call to Search by O_i initiated after the tree has stabilized will return pointers to the same three nodes, gp_i, p_i and l_i, by Lemma 20 and the postconditions of Search.

By Claim 1, the only types of successful CAS steps that occur infinitely often in α are dflag and backtrack CAS steps. Let α' be a suffix of α where

1. the only successful CAS steps in α' are dflag and backtrack CAS steps,
2. for all i, every invocation of Search by O_i in α' returns the pointers gp_i, p_i, l_i, and
3. for all i, the last invocation of Search by O_i in α prior to the beginning of α' also returned the pointers gp_i, p_i, l_i.

Choose gp_{lowest} to be one of the lowest of the nodes $gp_1, \ldots, gp_m$ in T (*i.e.*, none of the other gp_i's is a proper descendant of gp_{lowest} in T). Let $S_{lowest} = \{O_i : gp_i = gp_{lowest}\}$.

For all i, the only node that O_i can attempt a dflag CAS on in α' is gp_i. Thus, no operation attempts a dflag CAS on a proper descendant of gp_{lowest}. Therefore, there can be at most one successful backtrack CAS step applied to each proper descendant of gp_{lowest} in T, since there must be a successful dflag CAS on a node between two successful backtrack CAS steps on that node (by Lemma 3).

Thus, in some suffix of α', there are no successful CAS steps on the update fields of proper descendants of gp_{lowest} and the values in those fields stabilize. Let α'' be the suffix of α' where

1. the *update* fields of all proper descendants of gp_{lowest} in T never change, and
2. for all i, the last invocation of Search by O_i in α prior to the beginning of α'' began after the *update* fields of all proper descendants of gp_{lowest} in T had already stabilized.

Claim 3: If $O_i \in S_{lowest}$, then $p_i.state$ is not Mark during α''.

Proof of Claim 3: To derive a contradiction, suppose that during α'', $p_i.update = \langle \text{Mark}, \&f \rangle$ for some DInfo record f. The only step that can write this value is a mark CAS belonging to f, so $f.p$ points to p_i.

First, suppose O_i is an Insert. Then, O_i eventually calls Help($p_i.update$) on line 51. The Help routine then calls HelpMarked($\&f$), which performs a dchild CAS belonging to f. By Lemma 35, p_i is no longer reachable from the root after this dchild CAS. This contradicts the fact that O_i reaches p_i infinitely many times in α''.

Now, suppose O_i is a Delete. By Lemma 9, a successful dflag belonging to f was performed before $f.p$ was marked. By Lemma 35, no dchild CAS belonging to f occurs, since p_i remains in the tree forever. Thus, no dunflag CAS belonging to f ever occurs, since a dunflag CAS is always immediately preceded by

a dchild CAS. Furthermore, no backtrack CAS belonging to f occurs, by Lemma 6. Thus, the node that $f.gp$ points to remains flagged with a pointer to f forever, and hence remains in the tree forever, by Lemma 17. Furthermore, by Lemma 34, p_i remains a child of the node $f.gp$ points to forever. Thus $f.gp$ must point to gp_i since there is a unique path to each node in the tree (by Lemma 14(6)).

Therefore, O_i must eventually call HELP($\langle$DFLAG, $\&f\rangle$) on line 77, which calls HELPDELETE($\&f$). The mark CAS fails because $f.p \rightarrow update$ is already $\langle$MARK, $\&f\rangle$, so the process performing O_i then calls HELPMARKED($\&f$) which performs a dchild CAS belonging to f. By Lemma 35, p_i must not be reachable from the root after this occurs, contradicting the assumption that O_i's calls to SEARCH reach p_i infinitely often. This completes the proof of Claim 3.

Claim 4: S_{lowest} contains only DELETE operations.
Proof of Claim 4: To derive a contradiction, assume some $O_i \in S_{lowest}$ is an INSERT operation. (Recall that none of the operations in S can be FIND operations.) Note that $gp_i = gp_{lowest}$ and p_i is a child of gp_i, so p_i is a child of gp_{lowest}, and therefore $p_i \rightarrow update$ has stabilized in α'' (either to a flagged or clean value, by Claim 3). Consider an iteration of O_i's while loop in α''. If $p_i \rightarrow state$ is CLEAN in α'', the test on line 51 evaluates to FALSE, and O_i will perform a successful iflag CAS on $p_i \rightarrow update$, which is impossible because there are no successful iflag CAS steps in α''. Thus, $p_i \rightarrow state$ field must be either DFLAG or IFLAG in α'', and O_i invokes the HELP routine. By Lemma 33, the flag is eventually removed from p_i, contradicting the assumption that p_i's *update* field has stabilized in α''. Thus, S_{lowest} cannot contain any INSERT operations. This completes the proof of Claim 4.

Claim 5: In some configuration of α'', gp_{lowest} is clean.

Proof of Claim 5: If $gp_{lowest}.state$ is CLEAN at the beginning of α'', then the claim is clearly satisfied.

Next, suppose gp_{lowest} is marked at the beginning of α''. Let f be the DInfo record such that $gp_{lowest}.update = \langle$MARK, $\&f\rangle$. Then each process in S_{lowest} would eventually call HELP($\langle$MARK, $\&f\rangle$) from line 77, which would call HELPMARKED($\&f$). Thus, a dchild CAS belonging to f would be performed, and $f.gp = gp_{lowest}$ would be removed from the tree, by Lemma 35. This contradicts our assumption that each process in S_{lowest} reaches the node gp_{lowest} infinitely many times.

If $gp_{lowest}.state$ is IFLAG or DFLAG at the beginning of α'', then each process in S_{lowest} will eventually call HELP from line 77. By Lemma 33, gp_{lowest} will eventually become clean. This completes the proof of Claim 5.

Eventually, gp_{lowest} is clean, by Claim 5. The next change to gp_{lowest} can only be a dflag CAS, since no iflag CAS steps succeed in α''. We show that eventually some dflag CAS on gp_{lowest} does succeed. Suppose this is not the case. Then gp_{lowest} eventually stabilizes to a clean value. Thus, each DELETE operation O_i in S_{lowest} stops performing dflag steps. So, O_i must eventually evaluate the test on line 78 to be TRUE. Recall that p_i's *update* field does not change during α''. Since p_i is not marked, by Claim 3, p_i must be flagged throughout α''. Thus, O_i calls HELP on p_i's *update* field. By Lemma 33, the flag on p_i is eventually removed, contradicting the fact that p_i's *update* field does not change during α''. Thus, some DELETE O_i in S_{lowest} eventually dflags gp_{lowest}.

Finally, we derive the required contradiction. After its successful dflag CAS, O_i calls HELPDELETE, which attempts a mark CAS. Since p_i's update field has not changed since O_i read it in its previous SEARCH, the mark CAS succeeds, contradicting the fact that no successful mark CAS steps occur during α''.

This completes the proof that the implementation is non-blocking. ■

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 [18].

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 using a strategy similar to the one used for linked lists [5], 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 [19] 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 PODC 2010 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).

References

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

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. 17th ACM Symposium on Parallel Algorithms and Architectures*, pages 228–237, 2005.

[3] Joan Boyar, Rolf Fagerberg, and Kim 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] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proc. 22nd ACM Symposium on Parallel Algorithms and Architectures*, pages 335–344, 2010.

[5] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing*, pages 50–59, 2004.

[6] Keir Fraser and Timothy L. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):article 5, May 2007.

[7] Keir A. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, December 2003.

[8] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

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

[10] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proc. 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[11] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *Proc. 2nd ACM Symposium on Principles and Practice of Parallel Programming*, pages 197–206, 1990.

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

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

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

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

[16] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, 1980.

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

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

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

[20] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, 2002.

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

[22] Håkan Sundell and Philippas Tsigas. Scalable and lock-free concurrent dictionaries. In *Proc. 19th ACM Symposium on Applied Computing*, pages 1438–1445, 2004.

[23] John D. Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, 1995.

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