Non-blocking k-ary Search Trees

Trevor Brown and Joanna Helga

DisCoVeri Group, Dept. of Computer Science and Engineering, York University
4700 Keele St., Toronto, Ontario, Canada, M3J 1P3
tabrown@cs.toronto.edu, helga@yorku.ca

Abstract. This paper presents the first concurrent non-blocking k-ary search tree. Our data structure generalizes the recent non-blocking binary search tree of Ellen et al. [5] to trees in which each internal node has k children. Larger values of k decrease the depth of the tree, but lead to higher contention among processes performing updates to the tree. Our Java implementation uses single-word compare-and-set operations to coordinate updates to the tree. We present experimental results from a 16-core Sun machine with 128 hardware contexts, which show that our implementation achieves higher throughput than the non-blocking skip list of the Java class library and the leading lock-based concurrent search tree of Bronson et al. [3].

Keywords: data structures, non-blocking, concurrency, binary search tree, set

1 Introduction

With the arrival of machines with many cores, there is a need for efficient, scalable linearizable concurrent implementations of often-used abstract data types (ADTs) such as the set. Most existing concurrent implementations of the set ADT are lock-based (e.g., [3, 9]). However, locks have some disadvantages (e.g., [7]). Other implementations use operations not directly supported by most hardware, such as load-link/store-conditional [2] and multi-word compare-and-swap (CAS) [8]. Software transactional memory (STM) has been used to implement the set ADT (e.g., [10]), but this approach is currently inefficient [3].

Most multicore machines support (single-word) CAS operations. Non-blocking implementations of dictionaries have been given based on skip lists and binary search tree structures. Sundell and Tsigas [12], Fomitchev and Ruppert [6], and Fraser [8] have implemented a skip list using CAS operations. A binary search tree implementation using only CAS operations was sketched by Valois [13], but the first complete algorithm is due to Ellen et al. [5]. The non-blocking property ensures by definition

that, while a single operation may be delayed, the system as a whole will always make progress. (Some refer to this property as lock-freedom.)

In this paper, we generalize the binary search tree of Ellen et al. (BST) to a k-ary search tree (k-ST) in which nodes have up to $k-1$ keys and k children. This required generalizing the existing BST update operations to k-ary trees, creating new kinds of updates to handle insertion and deletion of keys from nodes, and verifying that the coordination scheme works with the new updates. Using larger values of k decreases the average depth of nodes, but increases the local work done at each internal node in routing searches and performing updates to the tree. However, the increased work at each node is offset by the improved spatial locality offered by larger nodes. By varying k, we can balance these factors to suit a particular system architecture, expected level of contention, or ratio of updates to searches. Searches are extremely simple and fast. Oblivious to concurrent updates, they behave exactly as they would in the sequential case.

We have implemented both the BST and our k-ST in Java, and have compared these implementations with ConcurrentSkipListMap (SL) of the Java class library, and the lock-based AVL tree of Bronson et al. (AVL) [3]. The AVL tree is the leading concurrent search tree implementation. It has been compared in [3] with SL, a lock-based red-black tree, and a red-black tree implemented using STM. Since SL and AVL drastically outperform the red-black tree implementations, we have not included the latter in our comparison. In our experiments, the BST and 4-ST (k-ST with $k = 4$) algorithms are top performers in both high and low contention cases. We did not observe significant benefits when using values of k greater than four, but we expect this will change with algorithmic improvements to the management of keys within nodes. This paper also provides the first performance data for the BST of Ellen et al. [5].

The BST and k-ST are both unbalanced trees. All performance tests in this paper use uniformly distributed random keys. If keys are not random then, in certain cases, SL (which uses randomization to maintain balance) and AVL (a balanced tree) will take the lead. Extending the techniques in this paper to provide balanced trees is the subject of current work.

2 *k*-ary Search Trees

2.1 The Structure

We use a leaf-oriented, non-blocking k-ST to implement the set ADT. A set stores a set of keys from an ordered universe. It does not admit duplicate keys. Here, we define the operations on the ADT to be FIND(key),

INSERT(key), and DELETE(key). The FIND operation returns TRUE if key is in the set, and FALSE otherwise. An INSERT(key) operation returns FALSE if key was already present in the set. Otherwise, it adds key to the set and returns TRUE. A DELETE(key) returns FALSE if key was not present. Otherwise, it removes key and returns TRUE. The other implementations we compare to the k-ST and BST can additionally associate a value with each key, and it is a simple task to modify our structure accordingly (discussed in [4]).

The k-ST is leaf-oriented, meaning that at all times, the keys in the set ADT are the keys in the leaves of the tree. Keys in internal nodes of the k-ST serve only to direct searches down the tree.

Each leaf in a BST contains one key. Each internal node has exactly two children and one key. In a k-ST, each leaf has at most $k-1$ keys. It is permitted for a leaf to have zero keys, in which case it is said to be an empty leaf. Each internal node has exactly k children and $k-1$ keys. Inside each node, keys are maintained in increasing order.

The search tree property for k-STs is a natural generalization of the familiar BST property. For any internal node with keys $a_1, a_2, ..., a_{k-1}$, sub-tree 1 (leftmost) contains keys $a < a_1$, sub-tree k (rightmost) contains keys $a > a_{k-1}$, and sub-tree $1 < i < k$ contains keys $a_i \leq a < a_{i+1}$.

2.2 Modifications to the Tree

We first describe a sequential implementation of the set operations, and subsequently transform it into a concurrent and non-blocking implementation. Since the k-ST is leaf-oriented, the INSERT and DELETE procedures always operate on leaves. Inserting a key into the set replaces a leaf by a "larger" leaf (with one more key), or by a small sub-tree if the leaf is already full (has $k-1$ keys). Deleting a key replaces a leaf by a smaller leaf (with one less key), or prunes the leaf and its parent out of the tree.

More precisely, the operation INSERT(key) first searches for key. If it is found, the INSERT returns FALSE. Otherwise, it proceeds according to two cases as follows (see Fig. 1). Let l be the leaf into which key should be inserted. If l is full (has $k-1$ keys) then INSERT replaces l by a *newly created* sub-tree of $k+1$ nodes. This sub-tree consists of an internal node n whose keys are the $k-1$ greatest out of the $k-1$ keys in l and the new key key. The children of n are k new nodes, each containing one of the k aforementioned keys. We call this first type of insertion a *sprouting insertion*. Otherwise, if l has fewer than $k-1$ keys, INSERT simply replaces l by a *new* leaf that includes key in addition to all of the keys that were in l. We call this second type of insertion a *simple insertion*.

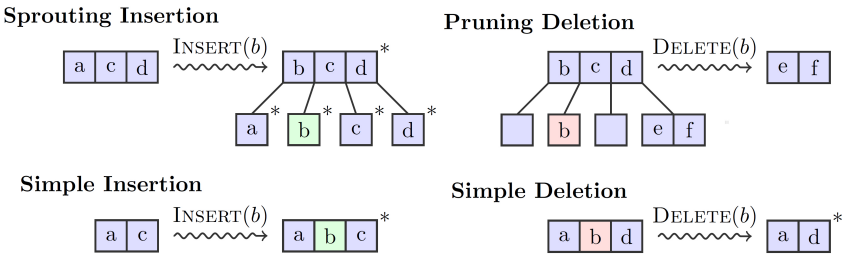

Fig. 1. The four types of modifications performed on the tree by an insertion or deletion. Asterisks indicate that nodes are newly created in freshly allocated memory.

The operation DELETE(key) first searches for key. If it is not found, then FALSE is returned. Otherwise, it proceeds according to two cases (see Fig. 1). Let l be the leaf from which key should be deleted. If l has only one key and the parent of l has exactly two non-empty children, then the entire leaf l can be deleted (since it will be empty after the deletion) and, because it has only one non-empty sibling s, the parent node is no longer useful (since its keys just direct searches). Thus, the DELETE procedure simply replaces the parent with s. We call this first type of deletion a *pruning deletion*. Otherwise, if l has more than one key or the parent of l has more than two non-empty children, DELETE replaces l by a *new* leaf with all of the keys of l except for key.We call this second type of deletion a *simple deletion*. Simple deletion can yield empty leaves. However, with this insertion and deletion scheme an internal node always has at least two non-empty children. One might be tempted to use NULL instead of empty children, but then child pointers would suffer the ABA problem.

Note that a pruning deletion changes a child pointer of the grandparent of l to point to l's only non-empty sibling. To avoid dealing with degenerate cases when there is no parent or grandparent of l, we initialize the tree with two dummy internal nodes and $2k - 1$ empty leaves at the top, as shown in Fig. 2(a). These internal nodes will not be deleted or replaced by an insertion. When $k = 2$, our algorithm is simply the BST of Ellen et al. [5], with some slight modifications, where nearly all insertions and deletions are sprouting insertions and pruning deletions (except for an INSERT into an empty tree and a DELETE on the last key in a tree).

2.3 Coordination Between Updates

Without some form of coordination, interactions between concurrent updates would produce incorrect results. Suppose that a pruning deletion and a simple insertion are performed concurrently in the 2-ary tree on

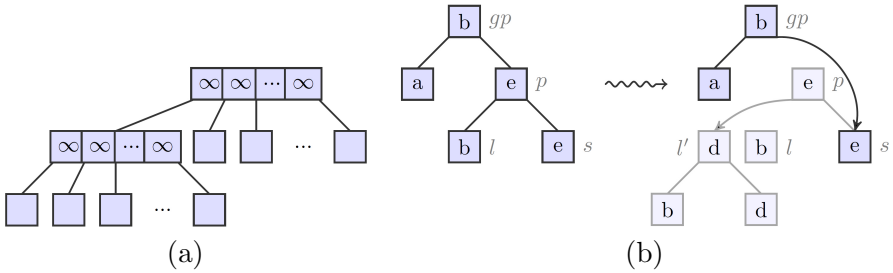

Fig. 2. (a) The initial state of the k-ary search tree. The root and its leftmost child have $k-1$ keys valued ∞ (a special key, larger than any key in the set). All other children of these nodes are empty leaves. Keys in the set are stored in the sub-tree rooted at the leftmost grandchild of the root. (b) Example of the danger of uncoordinated concurrent updates. Faintly shaded nodes are no longer in the tree. If $gp.right$ is changed to s by a DELETE(b), and $p.left$ is changed to l' by a concurrent INSERT(d), then the new key d is lost.

the left in Fig. 2(b). If the steps of the INSERT(d) and DELETE(b) are interleaved in a particular order, key d may be inserted as a grandchild of p, and erroneously deleted along with b.

To avoid situations such as this, each internal node is augmented to contain an UpdateStep object that indicates an operation has exclusive access to the child pointers of a node. This coordination scheme extends the work of Ellen et al. [5]. UpdateStep objects serve as something similar to locks, because all processes operate under the following agreement. When an operation intends to modify a child pointer of an internal node n, it first stores an UpdateStep object at n (using CAS). An operation cannot store an UpdateStep at node n if another operation x has already stored an UpdateStep at n, until x has relinquished control of n. Thus, UpdateStep objects behave like locks that are owned by an operation, rather than by a process, and this allows us to guarantee the non-blocking property by using the helping mechanism described in Sec. 2.4.

UpdateStep objects are divided into flags and marks. A flag is placed on a node to reserve its child pointers for exclusive access, indicating that one will be changed by an operation. A mark is similar to a flag except, where a flag is temporary (removed once a modification is completed), a mark is permanent, and is placed on a node that is to be removed from the tree. The mark permanently prevents the child pointers of the node from ever changing after it is removed. The final type of UpdateStep object is the Clean object which, if stored at a node x, indicates that no operation has exclusive access to x, and any operation is allowed to store a flag or mark there.

The details of the INSERT and DELETE operations, including flagging and marking steps, are as follows. In the following, l is the target leaf for insertion or deletion of a key, p is its parent, and gp is its grandparent. A simple insertion or simple deletion (see Fig. 1) creates the new leaf, flags p with a ReplaceFlag object (with a *flag CAS*), changes the child pointer of p (with a *child CAS*), and unflags p (with an *unflag CAS*) by writing a new Clean object. Similarly, a sprouting insertion creates the new sub-tree, flags p with a new ReplaceFlag object, changes the child pointer of p, and unflags p. A pruning deletion flags gp with a PruneFlag object, then attempts to mark p with a *mark CAS*. If the *mark CAS* is successful, then the child pointer of gp is changed, finishing the deletion, and gp is unflagged. Otherwise, if the marking step fails, then the DELETE must unflag gp (with a *Backtrack CAS*) and try again from scratch.

We now return to Fig. 2(b) to illustrate how flagging and marking resolves the issue. After DELETE(b) has successfully stored a PruneFlag at gp, it must store a Mark at p. Say the Mark is successfully stored at p. Then it is safe to prune l and p out of the tree, since no child pointer of p will ever change, and l is a leaf (which has no mutable fields). Once l and p are pruned out of the tree, gp is unflagged by an *unflag CAS* that replaces the PruneFlag stored by DELETE(b) by a newly created Clean object. If INSERT(a) subsequently tries to change a child reference belonging to p, it will first have to store a ReplaceFlag at p, which is impossible, since p is already marked. Otherwise, if the Mark cannot be successfully be stored at p (because p is already flagged or marked by another operation), then DELETE(b) will execute a *Backtrack CAS*, storing a new Clean object at gp (relinquishing control of gp to allow other operations to work with it), and retry from scratch.

2.4 Helping

To overcome the threat of deadlock that is created by the exclusive access that flags and marks grant to a single operation, we follow the approach taken by Ellen et al. [5], which has some similarities to Barnes' cooperative technique [1]. Suppose that a process P flags or marks a node hoping to complete some tree modification C. The flag or mark object is augmented to contain sufficient information so that any process can read the flag or mark and complete C on P's behalf. This allows the entire system to make progress even if individual processes are stalled indefinitely.

Unfortunately, while helping guarantees progress, it can mean duplication of effort. Several processes may come across the same UpdateStep

```
1   ▷ Type definitions:
2   type Node {
3     final Key ∪ {∞}  a_1, ..., a_{k-1}
4   }
5   subtype Leaf of Node {
6     final int keyCount
7   }
8   subtype Internal of Node {
9     Node  c_1, ..., c_k
10    UpdateStep  pending
        ▷ (initially a new Clean() object)
11  }
12  type UpdateStep { }
13  subtype ReplaceFlag of UpdateStep {
14    final Node  l, p, newChild
15    final int pindex
16  }
17  subtype PruneFlag of UpdateStep {
18    final Node  l, p, gp
19    final UpdateStep  ppending
20    final int gpindex
21  }
22  subtype Mark of UpdateStep {
23    final PruneFlag  pending
24  }
25  subtype Clean of UpdateStep { }
26  ▷ Initialization:
27  shared Internal root := the structure
      described in Fig. 2(a), with the pending
      fields of root and root.c_1 set to refer to
      new Clean objects.
```

Fig. 3. Type definitions and initialization.

object and perform the work necessary to advance the operation by performing some local work, followed by a *mark CAS*, *child CAS*, or *unflag CAS*, but only one process can successfully perform each CAS, so the local work performed by all other processes is wasted. For this reason it is advantageous to limit helping as much as possible. To this end, a search ignores flags and marks in our implementation, and proceeds down the tree without helping any operation. An INSERT or DELETE helps only those operations that interfere with its own completion. Thus, an INSERT will only help an operation that has flagged or marked p, and a DELETE will only help an operation that has flagged or marked p or gp (although they may help other operations recursively). After an INSERT or DELETE helps another operation, it restarts, performing another search from the top of the tree. An INSERT or DELETE operation is repeatedly attempted until it successfully modifies the tree or finds that it can return FALSE.

2.5 Pseudocode

Java-like pseudocode for all operations is found in Fig. 3 through Fig. 5. We borrow the concept of a *reference* type from Java. Any variable x of type C, where C is a type defined in Fig. 3, is a reference to an instance (or object) of type C. Such an x behaves like a C pointer, but does not require explicit dereferencing. References can point to an object or take on the value NULL, and management of their memory is automatic: memory is garbage-collected once it is unreachable from any executing thread. We use $a.b$ to refer to field b of the object referred to by a. We also adopt a

Java-like definition of CAS, that atomically compares a field R with an expected value *exp* and either writes a new value and returns true (if R contains *exp*), or returns false (otherwise).

The SEARCH(*key*) operation is straightforward. Beginning at the left-most child of *root* (line 29) and continuing until it reaches a leaf (line 32), it compares its argument *key* with each key stored at the current node and follows the appropriate child reference (line 36), saving some information along the way. (Keys of a node can be naively inspected in sequence because the keys of a node never change.) The FIND(*key*) operation returns TRUE if the SEARCH(*key*) operation finds a leaf containing *key*; otherwise it returns FALSE.

To perform an INSERT(*key*), a process P locates the leaf l and its parent p, and stores the parent's *pending* field in *ppending* and the index of the child reference of *gp* that contained p in *pindex* (line 49). If *key* is already in l, then the operation simply returns FALSE (line 50). Otherwise, P checks whether the parent's *pending* field was of type Clean when it was read (line 51). If not, then *p.pending* was occupied by a flag or mark belonging to some other operation x in progress at p. P helps x complete, and then re-attempts its own operation from scratch. Otherwise, if *p.pending* was Clean, P tries to flag p by creating *newChild*, a new leaf or sub-tree depending on which insertion case applies (lines 54 to 58), the ReplaceFlag object *op* (line 59), and executing an *Rflag CAS* to store it in the *pending* field of p (line 60). If the *Rflag CAS* succeeds, P calls HELPREPLACE(*op*) to finish the insertion (line 62) and the operation returns TRUE. Otherwise, if the *Rflag CAS* failed, another process must have changed p's *pending* field to a ReplaceFlag object, a PruneFlag object, a Mark object, or a new Clean object (different from the one read at line 49). Process P helps this other operation (if not a Clean object) complete, and then re-attempts its own operation.

A call to HELPREPLACE executes a *child CAS* to change the appropriate child pointer of p from l to *newChild* (line 116), and executes an *Runflag CAS* to unflag p (line 117).

When process P performs a DELETE(*key*) operation, it first locates the leaf l, its parent p and grandparent *gp*, and stores the parent's and grandparent's *pending* fields in *ppending* and *gppending*, and the indices of the child references of *gp* and p that contained p and l, respectively, in *gpindex* and *pindex* (line 78). If l does not contain *key*, then the operation simply returns FALSE (line 79). Otherwise, P checks *gppending* and *ppending* to determine whether *gp* and p were Clean when their *pending* fields were read (lines 80 and 82). If either has been flagged or marked

```
28  SEARCH(Key key) : ⟨Internal, Internal, Leaf, UpdateStep, UpdateStep⟩ {
      ▷ Used by INSERT, DELETE and FIND to traverse the k-ST
      ▷ SEARCH satisfies following postconditions:
      ▷ (1) leaf points to a Leaf node, and parent and gparent point to Internal nodes
      ▷ (2) parent.c_pindex has contained leaf, and gparent.c_gpindex has contained parent
      ▷ (3) parent.pending has contained ppending,
            and gparent.pending has contained gppending
29    Node gparent, parent := root, leaf := parent.c1
30    UpdateStep gppending, ppending := parent.pending
31    int gpindex, pindex := 1
32    while type(leaf) = Internal {        ▷ Save details for parent and grandparent of leaf
33      gparent := parent; gppending := ppending
34      parent := leaf; ppending := parent.pending
35      gpindex := pindex
36      ⟨leaf, pindex⟩ := ⟨appropriate child of parent by the search tree property,
                          index such that parent.c_pindex is read and stored in leaf⟩
37    }
38    return ⟨gparent, parent, leaf, ppending, gppending, pindex, gpindex⟩
39  }

40  FIND(Key key) : boolean {
41    if Leaf returned by SEARCH(key) contains key, then return TRUE, else return FALSE
42  }

43  INSERT(Key key) : boolean {
44    Node p, newChild
45    Leaf l
46    UpdateStep ppending
47    int pindex
48    while TRUE {
49      ⟨−, p, l, ppending, −, pindex, −⟩ := SEARCH(key)
50      if l already contains key then return FALSE
51      if type(ppending) ≠ Clean then {
52        HELP(ppending)                          ▷ Help the operation pending on p
53      } else {
54        if l contains k − 1 keys {              ▷ Sprouting insertion
55          newChild := new Internal node with pending := new Clean(),
                and with the k − 1 largest keys in S = {key} ∪ keys of l,
                and k new children, sorted by keys, each having one key from S
56        } else {                                ▷ Simple insertion
57          newChild := new Leaf node with keys: {key} ∪ keys of l
58        }
59        ReplaceFlag op := new ReplaceFlag(l, p, newChild, pindex)
60        boolean result := CAS(p.pending, ppending, op)          ▷ Rflag CAS
61        if result then {                        ▷ Rflag CAS succeeded
62          HELPREPLACE(op)                       ▷ Finish the insertion
63          return TRUE
64        } else {                                ▷ Rflag CAS failed
65          HELP(p.pending)                       ▷ Help the operation pending on p
66  } } } }

67  HELP(UpdateStep op) {
      ▷ Precondition: op ≠ NULL has appeared in x.pending for some internal node x
68    if type(op) = ReplaceFlag then HELPREPLACE(op)
69    else if type(op) = PruneFlag then HELPPRUNE(op)
70    else if type(op) = Mark then HELPMARKED(op.pending)
71  }
```

Fig. 4. Pseudocode for SEARCH, FIND, INSERT and HELP.

```
72  DELETE(Key key) : boolean {
73    Node gp, p
74    UpdateStep gppending, ppending
75    Leaf l
76    int pindex, gpindex
77    while TRUE {
78      ⟨gp, p, l, ppending, gppending, pindex, gpindex⟩ := SEARCH(key)
79      if l does not contain key, then return FALSE
80      if type(gppending) ≠ Clean then {
81        HELP(gppending)                    ▷ Help the operation pending on gp
82      } else if type(ppending) ≠ Clean then {
83        HELP(ppending)                     ▷ Help the operation pending on p
84      } else {                             ▷ Try to flag gp
85        int ccount := number of non-empty children of p (by checking them in sequence)
86        if ccount = 2 and l has one key then ▷ Pruning deletion
87          PruneFlag op := new PruneFlag(l, p, gp, ppending, gpindex)
88          boolean result = CAS(gp.pending, gppending, op)        ▷ Pflag CAS
89          if result then {                 ▷ Pflag CAS successful–now delete or unflag
90            if HELPPRUNE(op) then return TRUE;
91          } else {                         ▷ Pflag CAS failed
92            HELP(gp.pending)               ▷ Help the operation pending on gp
93          }
94        } else {                           ▷ Simple deletion
95          Node newChild := new copy of l with key removed
96          ReplaceFlag op := new ReplaceFlag(l, p, newChild, pindex)
97          boolean result := CAS(p.pending, ppending, op)         ▷ Rflag CAS
98          if result then {                 ▷ Rflag CAS succeeded
99            HELPREPLACE(op)                ▷ Finish inserting the replacement leaf
100           return TRUE
101         } else {                         ▷ Rflag CAS failed
102           HELP(p.pending)                ▷ Help the operation pending on p
103 } } } } }

104 HELPPRUNE(PruneFlag op) : boolean {      ▷ Precondition: op is not NULL
105   boolean result := CAS(op.p.pending, op.ppending, new Mark(op))  ▷ mark CAS
106   UpdateStep newValue := op.p.pending
107   if result or newValue is a Mark with newValue.pending = op then {
108     HELPMARKED(op)                       ▷ Marking successful–complete the deletion
109     return TRUE
110   } else {                               ▷ Marking failed
111     HELP(newValue)                       ▷ Help the operation pending on p
112     CAS(op.gp.pending, op, new Clean())  ▷ Unflag op.gp          ▷ Backtrack CAS
113     return FALSE
114 } }

115 HELPREPLACE(ReplaceFlag op) {           ▷ Precondition: op is not NULL
116   CAS(op.p.c_{op.pindex}, op.l, op.newChild)  ▷ Replace l by newChild ▷ Rchild CAS
117   CAS(op.p.pending, op, new Clean())     ▷ Unflag p                  ▷ Runflag CAS
118 }

119 HELPMARKED(PruneFlag op) {              ▷ Precondition: op is not NULL
120   Node other := any non-empty child of op.p
        (found by visiting each child of op.p), or op.p.c_1 if none
121   CAS(op.gp.c_{op.gpindex}, op.p, other)  ▷ Replace l by other       ▷ Pchild CAS
122   CAS(op.gp.pending, op, new Clean())     ▷ Unflag gp                ▷ Punflag CAS
123 }
```

Fig. 5. Pseudocode for DELETE, HELPPRUNE, HELPREPLACE and HELPMARKED.

by another operation, P helps complete this operation and re-attempts its own operation from scratch. Otherwise, it counts the number of non-empty children of p to determine the deletion case to apply. We shall explain why counting the children in sequence is not problematic when we discuss correctness. We consider the two types of deletion separately.

If the operation is a simple deletion (line 94), it creates *newChild*, a new copy of leaf l with *key* removed, and a new ReplaceFlag object *op* to facilitate helping (line 96). Next, P attempts an *Rflag CAS* to store *op* in *p.pending* (line 97) and, if it succeeds, it calls HelpReplace to finish the deletion (line 99). Otherwise, if the *Rflag CAS* fails, P helps any operation that may be pending on p. After helping, P retries its own operation from scratch. Note that, apart from the creation of the new leaf, this is identical to simple insertion.

If the operation is a pruning deletion (line 86), P creates a PruneFlag object (line 87), then attempts a *Pflag CAS* to store it in the *pending* field of *gp* (line 88). If the *Pflag CAS* succeeds, P calls HelpPrune(*op*) to finish the deletion (line 90) and the operation returns True (more on HelpPrune later). Otherwise, if the *Pflag CAS* fails, another process must have changed *gp*'s *pending* field to a ReplaceFlag object, a PruneFlag object, a Mark object, or a new Clean object (different from the one read at line 78). To allow any other operation pending on *gp* to make progress, P calls Help(*gp.pending*) (line 92) before retrying its own operation from scratch.

The HelpPrune procedure, invoked by the Delete operation (and by Help), attempts the second (marking) CAS step of a pruning deletion. Recall that *op*, created in the Delete routine, contains pointers to l, the leaf containing the key to be deleted, its parent p, and its grandparent *gp*. The HelpPrune procedure begins by attempting to mark the parent *op.p* (line 105). If the CAS successfully marks *op.p*, or another helping process already stored a mark for this operation, then the mark is considered to be successful. In this case, HelpMarked is called to finish the pruning deletion (line 108), and True is returned. Otherwise, if the CAS failed and the mark was not already stored by a helping process, then another operation involving *op.p* has interfered with the Delete. If the other operation is still in progress, it is helped (line 111), and then the operation backtracks, unflagging the grandparent *op.gp* (line 112), and False is returned by HelpPrune. The process that invoked the Delete procedure will ultimately retry the operation from scratch.

The HelpMarked procedure performs the final step of a pruning deletion, pruning out some dead wood by changing the appropriate child

pointer of *op.gp* from *op.p* to point to the only non-empty sibling of *op.l*. This sibling of *op.l* is found at line 120. (It is explained in Sec. 2.6 why this can be found simply by visiting each child of *op.p*.) The CAS-CHILD routine is invoked to change the child pointer of *op.gp* (line 121), and an *unflag CAS* is executed to unflag *op.gp* (line 122).

2.6 Correctness

It can be demonstrated that our algorithm exhibits linearizability (defined in [11]), and the argument is very similar to the one made in the proof in [5]. We simply give the linearization points of operations here, and defer the proof to Appendix A(see [4]). Consider some invocation of SEARCH(*key*). It can be proved that each node on the search path for *key* was in the tree at some point during the SEARCH, so we linearize SEARCH at a point when the leaf it returns was in the tree. An invocation of FIND(*key*) is linearized at the point its corresponding SEARCH was linearized. It can be proved that each INSERT or DELETE invocation that returns TRUE has executed a successful *child CAS*. An invocation of INSERT(*key*) or DELETE(*key*) that returns TRUE is linearized at this *child CAS*; an invocation that returns FALSE is linearized at the same point as the corresponding SEARCH that discovered *key* was already in the tree, or was not in the tree, respectively.

The k-ST algorithm differs significantly from the BST algorithm at lines 85 and 120, which both involve accessing several children in sequence. Let P be a process executing line 85. We note that no flagging or marking has yet been attempted by P, and the expected values to be used by the CASs at lines 88 and 97 were verified to be Clean a few lines prior. Further, if any process Q wants to add or remove a key from a child x of p that P will read at line 85, it must replace x, changing a child pointer of p. However, it must flag p to change its child pointers, overwriting the Clean object that was read earlier by P to be used as the expected value for its flag CAS. It is easy to prove that there is no ABA problem on pending fields, which implies that the expected value used by P for the CAS can never appear in *p.pending* again, so P's CAS must fail, and the operation will be retried. It can then be shown that if an operation *op* successfully flags or marks p, *ccount* contains the number of non-empty children of p until a *child CAS* is executed for *op*, and that only the first *child CAS* will be successful (occurring immediately after line 120). Thus, it can be shown that when line 120 is executed, the children of *op.p* are precisely *op.l* and one other leaf, or else the *child CAS* will fail, so the value of

other is irrelevant. This is rigorously demonstrated in the detailed proof of correctness presented in Appendix A(see [4]).

3 Experiments

In this section we present results from experiments comparing the performance of the BST of Ellen et al. [5], our k-ST algorithm, ConcurrentSkipListMap (SL) of the Java class library and the lock-based AVL tree (AVL) of Bronson et al. [3]. Experiments on each structure used put-if-absent and delete-if-present (set functions), returning TRUE if the operation could be completed, and FALSE otherwise. Preliminary experiments were run to tune the parameters of the final experimental set to maximize trial length while keeping standard deviations reasonable. The final experiments each consisted of selecting a particular algorithm and executing a sequence of 17 three-second trials, in which a fixed number of threads randomly perform INSERTs, DELETEs and FINDs according to a desired probability distribution (e.g., 5% INSERT, 5% DELETE, 90% FIND), on uniformly distributed random keys, drawn from a particular key range (e.g., the integers from 0 to 10^6). The average throughput (operations per second) was recorded for each trial, and the first few trials were discarded to account for the few seconds of "warm-up" time that the Java Virtual Machine (VM) needs to perform just-in-time compilation and optimization. We observed that throughput stabilized after the first three to five seconds of execution, so the first two trials (six seconds) of each experiment were discarded. Garbage collection was also triggered in between trials to minimize its haphazard impact on measurements.

Our experiments were run on a Sun machine at the University of Rochester, with two UltraSPARC-III CPUs, each having eight 1.2GHz cores capable of running 8 hardware threads apiece (totalling 128 hardware contexts), and 32GB of RAM, running Sun's Solaris 10 and the Java 64-bit VM version 1.6.0_21 (with 15GB initial and maximum heap sizes).

We call the probability distribution of INSERTs and DELETEs a *ratio*, and denote an experiment with x% INSERTs, y% DELETEs and $(100 - x - y)$% FINDs as, simply xi-yd. We denote the key range of integers from 0 to $10^x - 1$ by $[0, 10^x)$. The experimental results we present herein used algorithms BST, 4-ST, SL and AVL, key ranges $[0, 10^2)$ and $[0, 10^6)$ and ratios $0i$-$0d$, $5i$-$5d$, $8i$-$2d$ and $50i$-$50d$. The key ranges induce high and low levels of contention, respectively, with small trees increasing the probability that operations on random keys will coincide. The four ratios represent situations in which operations consist (1) entirely of searching,

(2) mostly of searching, (3) mostly of searching, but with far more INSERTs than DELETEs, and (4) entirely of updates. Initially, each data structure was empty for each trial, except when the ratio was $0i$-$0d$, since that would mean performing all operations on an empty tree. In this case, each structure was pre-filled at the beginning of each trial by performing random operations in the ratio $50i$-$50d$ until the structure's size stabilized (to within 5% of the expected half-full). Additional results, including more operation mixes and key ranges, and results from a 32-core system at Intel's Multicore Testing Lab can be found in [4]. For implementations of the BST and k-ST, see [4].

We now discuss the graphs presented in Fig. 6. The $[0, 10^2)$ key range represents very high contention. There were at most 10^2 keys in the set, and as many as 128 threads accessing the tree. Under this load, BST was the top performer in all experiments. The low degree of BST's nodes permits many simultaneous updates to different parts of the tree, and its simplicity offers strong performance. 4-ST matched BST's performance in the $0i$-$0d$ and $8i$-$2d$ cases, indicating that, in the absence of many deletions, it can perform just as well under extremely high contention. For the other two ratios, 4-ST's performance was similar to the lock-free SL, surpassing AVL by a fair margin. BST scaled very well in all cases; 4-ST scaled equally well when deletions were few.

The $[0, 10^6)$ key range represents low contention: with as many as one million keys and only 128 threads, the chance of collisions in random keys is quite small. With this level of contention, 4-ST exhibits strong performance, surpassing BST, and the other algorithms. This is in line with expectations; as the size of the tree increases, the higher degree of the 4-ST affords it a shallower depth, allowing all operations to complete more quickly. Unlike the $[0, 10^2)$ case, all algorithms scale reasonably well in the $[0, 10^6)$ case, approaching linear improvement in throughput with an increase in the number of hardware threads.

4 Conclusion and Future Work

BST has the greatest advantage in high contention settings. Its simplicity pushes its performance beyond the other algorithms. As trees get larger and contention decreases, 4-ST surpasses BST to become the top performer. Similar to 4-ST, AVL also performs well as the size of the data structure increases. SL tends to performs well when its set of keys is small.

AVL is a balanced tree, so it does some extra work in maintaining this property. However, since our experiments insert random keys, 4-ST and

BST also are nearly balanced. In this experimental setting, the balancing work of AVL does not pay off. In a situation where the keys inserted are not random, AVL might have a significant advantage over 4ST and BST. Since in many cases BST and 4ST outperform AVL and SL by a fair margin, we believe that it may be possible to add balancing and remain competitive, while offering a non-blocking progress guarantee.

Acknowledgments. We thank Michael L. Scott for providing access to the multi-core machine at the University of Rochester. Financial support for this research was provided by NSERC. We also thank Eric Ruppert and Franck van Breugel for their supervision and assistance in the preparation of this manuscript. Finally, we thank the anonymous OPODIS reviewers for their comments.

References

1. G. Barnes. A method for implementing lock-free data structures. In *Proc. 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.
2. M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. 17th ACM Symposium on Parallel Algorithms and Architectures*, pages 228–237, 2005.
3. N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming*, pages 257–268, 2010.
4. T. Brown and J. Helga. Non-blocking k-ary search trees. Technical Report CSE-2011-04, York University, 2011. Appendix and code available at `http://www.cs.toronto.edu/~tabrown/ksts`.
5. F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010. Full version in Tech. Report CSE-2010-04, York University.
6. M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing*, pages 50–59, 2004.
7. K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):5, 2007.
8. K. A. Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, 2003.
9. L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symp. on Foundations of Computer Science*, pages 8–21, 1978.
10. M. 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. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
12. H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *Proc. 19th ACM Symposium on Applied Computing*, pages 1438–1445, 2004.
13. J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

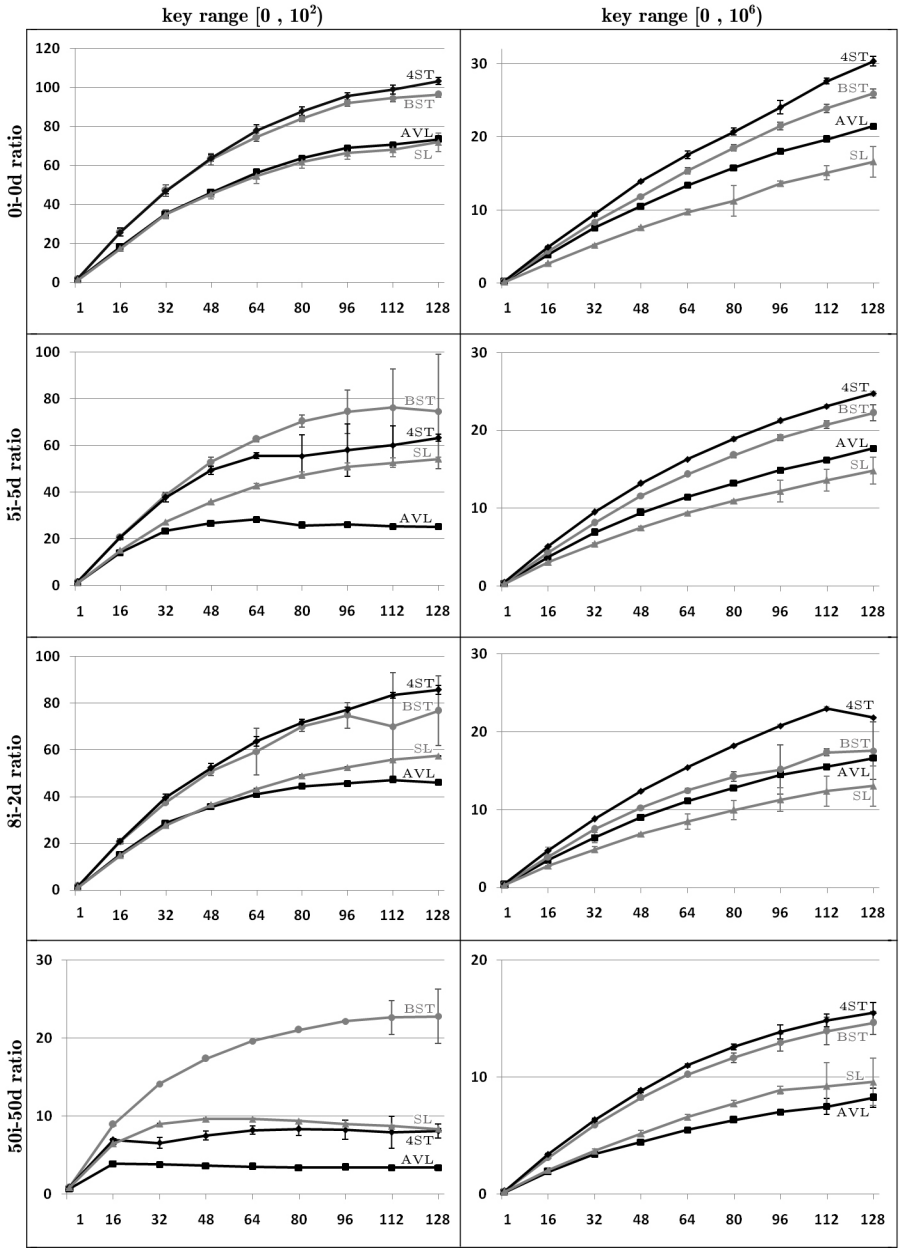

Fig. 6. Experimental results. Error bars are drawn to represent one standard deviation from the mean. Columns display ranges from which random keys are drawn. Rows display ratios of INSERTs to DELETEs to FINDs. The y-axis displays average throughput (millions of operations/s), and the x-axis displays the number of h/w threads.

5 Appendix A

5.1 Proof of Correctness

Since the k-ST algorithm is a generalization of the BST algorithm of Ellen et al. [5], it is not too difficult to adapt the proof presented in [5] to show that the k-ST is correct, and we include this adaptation for the sake of completeness. The proof presented in this section is identical to the proof presented in [5], with only minor changes. It is reproduced in full, for ease of reference, with permission from our supervisors, who were co-authors of [5]. We preserve the numbering of the results used in [5], and verify that each theorem of the BST proof carries over in the k-ST case. Before we begin the proof, there are some minor stylistic differences between the BST and the k-ST that should be mentioned.

The BST uses an Update object that is an ordered pair, consisting of a *state* field that contains one of CLEAN, MARK, IFLAG or DFLAG, and an *info* field that contains the information necessary to allow any process to help the operation complete. The k-ST instead encodes the *state* field in the type of the object, opting to have the corresponding Clean, Mark, ReplaceFlag and PruneFlag as subtypes of the UpdateStep type. This change, moving the state bits from the field into the object type, facilitates a somewhat simpler implementation in Java and does not affect correctness. Any time it affects the proof, we describe the modifications.

The BST places three sentinel nodes at the top of the tree, with $root$ and $root.right$ having key ∞_2, and $root.left$ having key ∞_1, where $\infty_1 < \infty_2$. This guarantees each leaf that has a key different from ∞_1 or ∞_2 will have a non-NULL parent and grandparent in the tree, while preserving the search-tree property. Further, the ∞ keys cannot be provided as the argument to the INSERT or DELETE routines, so the root cannot be changed. The k-ST instead uses a single ∞ value, and uses empty leaves instead of leaves with ∞ keys. The use of a single ∞ value does not preserve the search-tree property at the root, but this is not important, since the "real tree" begins at the leftmost grandchild of the root, and searches for $key \neq \infty$ will always descend left from any node with key ∞, so a search can only descend left from $root$, and left from the leftmost child of the root ($root.c1$), to enter the real tree, where the search-tree property *is* preserved. This required some changes to Lemma 22 and, indirectly, to Lemma 29. The empty children of $root$ and $root.c1$ guarantee that $root$ and $root.c1$ always have at most one non-empty child. This ensures pruning deletion can never apply to these nodes. Further, since ∞ is not a regular key, it cannot be an argument to DELETE, so simple deletion

can never apply to these nodes either. Since *root* and *root.c*1 are internal nodes, neither type of insertion operation can apply to them. This allows us to prove in Lemma 2 that *root* and *root.c*1 are never removed from the tree, and that *root.c*1 points to the same node at all times, maintaining the sentinel nodes at the root of the tree.

Also note that pruning deletion is identical to deletion in the BST, removing a leaf and its parent when that leaf has only a single non-empty sibling, by changing the appropriate child pointer of the leaf's grandparent to point to the sibling. Where insertion in the BST replaces a leaf by a new internal node and two new leaves, sprouting insertion replaces a leaf by a new internal node and k new leaves. This is, of course, a natural generalization of the BST case where $k = 2$. Simple insertion and simple deletion are very similar, replacing a leaf with a single new node. Lemma 22, 24, 25, 29 and 36, and Theorem 37(1) were updated to incorporate the new "simple" operations.

These new operations do not interfere with the coordination scheme between updates, since they can be viewed as insertions or deletions in the BST. Indeed, the coordination scheme in the two implementations is identical. The k-ST does not even require the addition of any new types of flags. ReplaceFlag and PruneFlag are simply aliases of IFlag and DFlag, respectively. The only difference between this set of operations, and the operations presented in the BST, is the new "replacement" node that will be added into the tree by simple insertion, simple deletion and sprouting insertion, but this replacement always satisfies the same properties as its analogue in the BST, being the root of a sub-tree of newly created nodes that have never before appeared in the tree. Unlike IFlag objects, which could only be stored by Insert operations in the BST, ReplaceFlag objects can be stored both Insert and Delete operations in the k-ST, requiring small changes to Lemma 22, 24 and 25, and Theorem 37(1).

Having up to $k - 1$ keys in a node (in separate fields) could be cause for concern but, since they are final and immutable after the creation of a node, it is a simple task to inspect keys in sequence, or count the number of non-∞ keys, without concurrency issues.

Similarly, the many child references of a node are protected in the same way that the two child references of a node in the BST are protected. Flagging or marking a node prevents all child pointers from changing, just as it did in the BST. Thus, the child pointers of a node can safely be inspected in sequence without experiencing concurrency issues, once a node is flagged or marked, as described in Sec. 2.6.

Another notable difference lies in how the k-ST selects the reference to be changed by a *child CAS*. The BST selected this reference by comparing the key of the node to be written into the tree with the key of the node whose child reference would be changed by the *child CAS*. The k-ST instead determines the index of the child reference that should be changed by an update while it is traversing the tree in SEARCH, and stores this index in an UpdateStep object for later use. This required modifying Lemma 7, and simplified the statement of Lemma 2(8).

A final difference that required special accommodation was the necessity in the k-ST to show that when a *child CAS* occurs for a pruning deletion, the leaf from which a key is being deleted has exactly one key, and exactly one non-empty sibling. (In the BST, every leaf has exactly one key and one non-empty sibling, so this could be taken for granted.) This added some small complications to the statement and proof of Corollary 16.

Any time that these differences manifest themselves in the proof that follows, they will be highlighted and their correctness justified. We can now begin proving the correctness of our k-ST implementation, mirroring the structure of the BST proof.

Basic Invariants and Preconditions

Observation 1 is amended to incorporate many keys, and the *keyCount* field.

Observation 1. *No key field $a_1, ..., a_{k-1}$ of a Node ever changes. No field of an UpdateStep object ever changes. The root pointer never changes. The keyCount field of a Leaf never changes. The pending field of an Internal node is never* NULL.

For the last point of Observation 1, note that *pending* fields initially contain a Clean object, and can only be written at line 60, 88, 97, 105, 112, 117, or 122. The first three of these write an object that was just created at the previous line. The remaining four write an object that is created in the same line. Thus, NULL can never be written into a *pending* field.

Lemma 2 is amended to incorporate the absence of a CAS-CHILD routine (Claim 5), multiple keys per node, the single infinite key and the revised search-tree property (Claim 8), the fact that Claim 9 is no longer needed, the modified top of the tree (Claim 10), and the fact that

f.newChild may now point to a leaf, so we can no longer claim it is internal (Claim 11).

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

1. *Each call to* HelpReplace *satisfies its preconditions.*
2. *Each call to* HelpPrune *satisfies its preconditions.*
3. *Each call to* HelpMarked *satisfies its preconditions.*
4. *Each call to* Help *satisfies its preconditions.*
5. *Each child CAS has non-*Null *arguments.*
6. *Any* Search *that has executed line 29 has a non-*Null *value in its local variable* $leaf$.
7. *Each call to* Search *that terminates satisfies its postconditions.*
8. *Let* $S = \{root, root.c_1\}$, $u \notin S$ *be any node, and let* $K = u.keyCount$ *if* u *is a leaf; otherwise, let* K *be the number of keys in an internal node.*
   (a) *For* $i = 2, ..., K$, $u.a_{i-1} < u.a_i < \infty$ *(keys are ordered and finite).*
   (b) *For* $i = K + 1, ..., k$, $u.a_i = \infty$ *(unoccupied keys are infinite).*
   (c) *If* u *is internal, then none of* u*'s child pointers are* Null.
9. *(Deleted–The claim made in the BST proof is no longer necessary, since the k-ST directly encodes the state field in the subtypes of UpdateStep.)*
10. *The top part of the tree is always as shown in Fig. 2(a). More precisely:*
    (a) $root$ *and* $root.c1$ *point to internal nodes*
    (b) *For* $i = 1, ..., k - 1$, $root.a_i = root.c_1.a_i = \infty$
    (c) *For* $i = 2, ..., k$, $root.c_i$ *and* $root.c_1.c_i$ *point to empty leaves (having* $keyCount = 0$*)*
    (d) $root.c_1.c_1$ *is non-*Null
11. *For any ReplaceFlag object* f, $f.p$ *is a pointer to an internal node,* $f.newChild$ *is non-*Null*, and* $f.l$ *is a pointer to a leaf node.*
12. *For any PruneFlag object* f, $f.gp$ *and* $f.p$ *are pointers to internal nodes,* $f.l$ *is a pointer to a leaf node and* $f.ppending$ *is a value that has previously appeared in* $f.p.pending$.

Proof. As in the BST proof, Claim 8 and 10 are initially true by initialization of the data structure, and the other claims are all true for an execution of zero steps. Assume all claims hold for the first i steps of an execution. We shall show that they hold after step $i + 1$. Let this $i + 1^{th}$ step be called s, as in the BST proof. In the following, for brevity, we denote the argument x of an invocation f of some function by $f.x$.

1. Let s be an invocation of HelpReplace. Then s is an execution of line 62, 99 or 68.
   - If it is line 62, then *op* was just created at line 59, and is non-Null.
   - If it is line 99, then *op* was just created at line 96, and is non-Null.
   - If it is line 68, then a previous step of the execution called Help, passing *op* as the argument so, by Claim 4 of the inductive hypothesis, *op* was not Null when it was passed to Help, so it is not Null when it is passed to HelpReplace.
2. Let s be an invocation of HelpPrune. Then s is an execution of line 90 or 69.
   - If it is line 90, then *op* was just created at line 87, and is non-Null.
   - If it is line 69, then a previous step of the execution called Help, passing *op* as the argument so, by Claim 4 of the inductive hypothesis, *op* was not Null when it was passed to Help, so it is not Null when it is passed to HelpPrune.
3. Let s be an invocation of HelpMarked. Then s is an execution of line 108 or 70.
   - If it is line 108, then a previous step of the execution called HelpPrune, passing *op* as the argument so, by Claim 4 of the inductive hypothesis, *op* was not Null when it was passed to HelpPrune, and it is not Null when it is passed to HelpMarked.
   - If it is line 70, then a previous step of the execution called Help, passing *op* as the argument so, by Claim 4 of the inductive hypothesis, *op* was not Null when it was passed to Help, and it is not Null when it is passed to HelpMarked.
4. Let s be an invocation of Help. As in the BST proof, we show that its argument *op* has appeared in the *pending* field of an internal node in the tree, which cannot contain Null (since *pending* fields are initially Clean, as is easily verified). We now consider each of the seven places a call to Help can occur.
   - If it is line 52, then *ppending* was read from *p.pending* by Postcondition (3) of the Search at line 49, according to Claim 7 of the inductive hypothesis and, by Postcondition (1), we know that p points to an internal node.
   - If it is line 65, then *result* was read from *p.pending* at the same line. Further, we know p is an internal node by the same reasoning as the previous case.
   - If it is line 81, then *gppending* was read from *gp.pending* by Postcondition (3) of the Search at line 78, according to Claim 7 of the inductive hypothesis and, by Postcondition (1), we know that *gp* points to an internal node.

- If it is line 83, then *ppending* was read from *p.pending* by Postcondition (3) of the SEARCH at line 78, according to Claim 7 of the inductive hypothesis and, by Postcondition (1), we know that p points to an internal node.
- If it is line 92, then *result* was read from *gp.pending* at the same line. Further, we know *gp* is an internal node by Postcondition (1) of the SEARCH at line 49, according to Claim 7 of the inductive hypothesis.
- If it is line 102, then *result* was read from *p.pending* at the same line. Further, we know p is an internal node by Postcondition (1) of the SEARCH at line 78, according to Claim 7 of the inductive hypothesis.
- If it is line 111, then *result* was read from *op.p.pending* at line 106. By Claim 12, *op.p* points to an internal node.

5. Let s be a *child CAS*. Then s can only be an execution of line 116 or 121.
   - If it is line 116 then, by Claim 11 of the inductive hypothesis, *op.p* is an internal node, and *op.l* and *op.newChild* are non-NULL, either Claim 8(c) or 10(d,e) applies, so all children of *op.p* are non-NULL.
   - If it is line 121 then, by Claim 12, *op.gp* is an internal node, and *op.p* is also an internal node (and so is non-NULL). We must still show that $op.gp.c_{op.gpindex}$ and *other* are non-NULL. Since *op.gp* and *op.p* are internal nodes, by Claim 8(c) and 10(d,e), all children of *op.gp* and *op.p* are non-NULL. Thus, $op.gp.c_{op.gpindex}$ is non-NULL, and any child of *op.p* assigned to *other* is non-NULL.
6. The BST proof of this claim carries through with no modification, other than changing line 32 to line 36 and adding induction hypothesis 10 to hypothesis 8 to guarantee line 36 will read a non-NULL reference.
7. As in the BST proof, let $\langle gp_{final}, p_{final}, l_{final}, ppending_{final}, gppending_{final}\rangle$ be the final values returned by a terminating SEARCH.
   - By inductive Claim 10, we know *root.c*1 points to an internal node prior to the return step at line 38, so the initial assignments at line 29 point $leaf$ and *parent* to internal nodes. Thus, the loop will be entered at least once and, before the first iteration, $leaf$ and *parent* both point to internal nodes. As a result, the first iteration of the loop will point *gparent* and *parent* to internal nodes. It is plain to see that *gparent* and *parent* are pointed to internal nodes every iteration, since *gparent*'s value comes from *parent*, whose value comes from $leaf$, which points to an internal node at the beginning of every iteration. Further, the exit condition of the

loop ensures that when the loop is exited, $leaf$ will point to a leaf node. Thus, Postcondition (1) is satisfied.
- Before the first iteration of the loop, the assignment at line 29 will satisfy that $leaf$ has been $parent.c_{pindex}$ (since $parent.c1$ is an internal node by inductive Claim 10). By the same reasoning as in the proof of Postcondition (1), the loop will be entered at least once and, after the first iteration of the loop, *parent* will have been $gparent.c_{gpindex}$, since the values in $leaf$ and *parent* will simply shift into *parent* and *gparent*. It is easy to see that every iteration through the loop will preserve this, since an iteration simply shifts values, then points $leaf$ to $parent.c_{pindex}$. Thus, Postcondition (2) is satisfied.
- It is a simple (and uninteresting) exercise to show that Postcondition (3) is also satisfied, using the same reasoning used to prove Postcondition (2). The important facts are the following. Before the loop is entered for the first time, *parent* points to an internal node, and *ppending* was read from *parent*. The loop is entered at least once, values shift from *ppending* to *gppending* alongside the values from *parent* to *gparent*, and *ppending* is read from *parent* each iteration.

8. Parts (a) and (b) of this claim are satisfied since the keys of a node are final, immutable except at creation time, and the lines where a node is created (lines 55, 57 and 95) are defined to preserve these fundamental structural invariants.
   Part (c) is satisfied since a child pointer can only be changed by a *child CAS* at lines 116 and 121 and, by inductive Claim 5, any *child CAS* has non-NULL arguments.
9. Claim is deleted and requires no proof.
10. Note that this is initially satisfied upon creation of the top of the tree. Also observe that shared field *root* never changes, and that the keys and *keyCount* field of a node never change, by Observation 1. Thus, it simply remains to show that no child pointer of *root* can change, and that only the leftmost child pointer of $root.c1$ can change.
    We first show no child pointer of *root* can change. Say s is a *child CAS* that changes a child pointer of *root*. Then s is called from HELPMARKED or HELPREPLACE.
    - Say s was called from an invocation HELPREPLACE. Then *op* was created at line 59 or 96.
      Assume *op* was created at line 59. Then $op.p = root$ was read by the SEARCH at line 49, which satisfied its postconditions by inductive Claim 7, so $op.l$, a child of *root*, is a leaf. However, we know

that Claim 8 was inductively satisfied throughout the execution of this SEARCH, so that the keys of $op.p = root$ must all be ∞, and that the argument *key* passed to SEARCH could not have been ∞, so the last execution of line 36 had to follow the leftmost child pointer to reach leaf $op.l$. However, this yields a contradiction, since we also know by Claim 10 that the leftmost child of *root* is an internal node, not a leaf. Thus, s cannot occur.
If we assume, however, that *op* was created at line 96, then $op.p = root$ was read by the SEARCH at line 78, which satisfied its postconditions by inductive Claim 7, so $op.l$, a child of *root*, is a leaf. However, by the same argument as in the previous case, $op.l = root.c1$ must be an internal node. Thus, s cannot occur.

– Next, say s was called from an invocation of HELPMARKED. Then *op* was created at line 87. Further, $op.l$, $op.p$ and $op.gp = root$ were read by the SEARCH at line 78, and the postconditions of SEARCH were satisfied by inductive Claim 7 when it terminated. Thus, $op.l$ is a leaf, and a child of $op.p$, which is a child of $op.gp = root$. Further, $op.p$ is an internal node so, by inductive Claim 10, $op.p$ is $root.c1$, and always has exactly one non-empty child, so $ccount = 1$, and the test at line 85 must evaluate to FALSE, so it is impossible for *op* to be created at line 87 with $op.gp = root$, yielding a contradiction. Thus, s cannot occur.

We now show that only the leftmost child pointer of $root.c1$ can change. Say s is an invocation of CAS-CHILD that changes a child pointer of $root.c1$ other than the leftmost. Then s is inside an invocation of HELPMARKED or HELPREPLACE, where *op* is created at line 59, 96 or 87.

– In the case of HELPREPLACE (line 59 and 96), a child of $op.p$ will be changed, so $op.p = root.c_1$ by our assumption. Further, the field $op.l$ is read at line 36 of a SEARCH invoked at line 49 or 78 so, by inductive Claim 7, $op.l$ is a leaf, and a child of $op.p$. In particular, by our assumption, $op.l$ is not the leftmost child of $op.p$. By inductive Claim 10, we also know that the keys of $op.p$ are all ∞ throughout the execution of SEARCH. Also note that it takes one iteration of the loop to assign $op.p := root.c_1$, so the following statement is well-formed. The last time line 36 of SEARCH was executed, the key *key* must have been greater than or equal to ∞ or else the SEARCH would have descended to the leftmost child. However, ∞ cannot be an argument to SEARCH, yielding a contradiction.

– In the case of HelpMarked (line 87), a child of $op.gp$ will be changed from $op.p$ to $other$, so $op.gp = root.c_1$ by our assumption. Further, the fields $op.l$ and $op.p$ are read by a Search invoked at line 78 so, by inductive Claim 7, $op.l$ is a leaf, and a child of $op.p$, which is a child of $op.gp$. In particular, by our assumption, $op.p$ is not the leftmost child of $op.gp$. As in the previous case, by inductive Claim 10, all keys of $op.gp$ are ∞ throughout the execution of Search. Note that it takes two iterations through the loop in Search for $root.c_1$ to reach variable $gpparent = op.gp$. Thus, the Search will iterate through the loop at least twice and, the second last time line 36 was executed (when the eventual value of $op.p$ was read from a child of $root.c_1$), it must have decided the appropriate child was not the leftmost, so key key must have been greater than or equal to ∞, the leftmost key of $root.c_1$. However, ∞ cannot be an argument to Search, so this is a contradiction. Thus, s cannot occur.

11. Since all fields of an ReplaceFlag object are final, and immutable after creation, we need only verify that this property holds at creation time. ReplaceFlag objects are only created at line 59 or 96.
    Say a ReplaceFlag object f is created at line 59. Then $f.newChild$ is created at line 55 or 57. Either way, it is a new, non-Null node. Additionally, $f.l$ and $f.p$ are read by the Search at line 49 and, by inductive Claim 7, the postconditions of Search are satisfied when this Search terminates. Thus, $f.l$ is a leaf, and $f.p$ is an internal node.
    Say a ReplaceFlag object f is created at line 96. Then $f.newChild$ is created at line 95, and is non-Null. Additionally, by the same reasoning as in the previous case, $f.l$ and $f.p$ are read by the Search at line 78, and $f.l$ is a leaf, while $f.p$ is an internal node.
12. Since all fields of an PruneFlag object are final, and immutable after creation, we need only verify that this property holds at creation time, i.e., at line 87.
    Say a PruneFlag f is created at line 87. Then $f.l$, $f.p$, $f.gp$ and $f.ppending$ are all returned by the Search at line 78 and, by inductive Claim 7, the postconditions of Search hold when the Search terminated, so $f.l$ points to a leaf, $f.p$ and $f.gp$ point to internal nodes, and $f.ppending$ was contained in $f.p.pending$ at some point. □

Behaviour of CAS steps on *pending* Fields

We first make some definitions to align with the nomenclature of the BST proof; they remain in effect for the remainder of the proof.

Definition 1. *Rflag CAS (standing for ReplaceFlag CAS) and Punflag CAS (standing for PruneFlag CAS) steps are executions of lines 60 and 97, respectively.*

Runflag CAS (standing for ReplaceUnflag CAS) and Punflag CAS (standing for PruneUnflag CAS) steps are executions of lines 117 and 122, respectively.

Rchild CAS (standing for ReplaceChild CAS) and Pchild CAS (standing for PruneChild CAS) steps, are executions of lines 116 and 121, respectively.

The Backtrack CAS steps are executions of line 112.

(The mapping to the naming conventions of the BST proof is: Rflag CAS = Iflag CAS, Pflag CAS = Dflag CAS, Runflag CAS = Iunflag CAS, Punflag CAS = Dunflag CAS, Rchild CAS = Ichild CAS, and Pchild CAS = Dchild CAS.)

Note that, unlike the BST, in the k-ST algorithm, *Rflag CAS*, *Runflag CAS* and *Rchild CAS* steps are invoked by both Insert and Delete (where they were only invoked by Insert in the BST). This is due to the similarity of simple deletion to insertion, which allows it simply to use ReplaceFlag objects and the HelpReplace function.

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

1. *When v is created, v.pending points to a Clean object.*
2. *Flag CAS and mark CAS steps on v.pending succeed only if it points to a Clean object.*
3. *Runflag CAS steps on v.pending succeed only if it points to a ReplaceFlag object.*
4. *Punflag CAS and Backtrack CAS steps on v.pending succeed only if it points to a PruneFlag object.*
5. *Once v is marked, its pending field never changes.*

Proof. As in the BST proof, we prove each statement in turn.

1. This true for the two internal nodes initially in the tree, by line 27. Elsewhere, internal nodes are only created at line 55, where their *pending* fields are initialized to point to Clean objects.
2. *Flag CAS* and *mark CAS* steps occur at lines 60, 88, 97, and 105. If it is line 60, then the expected value *ppending* for the CAS is Clean by the test at line 51.

If it is line 88, then the expected value *gppending* for the CAS is Clean by the test at line 80.
If it is line 97, then the expected value *ppending* for the CAS is Clean by the test at line 82.
If it is line 105, then the expected value for the CAS is *op.ppending*. The argument *op* could only have been created at line 87, so *op.ppending* = *ppending* is Clean by the test at line 82.

3. *Runflag CAS* steps occur only at line 117. Since the expected value for the CAS is *op*, which is a ReplaceFlag object, this claim is trivially satisfied.
4. *Punflag CAS* and *Backtrack CAS* steps occur only at lines 112 and 122. In either case, the expected value for the CAS is *op*, which is a PruneFlag object, satisfying the claim.
5. By part (2) of this Lemma, we know that no *flag CAS* or *mark CAS* can change *v.pending* once it points to a Mark object. By parts (3) and (4), we know that no *Runflag CAS*, *Punflag CAS* or *Backtrack CAS* can succeed. This accounts for all CAS steps that can change *v.pending*. □

Note that, as in the BST proof, we have assumed no garbage collection for simplicity. However, correctness is not affected if the algorithm is implemented in a system with safe garbage collection, meaning that no memory address is allocated or de-allocated as long as it is reachable from any user thread.

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

Proof. The *pending* field of v can be changed only by a CAS step at line 60, 88, 97, 105, 112, 117 or 122. In each of these cases, the new UpdateStep object to be written by the CAS step is created locally by the process executing the CAS, immediately before the CAS step, either in-line, or at the previous line. Thus, in each case, the new value to be written cannot have previously appeared in a *pending* field. □

As in the BST proof, we define what it means for CAS steps to *belong* to an UpdateStep object. The definition in the BST proof requires minimal modification to accommodate the fact that the State and Info types of the BST have been merged to produce UpdateStep objects.

Definition 2. *If a flag CAS stores a pointer to a ReplaceFlag or PruneFlag object f, then we say the flag CAS belongs to f. Mark, child, unflag*

and backtrack CAS steps all use information from an UpdateStep object that is the argument to the HelpPrune*,* HelpReplace *or* HelpMarked *invocation that performs the CAS step. Each of these CAS steps is also said to belong to the UpdateStep object. When an UpdateStep object is created, some information comes from an invocation of* Search*, which is also said to belong to the UpdateStep object.*

Lemma 5. *Only the first mark CAS belonging to a PruneFlag object can succeed.*

Proof. As in the BST proof, let f be a PruneFlag object, and $mcas$ be a successful *mark CAS* belonging to f. We show $mcas$ must be the first *mark CAS* that belongs to f. Let $mcas'$ be another successful *mark CAS* belonging to f, and preceding $mcas$, to derive a contradiction. Both $mcas$ and $mcas'$ attempt to change $f.p.pending$ from $f.ppending$ to a new Mark object. By the postconditions of the Search belonging to f, $f.ppending$ was read from $f.p.pending$ during the Search (at line 78), before f was created (at line 87) and, in turn, before $mcas'$. In order for $mcas$ to succeed, $f.p.pending$ must be $f.ppending$ when it occurs so, by Lemma 4, $f.p.pending$ must have been $f.ppending$ when $mcas'$ was executed (before $mcas$). Thus, $mcas'$ must have succeeded but, by Lemma 3, $f.pending$ can never again change after $mcas'$ has marked it, meaning $mcas$ cannot succeed, contradicting its definition. □

Lemma 6. *If a successful mark CAS belongs to a PruneFlag object f, then no Backtrack CAS belongs to f.*

Proof. Let f be a PruneFlag object. Assume a successful *mark CAS* belongs to f. Let H be an invocation of HelpPrune whose argument is f. By Lemma 5, either H's *mark CAS* succeeds, or an earlier *mark CAS* belonging to f succeeded, and permanently stored a Mark object m in $f.p.pending$, satisfying $m.pending = f$ (by Definition 2), which will be read at line 106 by H. Either way, the test at line 107 will evaluate to True, and H will not attempt a *Backtrack CAS* at line 112. □

Behaviour of Child CAS Steps

Lemma 7. *Let ccas be a child CAS belonging to some UpdateStep object op and let R be the register it attempts to change. Let r be the last or second-last execution of line 36 in the* Search *belonging to f, depending on whether ccas is an Rchild CAS or Pchild CAS, respectively. Then, r reads R and the value read by r is the expected value used for ccas.*

Proof. We consider two cases.

Case 1: *ccas* is an *Rchild CAS*. We note that r is well defined, since the SEARCH belonging to *ccas* performs at least one iteration, since $leaf = parent.c1$ points to an internal node at line 29, by Lemma 2(10). The SEARCH belonging to *ccas* executes r and saves the result in $op.l$, which will be used as the expected value for *ccas*, proving the second half of the claim.

Step *ccas* modifies the field $op.p.c_{op.pindex}$ which, according to the postconditions of the SEARCH belonging to *op*, contained $leaf = op.l$ at some point. Noting that *pindex* is set only at line 36, the same place that l is read (and where r occurs), we see that $op.l$ was read from $op.p.c_{op.pindex}$ by r, so the claim is proven.

Case 2: *ccas* is a *Pchild CAS*.

We now show that r is well defined. We first note that, by the same reasoning as in the previous case, the loop in the SEARCH S belonging to *ccas* must perform at least one iteration. To obtain a contradiction, suppose that the loop performs only one iteration, returning $p = root.c_1$. Then, by Lemma 2(10), the test at line 86 must evaluate to FALSE, since $p = root.c_1$ always has exactly one non-empty child, so *ccas* cannot be a *Pchild CAS*, yielding a contradiction. Thus, S enters the loop at least twice, so r is well defined.

When S executes r, the second-last time through the loop, it will read a child of $op.gp$ and, eventually, this child reference will be saved in $op.p$, which will be used as the expected value for the CAS, satisfying the second half of the claim. By the postconditions of S, the index of this child of $op.gp$ will be stored in $op.gpindex$, such that $op.gp.c_{op.gpindex} = R$ is the reference that was read by r. Clearly this is also the reference that will be modified by the *ccas*, satisfying the first half of the claim. □

Definition 3. *As in the BST proof, we define $ccas_1, ccas_2, ...$ to be the ordered sequence of successful child CAS steps in the execution, f_i to be the UpdateStep object to which $ccas_i$ belongs, $fcas_i$ to be the successful flag CAS that belongs to f_i, $pending_i$ is the field changed by $fcas_i$, and $ucas_i$ be the CAS that next changes the pending field flagged by $fcas_i$, after $fcas_i$, if such a change occurs. The definitions made in Lemma 7 are also extended. R_i is the register changed by $ccas_i$, and r_i is the last or second-last execution of line 36 in the* SEARCH *belonging to f_i, depending on whether $ccas_i$ is an Rchild CAS or a Pchild CAS, respectively.*

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

Proof. This corollary follows directly from Lemma 7, as in the BST proof. □

Mimicking the BST proof, the remaining lemmata of this section are devoted to showing that the CAS steps proceed in an orderly fashion, with a *child CAS* step coming after the corresponding *flag CAS* (and possibly *mark CAS*), and before a *Backtrack CAS* or *unflag CAS* and, also, that at most one *child CAS* belongs to an UpdateStep object, that *child CAS* steps avoid the ABA problem, and *child CAS* steps maintain the invariant that the structure is a full k-ary tree.

Lemma 9. *Each mark CAS that belongs to an UpdateStep object f is preceded by a successful Pflag CAS that belongs to f.*

Proof. Let *mcas* be any *mark CAS* belonging to f. Then *mcas* is an execution of line 105 in an invocation H of HelpPrune which is, in turn, called from line 90 or 69.

- If H was called from line 90, then the *Pflag CAS* at line 88 was clearly successful.
- If H was called from line 69, then $H.op$ is non-Null, and has appeared in the *pending* field of some internal node in the tree (by the preconditions of Help). Further, it is of type PruneFlag, by the test at line 69. Thus, the PruneFlag $H.op$ must have been written into the *pending* field of some internal node prior to the invocation H of HelpPrune, by a *Pflag CAS* at line 88.

Thus, in both cases, a successful *mark CAS* follows a successful *Pflag CAS*. □

Lemma 10. *Each Pchild CAS that belongs to an UpdateStep object f is preceded by a successful mark CAS that belongs to f.*

Proof. Let *ccas* be any *Pchild CAS* belonging to f (where f is necessarily a PruneFlag object). Then *ccas* is executed at line 121 by an invocation H of HelpMarked. This invocation H is called at line 108 by an invocation D of HelpPrune. Then $H.op = D.op = f$ by Definition 2. According to the test at line 107, which must evaluate to True (since line 108 is executed), either the *mark CAS* at line 105 succeeded (if *result* was True), or *op.p.pending* already contained a Mark object m, satisfying $m.pending = op$, which can only have been stored by a successful *mark CAS mcas*, executed at line 105. By Definition 2, $m.pending = op$ means that *mcas* belongs to f. Thus, in each case, *ccas* is preceded by a successful *mark CAS*. □

Lemma 11. *If a Pchild CAS belongs to a PruneFlag object f, no Backtrack CAS belongs to f.*

Proof. This carries through from the BST proof without modification. □

Lemma 12. *Each child CAS that belongs to an UpdateStep object 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 *Rchild CAS*, then it is executed at line 116 of HelpReplace, which is called from line 62, 99 or 68. By Definition 2, the argument *op* to HelpReplace is actually f. Lines 62 and 99 are executed immediately after their respective *Rflag CAS* steps at lines 60 and 97. Since these *Rflag CAS* steps write $op = f$, they belong to f, by Definition 2. Line 68 is called from within Help, which satisfies its preconditions, so its argument *op* is a non-Null ReplaceFlag object that has appeared in the *pending* field of an internal node, where it was stored by a successful *Rflag CAS*. Since this *Rflag CAS* wrote $op = f$, it belongs to f.
- It *ccas* is a *Pchild CAS* then, by Lemma 9 and Lemma 10, *ccas* comes after a successful *flag CAS* belonging to f. □

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

Proof. This carries through from the BST proof without modification. □

Lemma 14. *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 Rchild CAS, then $ccas_i$ writes a reference to the root of a finite, full k-ary tree consisting of new nodes that have never appeared in the data structure before (i.e., have never before been reachable from the root by following child references).*

4. *If $ccas_i$ is a Pchild CAS, then just prior to $ccas_i$, R_i contains $f_i.p$. If $ccas_i$ is an Rchild CAS, then just prior to $ccas_i$, R_i contains $f_i.l$.*
5. *If $ccas_i$ is a Pchild CAS, then $f_i.p$'s children do not change between the last execution of line 36 within the* SEARCH *belonging to f_i and $ccas_i$.*
6. *After $ccas_i$, the data structure is a full k-ary tree. (That is, every internal node that can be reached by following child references from root has exactly K children, where K is the degree of internal nodes, there are no cycles in the child references among these nodes, and there is at most one path of child references to any node from the root.)*
7. *$ccas_i$ writes a value into R_i that has never been stored there before.*

Proof. As in the BST proof, we prove this lemma using strong induction on i, and let k be a positive integer such that $ccas_k$ exists. We assume the claims are true for $ccas_i$ when $1 \leq i < k$, and prove they hold for $ccas_k$.

1. The proof of this claim carries through without modification from the BST proof.
2. The first two paragraphs of the BST proof of this claim require modification, and are given below. The rest of the proof carries through without modification.
   Suppose that $ucas_k$ (the first CAS step that succeeds in changing $pending_k$ after $fcas_k$) exists. It changes $pending_k$ from a ReplaceFlag or PruneFlag object f to something else (since $fcas_k$ writes f into $pending_k$). By Lemma 3, the only types of CAS steps to do this are *unflag CAS* or *Backtrack CAS* steps. Since the old value of $pending_k$ used by $ucas_k$ is f_k (since it succeeds, immediately following $fcas_k$), $ucas_k$ must belong to f_k.
   We first show that $ucas_k$ cannot be a *Backtrack CAS* belonging to f_k. If $ccas_k$ is an *Rchild CAS*, then there cannot be a *Backtrack CAS* belonging to the ReplaceFlag object f_k. If $ccas_k$ is a *Pchild 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), and $ucas_k$ is an *unflag CAS* that belongs to f_k.
3. If $ccas_k$ is an *Rchild CAS*, then it writes $f_k.newChild$, which refers to a node x created by the INSERT or DELETE (performing simple deletion) that executed $fcas_k$. When x is created, it is either a new leaf, or a new internal node with K newly created leaves $x.c_1, ..., x.c_K$ (where K is the degree of internal nodes in the k-ST). Thus, x is the root of a finite, full k-ary tree. None of these nodes can be reachable

from *root* until some *child CAS* changes a child reference to point to one of them. We consider all successful *child CAS* steps prior to $ccas_k$ and prove that none of them write pointers to any of these nodes.
To derive a contradiction, assume some *Rchild CAS* or *Pchild CAS* before $ccas_k$ writes a reference to x, or to some $y \in C = \{x.c_1, ..., x.c_K\}$. Consider the first *child CAS*, $ccas_j$ that writes a reference to x or y. We consider two cases.

- If $ccas_j$ is a *Pchild CAS*, then the value written by $ccas_j$ is read from a child reference of $f_j.p$ at line 120. Thus, $ccas_j$ could not have written a reference to x, because no child reference refers to x before $ccas_j$ (by definition of $ccas_j$), so it must write a reference to some $y \in C$. Then the process that performed $ccas_j$ had previously read a reference to y in a child field of the node that $f_j.p$ points to (at line 120). Thus, $f_j.p$ could only point to x since, prior to $ccas_j$, the only child reference that points to y belongs to the node x. So, one of the child references of the node that $f_j.gp$ points to contained a reference to x before the end of the SEARCH belonging to f_j, according to Postcondition (2) of the SEARCH, and so before $ccas_j$, but this is impossible, since no child reference refers to x before $ccas_j$.
- Suppose $ccas_j$ is a *Rchild CAS*. Only an *Rchild CAS* belonging to f_k can write a reference to x and, by Claim 1, there is no such *Rchild CAS* before $ccas_k$. Thus, $ccas_j$ cannot write a reference to x, so it must write a reference to some $y \in C$ (which implies x must be an internal node). Then f_k is created at line 59, and $f_k.newChild$ is created at line 55. Further, only $ccas_j$ belonging to $f_j \neq f_k$ such that $f_j.newChild = y$ can write a reference to y. (We know $f_j \neq f_k$, since we have assumed both $ccas_j$ and $ccas_k$ succeed and, if $f_j = f_k$ were true, then $ccas_k$ would not be the first successful *child CAS* belonging to f_k, which would contradict Claim 1.) Suppose that such an f_j exists, to derive a contradiction. Then $f_j.newChild$ is created at line 59 or 96. In either case, $f_j.newChild = y$ is newly created, a few lines prior. Thus, y is newly created (in freshly allocated memory) when f_k is created, and again when f_j is created. Without loss of generality, assume that $f_j.newChild$ is created before $f_k.newChild$. Then, since $f_k.newChild$ must be allocated a different memory address from $f_j.newChild$, it must be the case that $y = f_j.newChild \neq f_k.newChild = y$, which is a contradiction. Thus, such a $ccas_j$ cannot occur.

4. The proof of this claim carries through without modification from the BST proof.
5. The proof of this claim carries through from the BST proof with the following substitutions.
   - "Line 32" → "Line 36"
   - "Line 31" → "Line 34"
   - "x's *state* must be CLEAN" → "x.*pending* must refer to a Clean object"
   - "*update*" → "*pending*"
6. The proof of this claim carries through from the BST proof with the slight modification that the phrase "three new nodes," must be corrected to read "new nodes."
7. The proof of this claim carries through from the BST proof if "left child" is simply interpreted to mean "leftmost child." □

The following two corollaries of Lemma 14 describe the effects of a successful *Rchild CAS* or *Pchild CAS*.

Corollary 15. *For all i, if $ccas_i$ is an Rchild CAS, it changes a child reference of $f_i.p$, replacing $f_i.l$, which is a reference to a leaf, by a reference to the root of a full k-ary tree consisting of new nodes that have never before appeared in the data structure.*

Proof. The proof of this Lemma carries through without modification from the BST proof. □

The following corollary requires extra care in its restatement. In the BST, all nodes had one key, and internal nodes had two children, so pruning deletion was always the correct operation to apply. However, in the k-ST we must demonstrate that, when a *Pchild CAS* occurs, the leaf to be deleted from has one key, and precisely one non-empty sibling.

Corollary 16. *Consider any successful Pchild CAS, $ccas_i$. When line 36 was last executed by the* SEARCH *belonging to f_i, $f_i.p$ had exactly two non-empty children: $f_i.l$ and ps, where $f_i.l$ is a leaf with one key. Then ps and $f_i.l$ are still the only non-empty children of $f_i.p$ just before $ccas_i$, and $ccas_i$ changes a child reference of $f_i.gp$ from $f_i.p$ to ps.*

Proof. The *Pchild CAS* $ccas_i$ is executed by HELPMARKED, which is called from HELPPRUNE, which has a PruneFlag object *op* as its argument. This PruneFlag object was created at line 87 in DELETE, and the test at line 86 evaluated to TRUE. Thus, $ccount = 2$ was the count of non-empty children of $op.p$, and $op.l$ was one of them, having exactly one key.

By Lemma 14(5), and the fact that the keys of a node never change, this has been true since the last execution of line 36 by the SEARCH belonging to f_i, and will remain true until $ccas_i$ is executed. (In particular, that the children of $op.p$ have not changed during this time guarantees that *ccount* was the correct count of non-empty children when the test at line 86 was evaluated, and is correct just before $ccas_i$ is executed.) Thus, just before $ccas_i$, line 120 sets *other* to *ps* (still the only non-empty sibling of $op.l$), and the claim follows from the code of line 121. □

Tree Properties

As in the BST proof, having proved that the child pointers form a full k-ary tree (Lemma 14), we now prove some properties.

Definition 4. *An internal node is called* ***inactive*** *when it is first created. It becomes* ***active*** *when a successful Rchild CAS writes a pointer to it for the first time, and remains active from that point on. The sole exception is the root node, which is active from the beginning of the execution.*

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

Proof. The proof of this lemma carries through from the BST proof, except for the last paragraph, which requires some modification to account for the fact that internal nodes now have $K \geq 2$ children and $K - 1$ keys. The modified paragraph follows.

Suppose $ccas_i$ is a *Pchild CAS*. By Corollary 16, $ccas_i$ changes a child of $f_i.gp$ from $f_i.p$ to the only non-empty sibling of $f_i.l$. Thus, the only nodes that become unreachable as a result of this CAS are $f_i.p$, the subtree rooted at $f_i.l$, and any empty siblings of $f_i.l$. Since $f_i.l$ is a leaf, and the empty siblings of $f_i.l$ can only be leaves (since internal nodes cannot be empty), and $f_i.p$ is marked when $ccas_i$ occurs (by Lemma 10), the claim follows. □

The following definition must be altered slightly from the one made in the BST proof to incorporate the many children of an internal node.

Definition 5. *A node x is an i-descendant of internal node y if, in configuration C, x is in the tree and x is a descendant of $y.c_i$.*

The following lemma implies nodes cannot acquire new ancestors.

Lemma 18. *If x is an i-descendant of y at some configuration C, then x is an i-descendant of y in all configurations between the first time x is in the tree and C.*

Proof. The proof of this lemma carries through from the BST proof with the following substitutions.

- "left-descendant" → "1-descendant"
- "right-descendants" → "other i-descendants"
- "removes two nodes" → "removes K nodes (where K is the degree of internal nodes)" □

Definition 6. *For any configuration C, let T_C be the K-ary tree formed by the child references 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 K-ST search procedure. (Although we have no yet proved that T_C is always a K-ST, we can still define this search path just by looking at what a K-ST search would do if it were run on T_C. The path of a sequential K-ST search is well defined and unique because the keys of a node are sorted, by Lemma 2(8).)*

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 K be the degree of internal nodes, and let y be any ancestor of x in $T_{C'}$. If x is an i-descendant of y in configuration C', then x is an i-descendant of y in configuration C, by Lemma 18. Since x is both on the search path for k, and an i-descendent of y in configuration C, we know that $k \geq y.a_{i-1}$ (if $i > 1$) and $k < y.a_i$ (if $i < K$). Thus, if a traversal of $T_{C'}$ seeking key k decides which direction to go at each ancestor y of x based on a comparison of k with the keys of y, it will always choose to go towards the node x. Thus, x is on the search path for k in C'. □

Proof of Linearizability

In this section we prove that the implementation is linearizable. 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. The linearization points are as defined in Sec. 2.6. Intuitively, we define the linearization point for SEARCH(k) at a point when the leaf it returns was

on the search path for k. FIND(k) is linearized at the same time as the SEARCH that it performs. Each update operation that returns TRUE is linearized at the *child CAS* that it performs. Each update operation that returns FALSE is linearized at the SEARCH that it performs.

First, we define linearization points for SEARCH(k). As in the BST proof, we would like to define the linearization point so that the leaf returned by SEARCH is on the search path for k at the time that it is linearized. We can prove, using the following lemma, that the leaf was in the tree at some time during the execution of the SEARCH.

Definition 7. *We say a* SEARCH ***enters*** *a node when a pointer to this node is assigned to leaf on line 29 or line 36.*

Lemma 20. *Let $v_1, v_2, ...,$ 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. This proof carries over from the BST proof with the following substitutions.

- "line 32" → "line 36"
- "left child of the root" → "$root.c_1$ or $root.c_1.c_1$"
- "∞_1" → "∞"
- "the left child if $k < v_{j-1}.key$ and the right child otherwise" → "child $v_{j-1}.c_i$ such that $k \geq v_{j-1}.a_{i-1}$ (if $i > 1$) and $k < v_{j-1}.a_i$ (if $i < K$, where K is the degree of internal nodes)" □

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, as in the BST proof, 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 reference l to a leaf, and that leaf is on the search path for K at the linearization point of the* SEARCH*.*

Definition 8. *The k-ST property states the following (where k is the degree of internal nodes).*

- *for each leaf node x, $x.a_1 < x.a_2 < ... < x.a_{x.keyCount}$*

– *for each internal node x, $x.a_1 < x.a_2 < ... < x.a_{k-1}$, and the keys of every i-descendant are greater than or equal to key $x.a_{i-1}$ (if $i > 1$), and smaller than key $x.a_i$ (if $i < k$).*

Now that we know that each SEARCH ends at the "correct" leaf of the k-ST, we can prove that the k-ST property is an invariant of the data structure.

Definition 9. *Let the tree of child references rooted at $root.c_1.c_1$ be called the real tree.*

The *real tree* is so named because it excludes the sentinel nodes at the top of the tree and, as is proved in the next Lemma, contains any non-∞ keys that are in the tree.

Lemma 22. *In every configuration, the real tree is a k-ST, and if any non-∞ keys appear in the tree, they must appear in the real tree (and nowhere else).*

Proof. First, note that any non-∞ keys will always fall in the sub-tree rooted at $root.c_1.c_1$ since, according to Lemma 2(10), the keys of $root$ are always ∞, the children of $root$ are always $root.c_1$ and empty leaves, the keys of $root.c_1$ are always ∞, and all children $root.c_1.c_i \neq root.c_1.c_1$ are empty leaves. Thus, a non-∞ key must appear in the sub-tree rooted at $root.c_1.c_1$, so the second half of the claim is proved. It remains only to show that the *real tree* is always a k-ST.

The *real tree* is a k-ST in the initial configuration, since $root.c_1.c_1$ is initially an empty leaf, with no keys. Since keys stored in nodes never change, and the keys of nodes are properly ordered by Lemma 2(8), we need only check that every *child CAS* preserves the invariant.

Assume the *real tree* is a k-ST prior to a *Pchild CAS*. By Corollary 16, the *Pchild CAS* replaces a sub-tree of the tree by a sub-tree of that sub-tree, so the k-ST property holds after the *Pchild CAS*.

Now consider an *Rchild CAS*, and let C be the configuration just prior to it. Assume the *real tree* is a k-ST in C. We show that it it is still a k-ST just after the *Rchild CAS*. Let f be the ReplaceFlag object to which the *Rchild CAS* belongs, and let INSERT(k) or DELETE(k) be the operation O that created f. The *Rchild CAS* changes a child reference of $f.p$ from $f.l$ to $f.newChild$, which falls into one of three cases.

Case I: O is a simple insertion, and $f.newChild$ refers to a leaf with keys (keys of $f.l$) $\cup \{k\}$, properly ordered (by $f.newChild$'s construction at line 57). Thus, we must simply check that $f.l$ is on the search path

for both k and the keys of $f.l$ in C. Since $f.l$ is in the tree in C, and the tree satisfies the k-ST property in C, $f.l$ is definitely on the search path for any of its keys 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.

Case II: O is a simple deletion, and $f.newChild$ refers to a leaf with keys (keys of $f.l$) $\setminus \{k\}$, properly ordered (by $f.newChild$'s construction at line 95). Since k is removed from the tree by the *Rchild CAS*, we must simply check that $f.l$ is on the search path for any of its other keys in C. As in the previous case, since $f.l$ is in the tree in C, and the tree satisfies the k-ST property in C, $f.l$ is definitely on the search path for any of its keys in C.

Case III: O is a sprouting insertion. Let K be the degree of internal nodes. Then $f.newChild$ refers to an internal node with the $K-1$ largest keys in $S =$ (keys of $f.l$) $\cup \{k\}$, and K children, ordered according to the k-ST property, each having one key in S (by $f.newChild$'s construction at line 55). Thus, we must simply check that $f.l$ is on the search path for both k and the keys of $f.l$ in C, but this follows from the reasoning used in Case I. □

Next, we construct linearization points for the update operations. The next few lemmas focus on 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 UpdateStep object. The first child CAS that belongs to f (if there is one) must succeed.*

Proof. The proof of this lemma carries through from the BST proof with only simple substitutions, but their number justifies the proof's reproduction in full.

Let *ccas* be the *first child CAS* that belongs to f, and $fcas$ be the successful *flag CAS* for f, which exists and preceded *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 *pending* field of x at line 34. 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 *ccas*.

Because $fcas$ succeeds, the value in $x.pending$ just prior to $fcas$ must be the same as it was when it was read by r'. The step r' must have read a Clean object in $x.pending$; otherwise, the test at line 51 (for INSERT) or 80 or 82 (for DELETE) would evaluate to TRUE, and f would not have been created. Thus, by Lemma 4, $x.pending$ refers to a Clean object at all times between r' and $fcas$.

We now consider how $x.pending$ can change after $fcas$. If f is a PruneFlag, then the next change to $x.pending$ cannot be a *Backtrack CAS*, because a successful *mark CAS* belonging to f was performed (by Lemma 10) and, hence, there cannot be a *Backtrack CAS* belonging to f (by Lemma 6). Thus, if $x.pending$ 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 $ccas$, $x.pending$ either refers to a Clean object, or to f. By Lemma 12 and Lemma 14(2), when any successful *child CAS* $ccas_i$ occurs on x, $x.pending$ refers 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 UpdateStep object created by the operation.*

Proof. The proof of this lemma carries through from the BST proof with only the addition of an extra case for DELETE, and simple substitutions, but their number justifies the proof's reproduction in full.

Consider any INSERT or DELETE operation that returns TRUE, and let f be the UpdateStep object 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 that there is a *child CAS* that belongs to f by considering two cases.

If the operation is an INSERT, then it can return TRUE (at line 63) only after invoking HELPREPLACE using a reference to f on the preceding line. This HELPREPLACE performs an *Rchild CAS* belonging to f.

If the operation is a DELETE, then it can return TRUE at line 90 or 100. If it returns from line 90, then the call to HELPPRUNE at the same line returned TRUE, which means HELPPRUNE called HELPMARKED, which performed a *child CAS* belonging to f. Otherwise, if it returns

from line 100, then it invoked HelpReplace at the previous line, which performed an *Rchild CAS* belonging to f.

In all cases, there is some *child CAS* belonging 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 UpdateStep object f created by an* Insert *or* Delete *operation, then f is created during the last iteration of the operation's while loop.*

Proof. The proof of this lemma carries through from the BST proof with only the addition of an extra case for Delete, and simple substitutions, but their number justifies the proof's reproduction in full.

First, consider an Insert operation. Suppose there is a successful *Rchild CAS* belonging to a ReplaceFlag object f created by the Insert. By Lemma 12, there is a successful *Rflag CAS* belonging to f. After this successful *Rflag CAS* the Insert routine returns True at line 63 (unless it crashes), so the Insert does not perform any further iterations of its while loop.

Now, we consider two cases for the Delete operation.

Case I: Suppose there is a successful *Rchild CAS* belonging to a ReplaceFlag object f created by the Delete. By Lemma 12, there is a successful *Rflag CAS* belonging to f. After this successful *Rflag CAS* the Delete routine returns True at line 100 (unless it crashes), so the Delete does not perform any further iterations of its while loop.

Case II: Suppose there is a successful *Pchild CAS* belonging to a PruneFlag object f created by the Delete. Let *mcas* be the *first mark CAS* belonging to f (guaranteed to exist by Lemma 13). We argue that *mcas* must succeed. By Lemma 9, a successful *Pflag CAS* that belongs to f is performed before *mcas*. Prior to that *Pflag CAS*, the Search belonging to f reads the value *f.ppending* in *f.p.pending*. If *mcas* fails, the value of *f.p.pending* was changed to some value other than *f.ppending* before *mcas*, so it will never be changed back to *f.ppending* (by Lemma 4). Thus, all the later *mark CAS* steps belonging to f will also fail, contradicting Lemma 10. Hence, *mcas* must succeed. So, *f.p.pending* must contain a Mark object belonging to f at all times after *mcas*, by Lemma 3.

Now, after creating f, the Delete routine calls HelpPrune with a reference to f. When HelpPrune performs its *mark CAS*, it either

succeeds (if it is the first *mark CAS* that belongs to f) or it sees that *p.pending* already refers to a Mark object belonging to f. Thus, the test at line 107 will evaluate to TRUE, and HELPPRUNE 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 once successful child CAS belonging to the UpdateStep objects created by the the operation.*

Proof. The proof of this corollary carries through from the BST proof with the simple substitution "Info record" → "UpdateStep object." □

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

Proof. The proof of this corollary carries through from the BST proof with the simple substitutions "Info record" → "UpdateStep object" and "line 76" → "line 79." □

As in the BST proof, we can now define linearization points for update operations. If there is a successful *child CAS* belonging to an UpdateStep object created by an update operation, then the operation that created the UpdateStep object is linearized when that *child CAS* occurs. This choice of linearization point is unique, by Corollary 26. This defines linearization points for all updates that return TRUE, by Lemma 24, and does not define a linearization point for any update that returns FALSE, by Corollary 27. If an update operation returns FALSE, we linearize it at the same point 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. The proof of this lemma carries through from the BST proof without modification. □

As in the BST proof, the linearization points 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 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 UpdateStep object to which the *child CAS* belongs. However, several FIND operations and updates that return FALSE may be linearized at a single configuration. To make the linearization order a total ordering on the operations, we break ties arbitrarily. It remains to show that all terminating operations return the same results as they would if the operations were performed sequentially, in the linearization ordering.

Definition 10. *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 the 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 $O_1, ..., O_i$ were performed sequentially, in that order, starting with an empty dictionary. (I.e., $D_0 = \emptyset$ and $L_0 = \emptyset$ is the set of keys in the [empty] leaves of the tree in the initial configuration.)*

Note that the definition $L_0 = \emptyset$ differs slightly from the definition $L_0 = \{\infty_1, \infty_2\}$ given in the BST proof. This difference comes from the fact that leaves of the initial tree in the k-ST are empty (having no keys), and has the fortunate effect of simplifying the statement of Lemma 29(3).

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 $i \geq 0$, $L_i = D_i$.*

Proof. The proof of this lemma carries through from the BST proof with an extra case for DELETE, and simple substitutions, but their number justifies the proof's reproduction in full.

We prove the lemma by induction on i.

Base case ($i = 0$): $L_0 = \emptyset = D_0$.

Inductive step: Let $i \geq 1$. Assume the claims hold for $i - 1$. We consider two cases.

Case I: 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 ReplaceFlag object that $ccas_i$ belongs to. The SEARCH that belongs to f_i returned a reference $f_i.l$ with k not a key of $f_i.l$; otherwise, the INSERT would have returned FALSE at 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 $ccas_i$. Note that $f_i.l$ is in the *real tree* (rooted at $root.c_1.c_1$), since it is on the search path for $k \neq \infty$, and $root$ and $root.c_1$ have all ∞ keys, according to Lemma 2(10). Since the *real tree* is a k-ST (by Lemma 22), and there is a leaf $f_i.l$ on the search path for k in the *real tree* that does not contain the key k, k must not appear in the *real tree*. Further, by Lemma 22 and $k \neq \infty$, if k were in the tree, it would appear in the *real tree*. Hence, k must not appear in the tree. Thus, $k \notin L_{i-1}$ and, by induction hypothesis (3), $k \notin D_{i-1}$.

The effect of $ccas_i$ on the tree is to replace the leaf $f_i.l$ by either a leaf with keys $S =$ (keys of $f_i.l$) $\cup \{k\}$, or an internal node whose children contain precisely the keys of S (by Corollary 15 and the fact that $f.newChild$, the new node written by $ccas_i$, is created at line 55 or 57). 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 II: 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 PruneFlag or ReplaceFlag object that $ccas_i$ belongs to. The SEARCH that belongs to f_i returned a reference $f_i.l$ with k a key of $f_i.l$; otherwise, the DELETE would have returned FALSE at line 79.

We first consider the case when f_i is a PruneFlag object. 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 k-ST immediately before $ccas_i$ (by Lemma 17). By Corollary 16, $f_i.p$ is a child reference of $f_i.gp$ and $f_i.l$ is a child reference of $f_i.p$ immediately before $ccas_i$, so $f_i.l$ is also in the tree. Hence, $k \in$ (keys of $f_i.l$) is in L_{i-1} and, by induction hypothesis (3), $k \in D_{i-1}$.

The effect of $ccas_i$ is to replace $f_i.gp$'s reference to $f_i.p$ by a reference to $f_i.l$'s only non-empty sibling, thus removing exactly one non-empty leaf $f_i.l$ from the tree (by Corollary 16). Further, Corollary 16 states that $f_i.l$ had exactly one key when it was read by the SEARCH belonging to f. Since the keys of a node do not change, and we previously established that $f_i.l$ contains k, we know that the one key in $f_i.l$ is k. Thus, $L_i = L_{i-1} \setminus \{k\}$, so we have $L_i = L_{i-1} \setminus \{k\} = D_{i-1} \setminus \{k\} = D_i$.

We now consider the case when f_i is a ReplaceFlag object. 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, $f_i.p$ is in the k-ST immediately before $ccas_i$ (by Lemma 17). By Corollary 15, $f_i.l$ is a child reference of $f_i.p$ immediately before $ccas_i$, so $f_i.l$ is also in the tree. Hence, $k \in$ (keys of $f_i.l$) is in L_{i-1} and, by induction hypothesis (3), $k \in D_{i-1}$.

The effect of $ccas_i$ is to replace $f_i.p$'s reference to $f_i.l$ by a reference to a new leaf containing keys (keys of $f_i.l$) $\setminus \{k\}$ (by Corollary 15 and the fact that $f.newChild$, the new node written by $ccas_i$, is created at line 95). Thus, $L_i = L_{i-1} \setminus \{k\}$, so we have $L_i = L_{i-1} \setminus \{k\} = D_{i-1} \setminus \{k\} = D_i$. □

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* False*, then $k \notin D_i$.*
2. *If O is a* Find(k) *operation that returns* True*, 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. The proof of this lemma carries through from the BST proof with the following simple substitutions.

- "BST" → "k-ST"
- "returns $\perp$" → "returns False"
- "does not return $\perp$" → "returns True"
- "contains a key different from k" → "does not contain k" □

Lemma 31. *All* Find *operations linearized before $ccas_1$ return* False*. All* Delete *operations linearized before $ccas_1$ return* False*. There are no* Insert *operations linearized before $ccas_1$.*

Proof. The proof of this lemma carries through from the BST proof with the simple substitutions: "contains the key $\infty_1 \neq k$" → "is an empty leaf, containing no keys" and "$\perp$" → "False." □

Theorem 32. *The implementation given in Figure 3, 4 and 5 is a linearizable implementation of a dictionary.*

Proof. Just as in the BST 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 that Lemma 28 that the linearization respects the real-time ordering of operations.

Progress

We first prove the following three lemmas, which help to ensure progress for DELETE operations.

Lemma 33. *Suppose some process completes an execution of* HELP(f)*, where f is a ReplaceFlag or PruneFlag object. Then there is a successful unflag CAS or Backtrack CAS belonging to f.*

Proof. We consider two cases. If f is a ReplaceFlag object, then HELP calls HELPREPLACE(f), which performs an *Runflag CAS* belonging to f. The first such step succeeds.

If f is a PruneFlag object, then HELP calls HELPPRUNE(f). If the test at line 107 evaluated to TRUE, then HELPMARKED performs a *Punflag CAS* belonging to f. Otherwise, if the test evaluates to FALSE, a *Backtrack CAS* belonging to f is performed. The first *Punflag CAS* or *Backtrack CAS* that belongs to f succeeds. □

Lemma 34. *Let f be a PruneFlag object. If, in some configuration C, a node x has $x.pending = f$ and no Pchild CAS belonging to f has yet occurred, then $f.gp$ points to x and $f.p$ points to a child of x.*

Proof. The proof of this lemma carries over from the BST proof with simple substitutions, but their number justifies the proof's reproduction in full.

The only node that can be flagged with a pointer to f is the one that $f.gp$ points to (since *Pflag CAS* steps can only occur at line 88, where a PruneFlag object f is written into $f.gp$), so $f.gp$ must point to x. Let r be the step in which the SEARCH belonging to f read $f.p$ in a child reference of x, as described in Lemma 7. Let r' be the previous step of the SEARCH, which read the *pending* field of x at line 34.

Let $fcas$ be the (unique) *flag CAS* belonging to f. Because $fcas$ succeeded, the value in $x.pending$ just prior to $fcas$ must be the same as it was when it was read by r'. The step r' must have read a Clean object from $x.pending$. (Otherwise, the test at line 51 for INSERT, or 80 or 82 for DELETE, would evaluate to TRUE, and f would not have been created.) Thus, by Lemma 4, x refers to a Clean object at all times between r' and $fcas$. Further, since there is no ABA problem on *pending* fields, the fact that $x.pending = f$ in C implies that, at all times between r and C, $x.pending$ refers either to a Clean object or to f. By Lemma 12 and Lemma 14(2), when any successful *child CAS* $ccas_i$ occurs on x, x is flagged with a reference 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 $f.p$ refers to is still a child of x in configuration C. □

Lemma 35. *Let f be a PruneFlag object. At all times after some process performs a Pchild 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. The proof of this lemma carries through from the BST proof with the simple substitution "dchild CAS" → "*Pchild CAS*." □

Definition 11. *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 sub-graph 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 two nodes, with a single edge between them.

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*. We consider two cases.

By Corollary 15 and the fact that $f.newChild$ is created at line 55, 57 or 95, an *Rchild CAS* belonging to a ReplaceFlag object f either leaves G_i unchanged from G_{i-1} (if $f.newChild$ was a leaf, created at line 57 or 95), or adds an edge to G_i from some node to a node ($f.newChild$) with no outgoing edges in G_i (if $f.newChild$ was an internal node, created at line 55), so a cycle cannot be created in G_i, in either case.

By Corollary 16 a *Pchild CAS* belonging to a PruneFlag object 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 3, 4 and 5 is non-blocking.*

Proof. The proof of this final lemma carries through from the BST proof with only simple substitutions, but their number, and the size of the proof, would make this rather tedious to verify by hand. For this reason, the substitutions have been performed, and the proof reproduced in full below.

To derive a contradiction, assume there is some execution α that does not satisfy the non-blocking 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 CAS*, *mark CAS*, *Rflag CAS*, *Runflag CAS* and *Punflag CAS* steps in α.
Proof of Claim 1: There are a finite number of operations invoked in α (since it is assumed to stop making progress at some point). By Corollary 26, there is at most one successful *child CAS* per operation, so there are a finite number of successful *child CAS* steps. Thus, by Corollary 15 and Corollary 16 (which say that *Rchild CAS* steps add a finite number of nodes to the tree and *Pchild CAS* steps do not add nodes to the tree), 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 or Delete that successfully performs an *Rflag CAS* calls HelpReplace, which terminates in two CAS steps, and then the Insert or Delete itself terminates. Thus, each of the finitely many successful Insert and Delete operations in the execution has at most one successful *Rflag CAS*. By Lemma 3, two successful *Runflag CAS* steps on a node's *pending* field must have a successful *Rflag CAS* between them, so there are a finite number of successful *Rflag CAS* and *Runflag CAS* steps. There is at most one successful *Punflag CAS* belonging to each PruneFlag object f (by Lemma 3(4) and Lemma 4). 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 *Punflag 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 36 corresponds to moving the reference 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 HelpReplace and

HelpMarked clearly terminate in a constant number of steps.

Claim 2: Each call to Help and HelpPrune in α must terminate.
Proof of Claim 2: Help and HelpPrune do not contain loops, and call only each other or routines that are guaranteed to terminate. We must only show that they do not call each other in infinite, mutual recursion. To derive a contradiction, suppose this occurs. For all $i \geq 1$, let op_i be the PruneFlag argument to the ith call to HelpPrune in this infinite recursion, and let x_i be the node that $op_i.p$ refers to. Similarly, let u_i be the argument passed to Help by the ith invocation of HelpPrune.

Let $i > 1$. We show that (x_{i-1}, x_i) is an edge in G_α. When HelpPrune is called for the $(i-1)$th time, it performs an unsuccessful *mark CAS* on $x_{i-1}.pending$ and then calls Help using the value read from $x_{i-1}.pending$ as the argument u_{i-1}. Then, since the mutual recursion continues, u_{i-1} must refer to a PruneFlag object, and u_{i-1} will be passed as the argument op_i to HelpPrune. Since $op_i = u_{i-1}$ appeared in $x_{i-1}.pending$, $op_i.gp$ must refer to x_{i-1}, since it was placed there by a *Pflag CAS*. By definition, $op_i.p$ refers to x_i. The references $op_i.gp$ and $op_i.p$ were written into op_i after they were returned by a Search, and the postconditions of that Search ensure that $op_i.p$ was a child reference read in the node that $op_i.gp$ refers 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.

The only remaining routines whose termination has not been guaranteed are Insert and Delete. If a call to either routine does not terminate, then it must enter the routine's while loop infinitely many times and, hence, call Search infinitely many times. Thus, each operation in $S = \{O_1, ..., O_m\}$ performs infinitely many calls to Search with the same key as its argument. Since the tree eventually stabilizes to become T, every call to Search by O_i initiated after the tree has stabilized will return references 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 successful CAS steps that occur infinitely often in α are *Pflag CAS* and *Backtrack CAS* steps. As in the BST proof, we let α' be a suffix of α where:

1. the only successful CAS steps in α' are *Pflag CAS* 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 and l_i.

We also choose gp_{lowest} to be one of the lowest of the nodes $gp_1, ..., gp_m$ in T (so that no other gp_i is a proper descenant of gp_{lowest} in T), and let $S_{lowest} = \{O_i : gp_i = gp_{lowest}\}$.

For all i, the only node that O_i can attempt a *Pflag CAS* on in α is gp_i. Thus, no operation attempts a *Pflag CAS* on a proper descendant of gp_{lowest}. Therefore, there can be at most one successful *Backtrack CAS* applied to each proper descendant of gp_{lowest} in T, since there must be a successful *Pflag CAS* on a node between two successful *Backtrack CAS* steps on that node (by Lemma 3). Thus, in some suffix of α', no CAS steps succeed on the *pending* fields of proper descendants of gp_{lowest}, and the values in those fields stabilize. Let α'' be the suffix of α' where:

1. the *pending* 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 *pending* fields of all proper descendants of gp_{lowest} in T had already stabilized. (I.e., α'' starts once each O_i has invoked a SEARCH after the *pending* fields of proper descendants of gp_{lowest} in T have stabilized.)

Claim 3: If $O_i \in S_{lowest}$, then $p_i.pending$ does not refer to a Mark object during α''.

Proof of Claim 3: To derive a contradiction suppose that, during α'', $p_i.pending = m$ for some $Mark$ object m. For consistency of nomenclature with the BST proof, let f be shorthand for the PruneFlag object $m.pending$ to which m belongs. Then, the only step that can write m into $p_i.pending$ is a *mark CAS* belonging to f, so $f.p$ refers to p_i. We consider two cases.

First, suppose O_i is an INSERT. Then, O_i eventually calls HELP($p_i.pending$) at line 52. The HELP routine then calls HELPMARKED(f), which performs a *Pchild CAS* belonging to f. By Lemma 35, p_i is no longer reachable from the root after this *Pchild 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 *Pflag CAS* belonging to f was performed before $f.p$ was marked. By Lemma 35, no successful *Pchild CAS* belonging to f occurs, since p_i remains in the tree

forever. Moreover, according to the code, the first *Punflag CAS* belonging to f must be preceded by the first *Pchild CAS* belonging to f, which will succeed, by Lemma 23, so no successful *Punflag CAS* belonging to f ever occurs. Further, no *Backtrack CAS* belonging to f can occur, by Lemma 6. Thus, the node that $f.gp$ points to remains flagged with a reference to f forever and, hence, remains in the tree forever, by Lemma 17. Additionally, by Lemma 34, p_i remains a child of $f.gp$ forever. Thus, $f.gp$ must refer 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(f) at line 81, which calls HELPPRUNE(f). The *mark CAS* fails because $f.p.pending$ is already m with $m.pending = f$, so the process performing O_i then calls HELPMARKED(f), which performs a *Pchild 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 FINDs.) 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, $p_i.pending$ has stabilized in α'' (either to a flagged or clean value, since it cannot be marked by Claim 3). Consider an iteration of O_i's while loop in α''. If $p_i.pending$ refers to a Clean object in α'', then the test at line 52 evaluates to FALSE, and O_i will perform a successful *Rflag CAS* on $p_i.pending$, which is impossible because there are no successful *Rflag CAS* steps in α''. Thus, $p_i.pending$ must refer either to a PruneFlag or ReplaceFlag object in α'', and O_i invokes the HELP routine. By Lemma 33, the flag is eventually removed from p_i, contradicting the assumption that $p_i.pending$ has stabilized in α''. Thus, S_{lowest} cannot contain and 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 $gp_{lowest}.pending$ refers to a Clean object at the beginning of α'', then the claim is clearly satisfied.

Next, suppose that gp_{lowest} is marked at the beginning of α''. Let f be the PruneFlag object and m be the Mark object such that $m.pending = f$ and $gp_{lowest}.pending = m$. Then each process in S_{lowest} would eventually call HELP(m) from line 81, which would call HELPMARKED(f). Thus, a *Pchild 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 $gp_{lowest}.pending$ refers to a ReplaceFlag or PruneFlag object at the beginning of α'', then each process in S_{lowest} will eventually call HELP from line 81 and, by Lemma 33, gp_{lowest} will eventually become clean. This completes the proof of Claim 5.

Thus, eventually $gp_{lowest}.pending$ is Clean. The next change to gp_{lowest} can only be a *Pflag CAS*, since no *Rflag CAS* steps succeed in α''. We show that eventually some *Pflag CAS* on gp_{lowest} does succeed. Suppose this is not the case. Then $gp_{lowest}.pending$ eventually stabilizes to some Clean object. Thus, each DELETE operation O_i in S_{lowest} stops performing *Pflag CAS* steps. So, O_i must eventually evaluate the test at line 83 to be TRUE (since O_i must continue taking steps, but gp_{lowest} is Clean, and O_i cannot enter the else-block where it will perform a *Pflag CAS*). Recall that $p_i.pending$ 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 *pending* field. By Lemma 33, the flag on p_i is eventually removed, contradicting the fact that p_i's *pending* field does not change during α''. Thus, some DELETE O_i in S_{lowest} eventually performs a *Pflag CAS* on $gp_{lowest}.pending$.

Finally, we derive the required contradiction. After its successful *Pflag CAS*, O_i calls HELPPRUNE, which attempts a *mark CAS*. Since p_i's *pending* field has not changed since O_i read it in its previous SEARCH (since some SEARCH is guaranteed to have been started by O_i after the value of $p_i.pending$ stabilized, by construction of α''), 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. □