Non-blocking k-ary Search Trees

Trevor Brown[1] and Joanna Helga[2]

[1] Theory Group, Dept. of Computer Science, University of Toronto
tabrown@cs.toronto.edu
[2] DisCoVeri Group, Dept. of Computer Science and Engineering, York University
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 (see [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

A. Fernández Anta, G. Lipari, and M. Roy (Eds.): OPODIS 2011, LNCS 7109, pp. 207–221, 2011.

requires 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 $k > 4$, but we expect this would 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 to do so (as 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 our 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 with $a_i \leq a < a_{i+1}$.

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 new sub-tree if the leaf is full (has $k-1$ keys). Deleting a key replaces a leaf by a smaller leaf (without the deleted 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*.

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

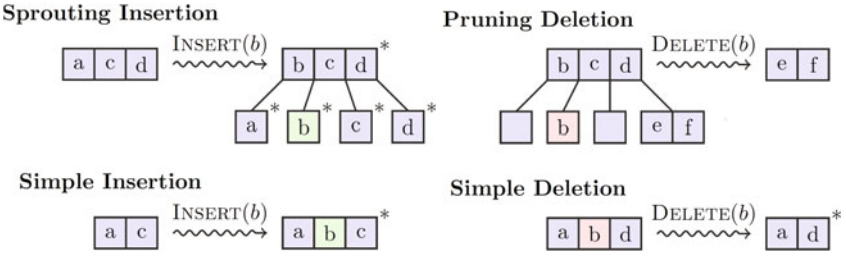

Fig. 1. The four types of modifications performed on the tree by an insertion or deletion. Asterisks indicate that nodes are newly created in freshly allocated memory.

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. Note that if NULL were used instead of empty children, then the ABA problem would occur on child pointers.

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 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 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

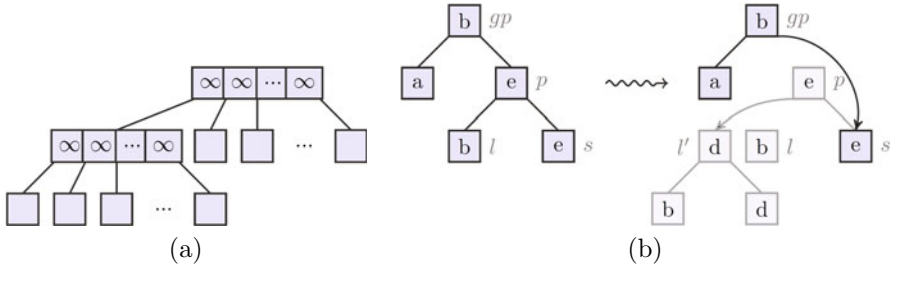

Fig. 2. (a) The initial state of the k-ary search tree. The root and its leftmost child have $k-1$ keys valued ∞ (a special key, larger than any key in the set). All other children of these nodes are empty leaves. Keys in the set are stored in the sub-tree rooted at the leftmost grandchild of the root.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

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 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 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.

```
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

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 a variable 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: it 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 leftmost child of *root* (line 29) and continuing until it reaches a leaf (line 32), it compares its argument key with the key stored at each node and follows the appropriate child reference (line 36), saving some information along the way. (The keys of a node can be naively inspected in sequence because they never change.) The FIND(key) operation returns TRUE if SEARCH finds a leaf containing key; otherwise it returns FALSE. FIND can actually call a highly optimized version of SEARCH, given in Appendix A (see [4]).

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), creating 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 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

```
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

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 help 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 HelpPrune returns False. 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. See [4] for the complete proof of correctness. Consider some invocation of Search(*key*). It can be proved

that each node visited by SEARCH was on the search path for *key* in the tree at some time during the execution of SEARCH, so we linearize SEARCH at a point when the leaf it returns was on the search path. 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.

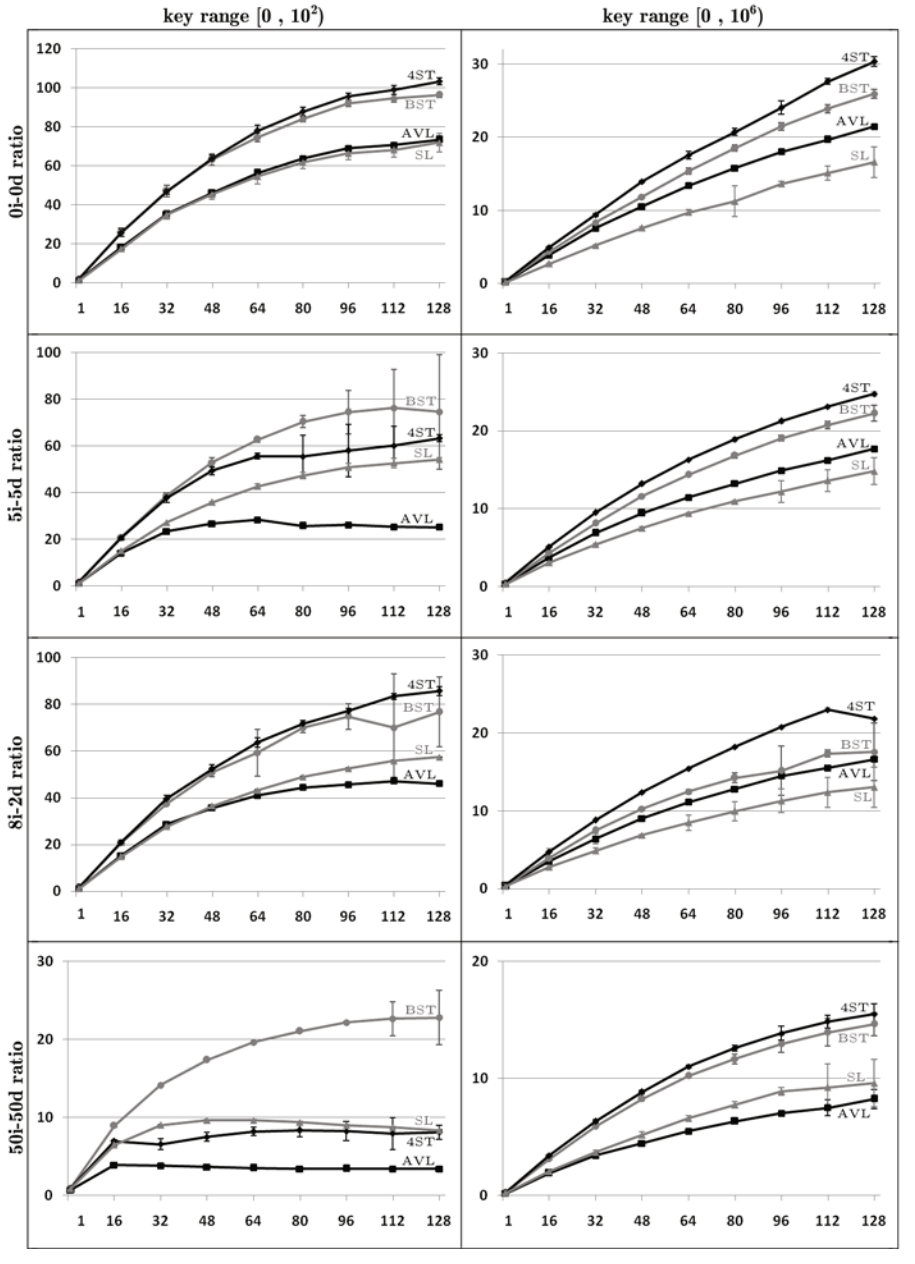

Fig. 6. Experimental results. Error bars are drawn to represent one standard deviation from the mean. Columns display ranges from which random keys are drawn. Rows display ratios of INSERTs to DELETEs to FINDs. The y-axis displays average throughput (millions of operations/sec.), and the x-axis displays 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 would 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 paper. Finally, we thank the anonymous OPODIS reviewers for their comments.

References

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

10. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proc. 22nd ACM Symposium on Principles of Distributed Computing, pp. 92–101 (2003)
11. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
12. Sundell, H., Tsigas, P.: Scalable and lock-free concurrent dictionaries. In: Proc. 19th ACM Symposium on Applied Computing, pp. 1438–1445 (2004)
13. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Proc. 14th ACM Symposium on Principles of Distributed Computing, pp. 214–222 (1995)