Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries

Panagiota Fatourou
FORTH ICS & University of Crete
Greece

Eric Ruppert
York University
Canada

FORTH ICS TR 470, May 2018

Abstract

This paper presents the *first* implementation of a search tree data structure in an asynchronous shared-memory system that provides a *wait-free* algorithm for executing range queries on the tree, in addition to non-blocking algorithms for Insert, Delete and Find. The implementation is linearizable, uses single-word compare-and-swap operations, and tolerates any number of crash failures. Insert and Delete operations that operate on different parts of the tree run fully in parallel (without any interference with one another). We employ a lightweight helping mechanism, where each Insert, Delete and Find operation helps only update operations that affect the local neighbourhood of the leaf it arrives at. Similarly, a RangeScan helps only those updates taking place on nodes of the part of the tree it traverses, and therefore RangeScans operating on different parts of the tree do not interfere with one another. Our implementation works in a dynamic system where the number of processes may change over time.

The implementation builds upon the non-blocking binary search tree implementation presented by Ellen *et al.* [13] by applying a simple mechanism to make the tree persistent.

1 Introduction

There has been much recent work on designing efficient concurrent implementations of *set* data structures [4, 5, 8, 10, 12, 13, 21, 29, 36, 38], which provide algorithms for Insert, Delete, and Find. There is increasing interest in providing additional operations for modern applications, including iterators [1, 32, 33, 35, 36, 37] or general range queries [6, 9]. These are required in many big-data applications [11, 26, 34], where shared in-memory tree-based data indices must be created for fast data retrieval and useful data analytics. Prevalent programming frameworks (e.g., Java [23], .NET [31], TBB [22]) that provide concurrent data structures have added operations to support (non-linearizable) iterators.

The Binary Search Tree (BST) is one of the most fundamental data structures. Ellen *et al.* [13] presented the first non-blocking implementation (which we will call NB-BST) of a BST from single-word CAS. NB-BST has several nice properties. Updates operating on different parts of the tree do not interfere with one other and Finds never interfere with any other operation. The code of NB-BST is modular and a detailed proof of correctness is provided in [14].

In this paper, we build upon NB-BST to get a persistent version of it, called PNB-BST. In a *persistent* data structure, old versions of the data structure are preserved when it is modified, so that one can access any old version. We achieve persistence on top of NB-BST by applying a relatively simple technique which fully respects the modularity and simplicity of NB-BST's design.

In a concurrent setting, a major motivation for providing data structure persistence is that it facilitates the implementation, in a wait-free way [18], of advanced operations (such as range queries) on top of the data structure. We exploit persistence in PNB-BST to provide the *first wait-free* implementation of RangeScan on top of tree data structures. RangeScan(a, b) returns a set containing

all keys in the implemented set that are between the given keys a and b. PNB-BST also provides *non-blocking* (also known as *lock-free* [18]) implementations of INSERT, DELETE, and FIND.

PNB-BST is *linearizable* [20], uses only single-word CAS, and tolerates any number of crash failures. As in NB-BST, updates in PNB-BST on different parts of the tree are executed in parallel without interfering with one another. A FIND simply follows tree edges from the root to a leaf and it may have to help an update operation only if the update is taking place at the parent or grandparent of the leaf that the search arrives at. Thus, FIND employs a lightweight helping mechanism. Similarly, RANGESCAN helps only those operations that are in progress on the nodes that it traverses. RANGESCAN may print keys (or perform some processing of the nodes, e.g., counting them) as it traverses the tree, thus avoiding any space overhead. PNB-BST does not require knowledge of the number of processes in the system, and therefore it works in a dynamic system where the set of participating processes changes.

The code of PNB-BST is as modular as that of NB-BST, making it fairly easy to understand. However, designing a linearizable implementation of RANGESCAN required solving several synchronization problems between RANGESCANs and concurrent update operations on the same part of the tree, so that a RANGESCAN sees all the successful update operations linearized before it but not those linearized after it. Specifically, we had to (a) apply a mechanism based on sequence numbers set by RANGESCANs, to split the execution into phases and assign each operation to a distinct phase, (b) design a scheme for linearizing operations that is completely different from that of of NB-BST by taking into consideration the phase to which each operation belongs, (c) ensure some additional necessary synchronization between RANGESCANs and updates, and (d) use a more elaborate helping scheme. The proof of correctness borrows from that of NB-BST. However, due to the mentioned complications, many parts of it are more intricate. The proof that RANGESCANs work correctly is completely novel.

2 Related Work

Our implementation is based on NB-BST, the binary search tree implementation proposed in [13]. Brown *et al.* [7] generalized the techniques in [13] to get the primitives LLX, SCX and VLX which are generalizations of load-link, store-conditional and validate. These primitives can be used to simplify the non-blocking implementation of updates in every data structure based on a down tree (see [8, 17] for examples). Unfortunately, our technique for supporting range queries cannot directly be implemented using LLX and SCX: the functionality hidden inside LLX must be split in two parts between which some synchronization is necessary to coordinate RANGESCANs with updates. The work in [13] has also been generalized in [38] to get a non-blocking implementation of a Patricia trie. None of these implementations of non-blocking search trees supports range queries.

Prokopec *et al.* [36] presented a non-blocking implementation of a concurrent hash trie which supports a SCAN operation that provides a consistent snapshot of the *entire* data structure. Their algorithm uses indirection nodes (i-nodes) [41] that double the height of the tree. To implement SCAN, the algorithm provides a persistent implementation of the trie in which updates may have to copy the entire path of nodes they traverse to synchronize with concurrent SCANs. Moreover, the algorithm causes a lot of contention on the root node. The algorithm could be adjusted to support RANGESCAN. However, every RANGESCAN would cause updates taking place anywhere in the tree to copy all the nodes they visit, even if they are not in the part of the tree being scanned.

Petrank and Timnat [35] gave a technique (based on [24]) to implement SCAN on top of non-blocking set data structures such as linked lists and skip lists. Concurrent SCANs share a *snap collector* object in which they record information about the nodes they traverse. To ensure that a SCAN appropriately synchronizes with updates, processes executing updates or FINDs must also record information about the operations they perform (or those executed by other processes they encounter) in the snap collector object. Although the snap collector object's primitive operations is wait-free, the following example shows that the implementation of SCAN using those primitives is non-blocking but not wait-free. Assume that the algorithm is applied on top of the non-blocking sorted linked list

implementation presented by Harris [16]. A SCAN must traverse the list, and this traversal may never complete if concurrent updates continue to add more elements to the end of the list faster than the SCAN can traverse them. In this case, the lists maintained in the snap collector will grow infinitely long. In case n is known, updates on different parts of the data structure do not interfere with one another and have been designed to be fast. However, SCAN is rather costly in terms of both time and space. Chatterjee [9] generalizes the algorithm of Petrank and Timnat to get a non-blocking implementation of RANGESCAN using partial snapshots [2]. In a different direction, work in [1, 37] characterizes when implementing the technique of [35] on top of non-blocking data structures is actually possible.

Brown *et al.* [6] presented an implementation of a k-ary search tree supporting RANGESCAN in an *obstruction-free* way [19]. Avni *et al.* [3] presented a skip list implementation which supports RANGESCAN. It can be either lock-free or be built on top of a transactional memory system, so its progress guarantees are weaker than wait-freedom. Bronson *et al.* [5] presented a *blocking* implementation of a relaxed-balance AVL tree which provides support for SCAN.

Some papers present *wait-free* implementations of SCAN (or RANGESCAN) on data structures other than trees or in different settings. Nikolakopoulos *et al.* [32, 33] gave a set of consistency definitions for SCAN and presented SCAN algorithms for the lock-free concurrent queue in [28] that ensure different consistency and progress guarantees. Fatourou *et al.* [15] presented a wait-free implementation of SCAN on top of the non-blocking deque implementation of [27]. Kanellou and Kallimanis [25] introduced a new graph model and provided a wait-free implementation of a node-static graph which supports partial traversals in addition to edge insertions, removals, and weight updates. Spiegelman *et al.* [39] presented two memory models and provided wait-free dynamic atomic snapshot algorithms for both.

3 Overview of the BST Implementation and Preliminaries

We provide a brief description of NB-BST (following the presentation in [13]) and some preliminaries.

NB-BST implements Binary Search Trees (BST) that are *leaf-oriented*, i.e., all keys are stored in the leaves of the tree. The tree is full and maintains the *binary search tree property*: for every node v in the tree, the key of v is larger than the key of every node in v's left subtree and smaller than or equal to the key of every node in v's right subtree. The keys of the Internal nodes are used solely for routing to the appropriate leaf during search. A leaf (internal) node is represented by an object of type Leaf (Internal, respectively); we say that Leaf and Internal nodes are of type Node (see Figure 2).

To insert a key k in a leaf-oriented tree, a search for k is first performed. Let ℓ and p be the leaf that this search arrives at and its parent. If ℓ does not contain k, then a subtree consisting of an internal node and two leaf nodes is created. The leaves contain k and the key of ℓ (with the smaller key in the left leaf). The internal node contains the bigger of these two keys. The child pointer of p which was pointing to ℓ is changed to point to the root of this subtree. Similarly, for a DELETE(k), let ℓ, p and gp be the leaf node that the search DELETE performs arrives at, its parent, and its grandparent. If the key of ℓ is k, then the child pointer of gp which was pointing to p is changed to point to the sibling of ℓ. By performing the updates in this way, the properties of the tree are maintained.

An implementation is *linearizable* if, in every execution α, each operation that completes in α (and some that do not) can be assigned a *linearization point* between the starting and finishing time of its execution so that the return values of those operations are the same in α as if the operations were executed sequentially in the order specified by their linearization points.

To ensure linearizability, NB-BST applies a technique that flags and marks nodes. A node is flagged before any of its child pointers changes. A node is permanently marked before it is removed. To mark and flag nodes, NB-BST uses CAS. CAS(O, u, v) changes the value of object O to v if its current value is equal to u, otherwise the CAS fails and no change is applied on O. In either case, the value that O had before the execution of CAS is returned.

NB-BST provides a routine, SEARCH(k), to search the data structure for key k. SEARCH returns pointers to the leaf node at which the SEARCH arrives, to its parent, and to its grandparent. FIND(k) executes SEARCH(k) and checks whether the returned leaf contains the key k. INSERT(k) executes

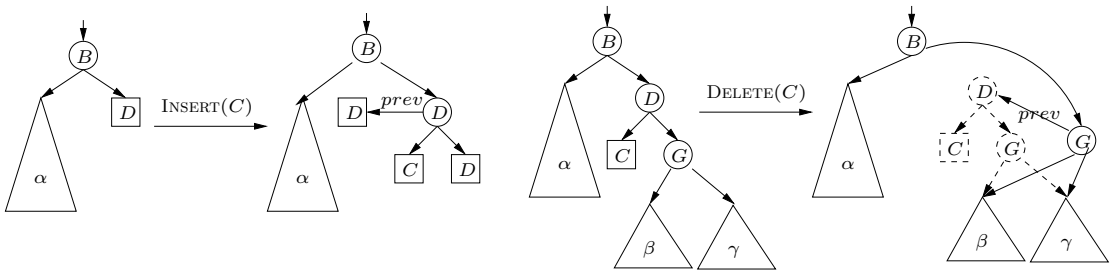

Figure 1: Examples of INSERT and DELETE.

SEARCH(k) to get a leaf ℓ and its parent p. It then performs a *flag* CAS, to flag p, then a *child* CAS to change the appropriate child pointer of p to point to the root of the newly created subtree of three nodes, and finally an *unflag* CAS to unflag p. If it fails to flag p, it restarts without executing the other two CAS steps. Similarly, a DELETE(k) calls SEARCH to get a leaf ℓ, its parent p, and its grandparent gp. It first executes a flag CAS trying to flag gp. If this fails, it restarts. If the flagging succeeds, it executes a *mark* CAS to mark p. If this fails, it unflags gp and restarts. Otherwise, it executes a child CAS to change the apropriate child pointer of gp to point from p to the sibling of ℓ, it unflags p and returns. Both INSERT and DELETE operations execute the body of a while loop repeatedly until they succeed. The execution of an iteration of the while loop is called *attempt*.

Processes may fail by *crashing*. An implementation is *non-blocking* if in every infinite execution, infinitely many operations are completed. NB-BST is non-blocking: Each process p that flags or marks a node stores in it a pointer to an Info object, which contains information about the operation op it performs (see Figure 2). This information includes the old and new values that should be used by the CAS steps that p will perform to complete the execution of op. Other processes that apply operations on the same part of the data structure can help this operation complete and unflag the node. Once they do so, they are able to retry their own operations. Helping is necessary only if an update operation wants to flag or mark a node already flagged or marked by another process.

4 A Persistent Binary Search Tree Supporting Range Queries

We modify NB-BST to get PNB-BST, a BST implementation that supports RANGESCAN, in addition to INSERT, DELETE, and FIND.

4.1 Overview

In a concurrent environment, care must be taken to synchronize RANGESCANs with updates since as a RANGESCAN traverses the tree, it may see an update op by a process p but it may miss an update that finishes before op starts, and was applied on the part of the tree that has already been visited by the RANGESCAN (thus violating linearizability).

To avoid such situations, PNB-BST implements a persistent version of the leaf-oriented tree, thus allowing a RANGESCAN to reconstruct previous versions of it. To achieve this, PNB-BST stores in each node an additional pointer, called *prev*. Whenever the child pointer of a node v changes from a node u to a node u', the *prev* pointer of u' points to u. (Figure 1 illustrates an example.)

PNB-BST maintains a shared integer, *Counter*, which is incremented each time a RANGESCAN takes place. Each operation has a *sequence number* associated with it. Each RANGESCAN starts its execution by reading *Counter* and uses the value read as its sequence number. Each other operation op reads *Counter* at the beginning of each of its attempts. The sequence number of op is the sequence number read in its last attempt. A successful update operation records its sequence number in the Info object it creates during its last attempt. Intuitively, each RANGESCAN initiates a new execution phase whenever it increments *Counter*. For each $i \geq 0$, phase i is the period during which *Counter*

```
1  type Update {                    ▷ stored in one CAS word
2      {Flag, Mark} type
3      Info *info
4  }
5  type Info {
6      {⊥, Try, Commit, Abort} state
7      Internal *nodes[]            ▷ nodes to be frozen
8      Update oldUpdate[]           ▷ old values for freeze CAS steps
9      Internal *mark[]             ▷ nodes to be marked
10     Internal *par                ▷ node whose child will change
11     Node *oldChild               ▷ old value for child CAS
12     Node *newChild               ▷ new value for the child CAS
13     int seq                      ▷ sequence number
14 }
15 type Internal {                   ▷ subtype of Node
16     Key ∪ {∞1, ∞2} key
17     Update update
18     Node *left, *right
19     Node *prev
20     int seq
21 }
22 type Leaf {                       ▷ subtype of Node
23     Key ∪ {∞1, ∞2} key
24     Update update
25     Node *prev
26     int seq
27 }
28 ▷ Initialization:
29 shared counter Counter := 0
30 shared Info *Dummy := pointer to a new Info object whose state field is Abort, and whose other fields are ⊥
31 shared Internal *Root := pointer to new Internal node with key field ∞2, update field ⟨Flag, Dummy⟩,
       prev field ⊥, seq field 0, and its left and right fields pointing to new Leaf nodes whose prev fields
       are ⊥, seq fields are 0, and keys ∞1 and ∞2, respectively
```

Figure 2: Type definitions and initialization.

has the value i. We say that all operations with sequence number i belong to phase i.

Each tree node has a *sequence number* which is the sequence number of the operation that created it. In this way, a RangeScan may figure out which nodes have been inserted or deleted by updates that belong to later phases. For any Internal node v whose sequence number is at most i, we define the *version-i left (or right) child of v* to be the node that is reached by following the left (or right) child pointer of v and then following its *prev* pointers until reaching the first node whose *seq* field is less than or equal to i. (We prove that such a node exists.) For every configuration C, we define graph $D_i(C)$ as follows. The nodes of $D_i(C)$ is the set of all existing nodes in C and the edges go from nodes to their version-i children; $T_i(C)$ is the subgraph of $D_i(C)$ containing those nodes that are reachable from the root node in $D_i(C)$. We prove that $T_i(C)$ is a binary search tree.

We linearize every Scan operation with sequence number i at the end of phase i, with ties broken in an arbitrary way. Moreover, we linearize all Insert, Delete and Find operations that belong to phase i during phase i. To ensure linearizability, PNB-BST should guarantee that a RangeScan with sequence number i ignores all changes performed by successful update operations that belong to phases with sequence numbers bigger than i. To ensure this, each operation with sequence number i ignores those nodes of the tree that have sequence numbers bigger than i by moving from a node to its appropriate version-i child. Thus, each operation with sequence number i always operates on T_i.

To ensure linearizability, PNB-BST should also ensure that each RangeScan sees all the successful updates that belong to phases smaller than or equal to i. To achieve this, PNB-BST employs a handshaking mechanism between each scanner and the updaters. It also uses a helping mechanism which is more elaborate than that of NB-BST.

To describe the handshaking mechanism in more detail, consider any update operation *op* initiated by process p. No process can be aware of *op* before p performs a successful flag CAS for *op*. Assume that p flags node v for *op* in an attempt *att* with sequence number i. To ensure that no RangeScan with sequence number i will miss *op*, p checks whether *Counter* still has the value i after the flag CAS has occurred. We call this check the *handshaking check* of *att*. If the handshaking check succeeds, it is guaranteed that no RangeScan has begun its traversal between the time that p reads *Counter* at the beginning of the execution of *att* and the time the handshaking check of *att* is executed. Note that any future RangeScan with sequence number i that traverses v while *att* is still in progress, will see that v is flagged and find out the required information to complete *op* in its Info object. In PNB-BST, the RangeScan helps *op* complete before it continues its traversal.

However, if the handshaking check fails, p does not know whether any RangeScan that incre-

```
32 SEARCH(Key k, int seq): ⟨Internal*, Internal*, Leaf*⟩ {
33     ▷ Precondition: seq ≥ 0
34     Internal *gp, *p
35     Node *l := Root
36     while l points to an internal node {
37         gp := p                                          ▷ Remember parent of p
38         p := l                                           ▷ Remember parent of l
39         l = READCHILD(p, k < p → key, seq)               ▷ Go to appropriate version-seq child of p
40     }
41     return ⟨gp, p, l⟩
42 }

43 READCHILD(Internal * p, Boolean left, int seq): Node* {
44     ▷ Precondition: p is non-⊥ and p → seq ≤ seq
45     if left then l := p → left else l := p → right   ▷ Move down to appropriate child
46     while (l → seq > seq) l := l → prev
47     return l;
48 }

49 VALIDATELINK(Internal *parent, Internal *child, Boolean left): ⟨Boolean, Update⟩ {
50     ▷ Preconditions: parent and child are non-⊥
51     Update up
52     up := parent → update
53     if FROZEN(up) then {
54         HELP(up.info)
55         return ⟨FALSE, ⊥⟩
56     }
57     if (left and child ≠ parent → left) or (¬left and child ≠ parent → right) then return ⟨FALSE, ⊥⟩
58     else return ⟨TRUE, up⟩
59 }

60 VALIDATELEAF(Internal *gp, Internal *p, Leaf *l, Key k) : ⟨Boolean, Update, Update⟩ {
61     ▷ Preconditions: p and l are non-⊥ and if p ≠ Root then gp is non-⊥
62     Update pupdate, gpupdate := ⊥
63     Boolean validated
64     ⟨validated, pupdate⟩ := VALIDATELINK(p, l, k < p → key)
65     if validated and p ≠ Root then ⟨validated, gpupdate⟩ := VALIDATELINK(gp, p, k < gp → key)
66     validated := validated and p → update = pupdate and (p = Root or gp → update = gpupdate)
67     return ⟨validated, gpupdate, pupdate⟩
68 }

69 FIND(Key k): Leaf* {
70     Internal * gp, p
71     Leaf *l
72     Boolean validated

73     while TRUE {
74         seq := Counter
75         ⟨−, p, l⟩ := SEARCH(k, seq)
76         ⟨validated, −, −⟩ := VALIDATELEAF(gp, p, l, k)
77         if validated then {
78             if l → key = k then return l
79             else return ⊥
80         }
81     }
82 }

83 CAS-CHILD(Internal *parent, Node *old, Node *new) {
       ▷ Precondition: parent points to an Internal node and new points to a Node (i.e., neither is ⊥) and new → prev = old
       ▷ This routine tries to change one of the child fields of the node that parent points to from old to new.
84     if new → key < parent → key then
85         CAS(parent → left, old, new)                     ▷ child CAS
86     else
87         CAS(parent → right, old, new)                    ▷ child CAS
88 }
```

Figure 3: Pseudocode for SEARCH, FIND and some helper routines.

```
89  Frozen(Update up): Boolean {
90      return ((up.type = Flag and up.info → state ∈ {⊥, Try}) or
                (up.type = Mark and up.info → state ∈ {⊥, Try, Commit}))
91  }

92  Execute (Internal *nodes[], Update oldUpdate[], Internal *mark[], Internal *par,
                Node *oldChild, Node *newChild, int seq): Boolean {
93      ▷ Preconditions: (a) Elements of nodes are non-⊥, (b) mark is a subset of nodes, (c) par is an element of nodes,
94          ▷ (d) oldChild and newChild are distinct and non-⊥, (e) oldChild is an element of mark,
95          ▷ (f) newChild → prev = oldChild, and (g) if par = Root then newChild → key is infinite.
96      for i := 1 to length of oldUpdate {
97          if Frozen(oldUpdate[i]) then {
98              if oldUpdate[i].info → state ∈ {⊥, Try} then Help(oldUpdate[i].info)
99              return False
100         }
101     }
102     infp := pointer to a new Info record containing ⟨⊥, nodes, oldUpdate, mark, par, oldChild, newChild, seq⟩
103     if CAS(nodes[1] → update, oldUpdate[1], ⟨Flag, infp⟩) then          ▷ freeze CAS
104         return Help(infp)
105     else return False
106 }

107 Help(Info *infp): boolean {
108     ▷ Precondition: infp is non-⊥ and does not point to the Dummy Info object
109     int i := 2
110     boolean continue

111     if Counter ≠ infp → seq then
112         CAS(infp → state, ⊥, Abort)                                    ▷ abort CAS
113     else CAS(infp → state, ⊥, Try)                                     ▷ try CAS
114     continue := (infp → state = Try)
115     while continue and i ≤ length of infp → nodes do {
116         if infp → nodes[i] appears in infp → mark then
117             CAS(infp → nodes[i] → update, infp → oldUpdate[i], ⟨Mark, infp⟩) ▷ freeze CAS
118         else CAS(infp → nodes[i] → update, infp → oldUpdate[i], ⟨Flag, infp⟩)  ▷ freeze CAS
119         continue := (infp → nodes[i] → update.info = infp)
120         i := i + 1
121     }
122     if continue then {
123         CAS-Child(infp → par, infp → oldChild, infp → newChild)
124         infp → state := Commit                                         ▷ commit write
125     } else if infp → state = Try then
126         infp → state := Abort                                          ▷ abort write
127     return (infp → state = Commit)
128 }

129 RangeScan(int a, int b): Set {
130     seq := Counter
131     Inc(Counter)
132     return ScanHelper(Root, seq, a, b)
133 }

134 ScanHelper(Node *node, int seq, int a, int b): Set {
135     ▷ Precondition: node points to a node with node → seq ≤ seq
136     Info * infp

137     if node points to a leaf then return {node → key} ∩ [a, b]
138     else {
139         infp := node → update.info
140         if infp → state ∈ {⊥, Try} then Help(infp)
141         if a > node → key then return ScanHelper(ReadChild(node, False, seq), a, b)
142         else if b < node → key then return ScanHelper(ReadChild(node, True, seq), a, b)
143         else return ScanHelper(ReadChild(node, False, seq), a, b) ∪
144                 ScanHelper(ReadChild(node, True, seq), a, b)
145     }
146 }
```

Figure 4: Pseudocode for Execute, Help and Scan.

```
147 INSERT(Key k): boolean {
148     Internal * gp, *p, *newInternal
149     Leaf *l, *newSibling
150     Leaf *new
151     Update pupdate
152     Info *infp
153     Boolean validated

154     while TRUE {
155         seq := Counter
156         ⟨gp, p, l⟩ := SEARCH(k, seq)
157         ⟨validated, −, pupdate⟩ := VALIDATELEAF(gp, p, l, k)
158         if validated then {
159             if l → key = k then return FALSE                    ▷ Cannot insert duplicate key
160             else {
161                 new := pointer to a new Leaf node whose key field is k, its seq field is equal to seq, and its prev field is ⊥
162                 newSibling := pointer to a new Leaf whose key is l → key,
                        its prev field is equal to ⊥ and its seq field is equal to seq
163                 newInternal := pointer to a new Internal node with key field max(k, l → key),
                        update field ⟨FLAG, Dummy⟩, its seq field equal to seq and its prev field equal to l,
                        and with two child fields equal to new and newSibling
                        (the one with the smaller key is the left child),
164                 if EXECUTE([p, l], [pupdate, l → update], [l], p, l, newInternal, seq) then return TRUE
165             }
166         }
167     }
168 }

169 DELETE(Key k): boolean {
170     Internal *gp, *p
171     Leaf *l
172     Node *sibling, *newnode
173     Update pupdate, gpupdate, supdate
174     Info *infp
175     Boolean validated

176     while TRUE {
177         seq := Counter
178         ⟨gp, p, l⟩ := SEARCH(k, seq)
179         ⟨validated, gpupdate, pupdate⟩ := VALIDATELEAF(gp, p, l, k)
180         if validated then {
181             if l → key ≠ k then return FALSE                    ▷ Key k is not in the tree
182             sibling := READCHILD(p, l → key ≥ p → key, seq)
183             ⟨validated, −⟩ := VALIDATELINK(p, sibling, l → key ≥ p → key)
184             if validated then {
185                 newNode := pointer to a new copy of sibling with its seq field set to seq and its prev pointer set to p
186                 if sibling is Internal then {
187                     ⟨validated, supdate⟩ := VALIDATELINK(sibling, newNode → left, TRUE)
188                     if validated then ⟨validated, −⟩ := VALIDATELINK(sibling, newNode → right, FALSE)
189                 } else supdate = sibling → update
190                 if validated and EXECUTE([gp, p, l, sibling], [gpupdate, pupdate, l → update, supdate],
                                             [p, l, sibling], gp, p, newNode, seq) then
191                     return TRUE
192             }
193         }
194     }
195 }
```

Figure 5: Pseudocode for INSERT and DELETE.

mented *Counter* to a value greater than i has already traversed the part of the tree that *op* is trying to update, and has missed this update. At least one of these RangeScans will have sequence number equal to i. Thus, if *op* succeeds, linearizability could be violated. To avoid this problem, p pro-actively aborts its attempt of *op* if the handshaking check fails, and then it initiates a new attempt for *op* (which will have a sequence number bigger than i). This abort mechanism is implemented as follows. The Info object has a field, called *status*, which takes values from the set $\{\perp, \text{Try}, \text{Commit}, \text{Abort}\}$ (initially $\perp$). Each attempt creates an Info object. To abort the execution of an attempt, p changes the *status* field of its Info object to Abort. Once an attempt is aborted, the value of the *status* field of its Info object remains Abort forever. If the handshaking check succeeds, then p changes the *status* field of the Info object of *att* to Try and tries to execute the remaining steps of this attempt. If *op* completes successfully, it changes the *status* field of the Info object to Commit. Info objects whose *status* field is equal to $\perp$ or Try belong to update operations that are still *in progress*.

We now describe the linearization points in more detail. If an attempt of an Insert or Delete ultimately succeeds in updating a child pointer of the tree to make the update take effect, we linearize the operation at the time that attempt first flags a node: this is when the update first becomes visible to other processes. (This scheme differs from the original NB-BST, where updates are linearized at the time they actually change a child pointer in the tree.) Because of handshaking, this linearization point is guaranteed to be before the end of the phase to which the operation belongs.

When a Find operation completes a traversal of a branch of the tree to a leaf, it checks whether an update has already removed the leaf or is in progress and could later remove that leaf from the tree. If so, the Find helps the update complete and retries. Otherwise, the Find terminates and is linearized at the time when the leaf is in the tree and has no pending update that might remove it later. (As in the original NB-BST, the traversal of the branch may pass through nodes that are no longer in the tree, but so long as it ends up at a leaf that is still present in the current tree we prove that it ends up at the correct leaf of the current tree.) An Insert(k) that finds key k is already in the tree, and a Delete(k) that discovers that k is not in the tree are linearized similarly to Find operations.

The helping mechanism employed by Find operations ensures that the Find will see an update that has been linearized (when it flags a node) before the Find but has not yet swung a child pointer to update the shape of the tree. But it is also crucial for synchronizing with RangeScan operations, for the following reason. Assume that a process p_1 initiates an Insert(1). It reads 0 in *Counter* and successfully performs its flag CAS. Then, a RangeScan is initiated by a process p_2 and changes the value of *Counter* from 0 to 1. Finally, a Find(1) is initiated by a process p_3 and reads 1 in *Counter*. Find(1) and Insert(1) will arrive at the same leaf node ℓ (because Insert(1) has not performed its child CAS by the time Find reaches the leaf). If Find(1) ignores the flag that exists on the parent node of ℓ and does not help Insert(1) to complete, it will return False. If Insert(1) now continues its execution, it will complete successfully, and given that it has sequence number 0, it will be linearized before Find(1) which has sequnce number 1. That would violate linearizability.

4.2 Detailed Implementation

A RangeScan(a, b) first determines its sequence number *seq* (line 130) and then increments *Counter* to start a new phase (line 131). To traverse the appropriate part of the tree, it calls ScanHelper($Root, seq, a, b$) (line 132). ScanHelper starts from the root and recursively calls itself on the version-*seq* left child of the current node v if a is greater than v's key, or on v's version-*seq* right child if b is smaller than v's key, or on both version-*seq* children if v's key is between a and b (lines 137–144). Whenever it visits a node where an update is in progress, it helps the update to complete (line 140). ReadChild is used to obtain v's appropriate version-*seq* child.

Search(k, seq) traverses a branch of T_{seq} from the root to a leaf node (lines 36–39). Find gets a sequence number *seq* (line 74) and calls Search(k, *seq*) (line 75) to traverse the BST to a leaf l. Next, it calls ValidateLeaf to ensure that there is no update that has removed l or has flagged l's parent p or grandparent gp for an update that could remove l from the tree. If the validation succeeds,

the FIND is linearized at line 66. If it finds an update in progress, the FIND helps complete it at line 54. If the validation is not successful, FIND retries.

An INSERT(k) performs repeated attempts. Each attempt first determines a sequence number *seq*, and calls SEARCH(k, *seq*) (line 156) to traverse to the appropriate leaf l in T_{seq}. It then calls VALIDATELEAF, just as FIND does. If the validation is successful and k is not already in the tree (line 159), a subtree of three nodes is created (lines 161–163). EXECUTE (line 164) performs the remaining actions of the INSERT, in a way that is similar to the INSERT of NB-BST.

In a way similar to INSERT(k), a DELETE(k) performs repeated attempts (line 176). Each attempt determines its sequence number *seq* (line 177) and calls SEARCH(k, *seq*) (line 178) to get the leaf ℓ, its parent p and grandparent gp. Next, it validates the leaf (as in FIND). If successful, it finds the sibling of ℓ (lines 182–189) and calls EXECUTE (line 190) to perform the remaining actions. We remark that, in contrast to what happens in NB-BST which changes the appropriate child pointer of gp to point to the sibling of ℓ, PNB-BST creates a new node where it copies the sibling of ℓ and changes the appropriate child pointer of gp to point to this new copy. This is necessary to avoid creating cycles consisting of *prev* and *child* pointers, which could cause infinite loops during SEARCH.

Finally, we discuss EXECUTE and HELP. EXECUTE checks whether there are operations in progress on the nodes that are to be flagged or marked and helps them if necessary (lines 96–99). If this is not the case, it creates a new Info object (line 102), performs the first flag CAS to make the Info object visible to other processes (line 103) and calls HELP to perform the remaining actions (line 104). HELP($infp$) first performs the handshaking (line 111–113). If *op* does not abort (line 114), HELP attempts to flag and mark the remaining nodes recorded in the Info object pointed to by $infp$ (lines 114–121). If it succeeds (line 122), it executes a child CAS to apply the required change on the appropriate tree pointer (line 123). If the child CAS is successful, *op* commits (line 124), otherwise it aborts (line 126).

We start by presenting an outline of the proof of correctness. We first prove each call to a subroutine satisfies its preconditions. This is proved together with some simple invariants, for instance, that READCHILD($-, -,$ *seq*) returns a pointer to a node whose sequence number is at most *seq*. Next, we prove that *update* fields of nodes are updated in an orderly way and we study properties of the child CAS steps. A node v is *frozen for an Info object* I if $v.update$ points to I and a call to FROZEN($v.update$) would return TRUE. A freeze CAS (i.e., a flag or mark CAS) *belongs to an Info object* I if it occurs in an instance of HELP whose parameter is a pointer to I, or on line 103 with I being the Info object created on line 102. We prove that only the first freeze CAS that belongs to an Info object I on each of the nodes in $I.nodes$ can be successful. Only the first child CAS belonging to I can succeed and this can only occur after all nodes in $I.nodes$ have been frozen. If a successful child CAS belongs to I, the *status* field of I never has the value ABORT. Specifically, this field is initially $\perp$ and changes to TRY or ABORT (depending on whether handshaking is performed successfully on lines 111-113). If it changes to TRY, then it may become COMMIT or ABORT later (depending on whether all nodes in $I.nodes$ are successfully frozen for I). A node remains frozen for I until $I.status$ changes to COMMIT or ABORT. Once this occurs, the value of $I.status$ never changes again. Only then can the *update* field of the node become frozen for a different Info object. Values stored in *update* fields of nodes and in *child* pointers are distinct (so no ABA problem may arise).

An *ichild* (*dchild*) CAS is a child CAS belonging to an Info object that was created by an INSERT (DELETE, respectively). Note that executing a successful freeze CAS (belonging to an Info object I with sequence number *seq*) on a node v acts as a "lock" on v set on behalf of the operation that created I. A successful child CAS belonging to I occurs only if the nodes that it will affect have been frozen. Every such node has sequence number less than or equal to *seq*. The ichild CAS replaces a leaf ℓ with sequence number $i \leq seq$ with a subtree consisting of an internal node v and two leaves (see Figure 1). All three nodes of this subtree have sequence number *seq* and have never been in the tree before. Moreover, the *prev* pointer of the internal node of this subtree points to ℓ (whereas those of the two leaves point to $\perp$). These changes imply that the execution of the ichild CAS does not affect any of the trees T_i with $i < seq$. The part of the tree on which the ichild CAS is performed cannot

change between the time all of the freeze CAS steps (for I) were performed and the time the ichild CAS is executed. So, the change that the ichild CAS performs is visible in every T_i with $i \geq seq$ just after this CAS has been executed. Similarly, a dchild CAS does not cause any change to any tree T_i with $i < seq$. However, for each $i \geq seq$, it replaces a node in T_i with a copy of the sibling of the node to be deleted (which is a leaf), thus removing three nodes from the tree (see Figure 1).

Characterizing the effects of child CAS steps in this way allows us to prove that no node in T_i, $i \geq 0$, ever acquires a new ancestor after it is first inserted in the tree. Using this, we also prove that if a node v is in the search path for key k in T_i at some time, then it remains in the search path for k in T_i at all later times. We also prove that for every node v an instance of SEARCH(k, seq) traverses, v was in T_{seq} (and on the search path for k in it) at some time during the SEARCH. These facts allows us to prove that every T_i, $i \geq 0$, is a BST at all times. Moreover, we prove that our validation scheme ensures that all successful update operations are applied on the latest version of the tree.

Fix an execution α. An update is *imminent* at some time during α if it has sucessfully executed its first freeze CAS before this time and it later executes a successful child CAS in α. We prove that at each time, no two imminent updates have the same key. For configuration C, let $Q(C)$ be the set of keys stored in leaves of T_∞ at C *plus* the set of keys of imminent INSERT operations at C *minus* the set of keys of imminent DELETE operations at C. Let the *abstract set* $L(C)$ be the set that would result if all update operations with linearization points at or before C would be performed atomically in the order of their linearization points. We prove the invariant that $Q(C) = L(C)$. Once we know this, we can prove that each operation returns the same result as it would if the operations were executed sequentially in the order defined by their linearization points, to complete the linearizability argument.

A RANGESCAN with sequence number i is wait-free because it traverses T_i, which can only be modified by updates that begin before the RANGESCAN's increment of the *Counter* (due to handshaking). To prove that the remaining operations are non-blocking, we show that an attempt of an update that freezes its first node can only be blocked by an update that freezes a lower node in the tree, so the update operating at a lowest node in the tree makes progress.

5 Proof of Correctness

5.1 Proof Outline

We start by presenting an outline of the proof of correctness. We first prove each call to a subroutine satisfies its preconditions. This is proved together with some simple invariants, for instance, that READCHILD($-, -, seq$) returns a pointer to a node whose sequence number is at most seq. Next, we prove that *update* fields of nodes are updated in an orderly way and we study properties of the child CAS steps. A node v is *frozen for an Info object* I if $v.update$ points to I and a call to FROZEN($v.update$) would return TRUE. A freeze CAS (i.e., a flag or mark CAS) *belongs to an Info object* I if it occurs in an instance of HELP whose parameter is a pointer to I, or on line 103 with I being the Info object created on line 102. We prove that only the first freeze CAS that belongs to an Info object I on each of the nodes in $I.nodes$ can be successful. Only the first child CAS belonging to I can succeed and this can only occur after all nodes in $I.nodes$ have been frozen. If a successful child CAS belongs to I, the *status* field of I never has the value ABORT. Specifically, this field is initially $\perp$ and changes to TRY or ABORT (depending on whether handshaking is performed successfully on lines 111-113). If it changes to TRY, then it may become COMMIT or ABORT later (depending on whether all nodes in $I.nodes$ are successfully frozen for I). A node remains frozen for I until $I.status$ changes to COMMIT or ABORT. Once this occurs, the value of $I.status$ never changes again. Only then can the *update* field of the node become frozen for a different Info object. Values stored in *update* fields of nodes and in *child* pointers are distinct (so no ABA problem may arise).

An *ichild* (*dchild*) CAS is a child CAS belonging to an Info object that was created by an INSERT (DELETE, respectively). Note that executing a successful freeze CAS (belonging to an Info object

I with sequence number seq) on a node v acts as a "lock" on v set on behalf of the operation that created I. A successful child CAS belonging to I occurs only if the nodes that it will affect have been frozen. Every such node has sequence number less than or equal to seq. The ichild CAS replaces a leaf ℓ with sequence number $i \leq seq$ with a subtree consisting of an internal node v and two leaves (see Figure 1). All three nodes of this subtree have sequence number seq and have never been in the tree before. Moreover, the $prev$ pointer of the internal node of this subtree points to ℓ (whereas those of the two leaves point to $\perp$). These changes imply that the execution of the ichild CAS does not affect any of the trees T_i with $i < seq$. The part of the tree on which the ichild CAS is performed cannot change between the time all of the freeze CAS steps (for I) were performed and the time the ichild CAS is executed. So, the change that the ichild CAS performs is visible in every T_i with $i \geq seq$ just after this CAS has been executed. Similarly, a dchild CAS does not cause any change to any tree T_i with $i < seq$. However, for each $i \geq seq$, it replaces a node in T_i with a copy of the sibling of the node to be deleted (which is a leaf), thus removing three nodes from the tree (see Figure 1).

Characterizing the effects of child CAS steps in this way allows us to prove that no node in T_i, $i \geq 0$, ever acquires a new ancestor after it is first inserted in the tree. Using this, we also prove that if a node v is in the search path for key k in T_i at some time, then it remains in the search path for k in T_i at all later times. We also prove that for every node v an instance of SEARCH(k, seq) traverses, v was in T_{seq} (and on the search path for k in it) at some time during the SEARCH. These facts allows us to prove that every T_i, $i \geq 0$, is a BST at all times. Moreover, we prove that our validation scheme ensures that all successful update operations are applied on the latest version of the tree.

Fix an execution α. An update is *imminent* at some time during α if it has sucessfully executed its first freeze CAS before this time and it later executes a successful child CAS in α. We prove that at each time, no two imminent updates have the same key. For configuration C, let $Q(C)$ be the set of keys stored in leaves of T_∞ at C *plus* the set of keys of imminent INSERT operations at C *minus* the set of keys of imminent DELETE operations at C. Let the *abstract set* $L(C)$ be the set that would result if all update operations with linearization points at or before C would be performed atomically in the order of their linearization points. We prove the invariant that $Q(C) = L(C)$. Once we know this, we can prove that each operation returns the same result as it would if the operations were executed sequentially in the order defined by their linearization points, to complete the linearizability argument.

A RANGESCAN with sequence number i is wait-free because it traverses T_i, which can only be modified by updates that begin before the RANGESCAN's increment of the *Counter* (due to handshaking). To prove that the remaining operations are non-blocking, we show that an attempt of an update that freezes its first node can only be blocked by an update that freezes a lower node in the tree, so the update operating at a lowest node in the tree makes progress.

5.2 Formal Proof

We now provide the full proof of correctness. Specifically, we prove that the implementation is linearizable and satisfies progress properties. The early parts of the proof are similar to proofs in previous work [7, 14, 38], but are included here for completeness since the details differ. Most of the more novel aspects of the proof are in Sections 5.2.4 and 5.2.5.

5.2.1 Basic Invariants

We start by proving some simple invariants, and showing that there are no null-pointer exceptions in the code.

Observation 1 *The key, prev and seq fields of a Node never change. No field of an Info record, other than state, ever changes. The Root pointer never changes.*

Observation 2 *If an Info object's state field is* COMMIT *or* ABORT *in some configuration, it can never be* $\perp$ *or* TRY *in a subsequent configuration.*

Proof: The state of an Info object can be changed only on lines 112, 113, 124 and 126. None of these can change the value from COMMIT or ABORT to $\perp$ or TRY. ■

Observation 3 *The value of Counter is always non-negative, and for every configuration C and every node v in configuration C, $v.seq \leq Counter$.*

Proof: The *Counter* variable is initialized to 0 and never decreases. All nodes in the initial configuration have *seq* field 0. Whenever a node is created by an INSERT or DELETE, its *seq* field is assigned a value that the update operation read from *Counter* earlier. ■

Invariant 4 *The following statements hold.*

1. *Each call to a routine satisfies its preconditions.*
2. *Each* SEARCH *that has executed line 35 has local variables that satisfy the following: $l \neq \perp$ and $l \rightarrow seq \leq seq$.*
3. *Each* SEARCH *that has executed line 38 has local variables that satisfy the following: $p \neq \perp$ and $p \rightarrow seq \leq seq$.*
4. *Each* SEARCH *that has executed line 35 has local variables that satisfy the following: if $l \rightarrow key$ is finite then $gp \neq \perp$ and $gp \rightarrow seq \leq seq$.*
5. *Each* READCHILD *that has executed line 45 has local variables that satisfy the following: $l \neq \perp$ and there is a chain of prev pointers from l to a node whose seq field is at most seq.*
6. *Each* READCHILD *that terminates returns a pointer to a node whose sequence number is at most seq.*
7. *Each* FIND *that has executed line 75 has non-$\perp$ values in its local variables p and l.*
8. *Each* INSERT *that has executed line 156 has local variables that satisfy the following: $p \neq \perp$ and $l \neq \perp$ and $p \rightarrow seq \leq seq$.*
9. *Each* DELETE *that has executed line 178 has local variables that satisfy the following: $p \neq \perp$ and $l \neq \perp$ and $p \rightarrow seq \leq seq$. Moreover, if $l \rightarrow key = k$, then $gp \neq \perp$ and $gp \rightarrow seq \leq seq$.*
10. *For each Internal node v, v's children pointers are non-$\perp$. Moreover, one can reach a node with sequence number at most $v.seq$ by tracing prev pointers from either of v's children.*
11. *For each Info object I except Dummy, all elements of $I.nodes$ are non-$\perp$, $I.mark$ is a subset of $I.nodes$, $I.par$ is an element of $I.nodes$, $I.oldChild$ and $I.newChild$ are distinct and non-$\perp$, $I.oldChild$ is an element of $I.mark$, and $I.newChild \rightarrow prev = I.oldChild$.*
12. *Each Update record has a non-$\perp$ $info$ field.*
13. *For any Internal node v, any node u reachable from $v.left$ by following a chain of prev pointers has $u.key < v.key$ and any node w reachable from $v.right$ by following a chain of prev pointers has $w.key \geq v.key$.*
14. *For any Info object I, if $I.par = Root$, then $I.newChild \rightarrow key$ is infinite.*
15. *Any node u that can be reached from $Root \rightarrow left$ by following a chain of prev pointers has an infinite key.*

16. *For any Internal node v, any terminating call to* ReadChild(v, left, seq) *returns a node whose key is less than $v.key$, and any terminating call to* ReadChild(v, right, seq) *returns a node whose key is greater than or equal to $v.key$. Any call to* ReadChild$(Root, \text{left}, seq)$ *returns a node whose key is infinite.*

Proof: We prove that all claims are satisfied in every finite execution by induction on the number of steps in the execution.

For the base case, consider an execution of 0 steps. Claims 1 to 9 are satisfied vacuously. The initialization ensures that claims 10 to 15 are true in the initial configuration.

Assume the claims hold for some finite execution α. We show that the claims hold for $\alpha \cdot s$, where s is any step.

1. If s is a call to Search at line 75, 156 or 178, the value of *seq* was read from *Counter* in a previous line. The value of *Counter* is always non-negative, so the precondition of the Search is satisfied.

   If s is a call to ReadChild on line 39, the preconditions are satisfied by induction hypothesis 3. If s is a call to ReadChild on line 182, the preconditions are satisfied by induction hypothesis 9. If s is a call to ReadChild on line 141 to 144, the preconditions are satisfied because ScanHelper's preconditions were satisfied (by induction hypothesis 1).

   If s is a call to ValidateLink on line 64 or 65 of ValidateLeaf, the preconditions follow from the preconditions of ValidateLeaf, which are satisfied by induction hypothesis 1. (In the latter case, we know from the test on line 65 that $p \neq \perp$.) If s is a call to ValidateLink on line 183, the preconditions are satisfied because the Search on line 178 returned a node p with sequence number at most *seq* by induction hypothesis 3, and then ReadChild on line 182 returned a node, by induction hypothesis 6. If s is a call to ValidateLink on line 187 or 188, the preconditions are satisfied by induction hypothesis 6 applied to the preceding call to ReadChild on line 182.

   If s is a call to ValidateLeaf on line 76, 157 or 179, then the preconditions follow from induction hypotheses 2, 3, 4 and readchild-result applied to the preceding call to Search on line 75, 156 or 178, respectively.

   If s is a call to Execute on line 164 of Insert, preconditions (a)–(f) follow from induction hypothesis 8 and the fact that line 163 creates *newInternal* after reading l and sets $newInternal \rightarrow prev$ to l. It remains to prove precondition (g). Suppose $p = Root$. Since ValidateLeaf on line 157 returned True, the call to ValidateLink on line 64 also returned True. So, l was the result of the ReadChild$(Root, left, seq)$ on line 57 of ValidateLink. By induction hypothesis 16, l has an infinite key. Thus, the new Internal node created on line 163 of the Insert has an infinite key, as required to satisfy precondition (g).

   If s is a call to Execute on line 190 of Delete, preconditions (a)–(c) follow from induction hypothesis 9 and the fact that $l \rightarrow key = k$ (since the Delete did not terminate on line 181), and induction hypothesis 6 applied to the preceding call to ReadChild on line 182. Precondition (d) follows from the additional fact that *newNode* is created on line 185 after reading a pointer to *sibling*, which as already argued is non-$\perp$. Precondition (e) is obviously satisfied. Precondition (f) follows from the fact that line 185 sets $newNode \rightarrow prev$ to be p. It remains to prove precondition (g). Suppose $gp = Root$. Since ValidateLeaf on line 179 returned True, the call to ValidateLink on line 65 also returned True. Then, p was the result of the ReadChild$(Root, \text{left}, seq)$ on line 57 of ValidateLink. By induction hypothesis 16, p has an infinite key. The ReadChild(p, right, seq) on line 182 returns *sibling*, which also has an infinite key by induction hypothesis 16. Thus, the node *newNode* created at line 185 has an infinite key, as required to satisfy precondition (g).

If s is a call to HELP on line 54, 98 or 140, the argument is non-$\perp$, by induction hypothesis 12. Moreover, the preceding call to INPROGRESS returned true, so the Info object had state $\perp$ or TRY. By Observation 2, this Info object cannot be the Dummy object, which is initialized to have state ABORT. If s is a call to HELP on line 104, the precondition is satisfied, since the argument $infp$ is created at line 102.

If s is a call to CAS-CHILD on line 123, the Info object $infp$ is not the Dummy, by the precondition to HELP, which was satisfied when HELP was called, by induction hypothesis 1. So, the preconditions of CAS-CHILD are satisfied by induction hypothesis 11.

If s is a call to SCANHELPER on line 132, the precondition is satisfied since $Root \rightarrow seq = 0$ and the value of $Counter$ is always non-negative. If s is a call to SCANHELPER on line 141 to 144, the precondition is satisfied by induction hypothesis 6.

2. By Observation 1, the seq field of a node does not change. So it suffices to prove that any update to l in the SEARCH routine preserves the invariant.

   Line 35 sets l to $Root$ which has $Root \rightarrow seq = 0$. By induction hypothesis 1, the SEARCH has $seq \geq 0$, so claim 2 is satisfied.

   Line 39 sets l to the result of a READCHILD, so claim 2 is satisfied by induction hypothesis 6.

3. It suffices to prove that any upate to p in the SEARCH routine preserves the invariant. Whenever p is updated at line 38, it is set to the value stored in l, so claim 3 follows from induction hypothesis 2.

4. First, suppose s is the first step of a SEARCH that sets l so that $l \rightarrow key$ is finite. Then s is not an execution of line 35, because $Root$ never changes and has key ∞_2, by Observation 1. Likewise, s is not the assignment to l that occurs in the first execution of line 39, since the READCHILD on that line (which terminates before s) would have returned a node with an infinite key, by induction hypothesis 16. Thus, s occurs after the second execution of line 37, which happens after the first execution of line 38. By induction hypothesis 3, the second execution of line 38 assigns a non-null value to gp, and $gp \rightarrow seq \leq seq$.

   It remains to consider any step s that assigns a new value to gp (at line 37) after the first time l is assigned a node with a finite value. As argued in the previous paragraph, this execution of line 37 will not occur in the first two iterations of the SEARCH's while loop. So the claim follows from induction hypothesis 3.

5. By Observation 1, $prev$ fields are never changed. Thus, it suffices to show that any step s that updates l inside the READCHILD routine maintains this invariant.

   If s is a step that sets l to a child of p at line 45, the claim follows from induction hypothesis 10 applied to the configuration just before s.

   If s is an execution of line 46, the claim is clearly preserved.

6. If s is a step in which READCHILD terminates, the claim follows from induction hypothesis 5 applied to the configuration prior to s.

7. It suffices to consider the step s in which the SEARCH called at line 75 terminates. That SEARCH performed at least one iteration of its while loop (since $Root$ is an Internal node). So, by induction hypotheses 2 and 3, it follows that the values that SEARCH returns, which the FIND stores in p and l, are not $\perp$.

8. It suffices to consider the step s in which the SEARCH called at line 156 terminates. That SEARCH performed at least one iteration of its while loop (since $Root$ is an Internal node). So,

by induction hypotheses 2 and 3, it follows that the values that SEARCH returns, which the INSERT stores in p and l, are not $\perp$ and have *seq* fields that are at most *seq*.

9. It suffices to consider the step s in which the SEARCH called at line 178 terminates. That SEARCH performed at least one iteration of its while loop (since *Root* is an Internal node). So, by induction hypotheses 2 and 3, it follows that the values that SEARCH returns, which the DELETE stores in p and l, are not $\perp$ and have *seq* fields that are at most *seq*. If $l \rightarrow key = k$, it follows from induction hypothesis 4 that the value SEARCH returns, which the DELETE stores in *gp*, is not $\perp$ and that $gp \rightarrow seq \leq seq$.

10. By Observation 1, *prev* pointers are never changed. Thus, it suffices to show that every step s that changes a child pointer preserves this invariant. Consider a step s that changes a child pointer by executing a successful child CAS (at line 85 or 87). By the precondition of CAS-CHILD, the new child pointer will be non-$\perp$ and this new child's *prev* pointer will point to the previous child. Since one could reach a node with *seq* field at most *seq* by following *prev* pointers from the old child (by induction hypothesis 10), this will likewise be true if one follows *prev* pointers from the new child.

11. By Observation 1, the $nodes, mark, par, oldChild$ and $newChild$ fields of an Info object never change. Thus it is sufficient to consider the case where the step s is the creation of a new Info object at line 102 of the EXECUTE routine. Claim 11 for the new Info object follows from the fact that the preconditions of EXECUTE were satisfied when it was invoked before s.

12. We consider all steps s that construct a new Update record. If s is an execution of line 103, the $info$ field of the new Update record is $infp$, which is defined on the previous line to be non-$\perp$. If s is an execution of line 117 or 118 in the HELP routine, the $info$ field of the new Update record is $infp$, which is non-$\perp$, since induction hypothesis 1 ensures that the preconditions of the HELP routine were satisfied when it was called. If s is an execution of line 163, the *update* field of the newly created node is set to a new Update record, $\langle \text{FLAG}, Dummy \rangle$, which has a non-$\perp$ info field.

13. If s is a step that creates a new Internal node v (by executing line 163), v's left and right children are initialized to satisfy the claim.

    By observation 1, *key* and *prev* fields of nodes are never changed, so it suffices to consider steps that change a child pointer. If s is a step that changes v's child pointer (by executing line 85 or 87 in the CAS-CHILD routine) from *old* to *new*, it follows from the test on line 84 that the new child *new* has a key that satisfies the claim. Moreover, by induction hypothesis 1, the precondition of CAS-CHILD was satisfied when it was called, so $new \rightarrow prev = old$. By induction hypothesis 13, every node reachable from *old* by following *prev* pointers satisfied the claim. So every node reachable from *new* by following *prev* pointers satisfies the claim too.

14. By Observation 1, an Info object's *par* and *newChild* fields do not change, and *prev* and *key* fields of nodes do not change. Thus, it suffices to consider steps s that create a new Info object (at line 102 of the EXECUTE routine). The claim follows from the fact that the preconditions of EXECUTE were satisfied when it was called, by induction hypothesis 1.

15. By observation 1, *key* and *prev* fields of nodes are never changed, so it suffices to consider steps that change the left child pointer of *Root*. Suppose s is a step that changes $Root \rightarrow left$ (by executing line 85 or 87 in the CAS-CHILD routine) from *old* to *new*. That CAS-CHILD was called at line 123 of HELP. By induction hypothesis 14, *new* has an infinite key. Moreover, by induction hypothesis 1, the precondition of CAS-CHILD was satisfied when it was called, so $new \rightarrow prev = old$. By induction hypothesis 13, every node reachable from *old* by following

prev pointers has an infinite key. So every node reachable from *new* by following *prev* pointers has an infinite key.

16. Suppose s is the step in which a call to ReadChild returns. By induction hypothesis 13 and 15, when the ReadChild executed line 45, every node reachable from l by following a chain of *prev* pointers had the required property. By Observation 1, *prev* pointers do not change. So, the node returned by ReadChild has the required property.

■

Invariant 5 *For each Info object I and each i, $I.nodes[i] \rightarrow seq \leq I.seq$.*

Proof: By Observation 1, the *nodes* and *seq* fields of Info objects, and the *seq* fields of nodes do not change. So it suffices to show that the claim is true whenever a new info object I is created (at line 102 of Execute). The Execute creates I using the *nodes* and *seq* parameters of the call to Execute, which is called at line 164 or 190.

If Execute is called at line 164 of an Insert, the *nodes* parameter contains nodes returned from a call to Search(k, seq). The sequence numbers of these two nodes are at most *seq*, by Invariant 4.3 and 4.2, respectively.

If Execute is called at line 190 of a Delete, the *nodes* parameter contains nodes returned from a call to Search(k, seq) on line 178 and a call to ReadChild on line 182. The sequence numbers of these four nodes are at most *seq*, by Invariant 4.4, 4.3, 4.2 and 4.6, respectively. ■

5.2.2 How the *update* Fields are Changed

The next series of lemmas describes how *update* fields of nodes are changed. This part of the proof is quite similar to other papers that have used similar techniques for flagging or marking nodes, e.g., [14, 7]. However, since we use a slightly different coordination scheme from those papers, we include the lemmas here for the sake of completeness.

Lemma 6 *For each Info object I and all i, $I.oldUpdate[i]$ was read from the update field of $I.nodes[i]$ prior to the creation of I.*

Proof: Consider the creation of an Info object I (at line 102 of Execute, which is called either at line 164 or 190). So, it suffices to show that the claim is true for the arguments *nodes* and *oldUpdate* that are passed as arguments in these calls to Execute.

If Execute$([p, l], [pupdate, l \rightarrow update], ...)$ was called at line 164, then *pupdate* was read from $p \rightarrow update$ in the call to ValidateLeaf at line 157, and l's update field is read at line 164.

If Execute$([gp, p, l, sibling], [gpupdate, pupdate, l \rightarrow update, supdate], ...)$ was called at line 190, then *gpupdate* and *pupdate* were read from the *update* fields of gp and p during the ValidateLeaf routine called at line 179. The value of *supdate* was read from $sibling \rightarrow update$ either during the call to ValidateLink at line 187 or at line 189, depending on whether *sibling* is an Internal node or a Leaf. Finally l's update field is read at line 190 itself. ■

The following lemma shows that no ABA problem ever occurs on the *update* field of a node.

Lemma 7 *For each node v, the field $v.update$ is never set to a value that it has previously had.*

Proof: The $v.update$ field can only be changed by the CAS steps at line 103, 117 or 118. By Lemma 6, the CAS changes the $info$ subfield from a pointer to some Info object I to a pointer to another Info object I', where I' was created after I. The claim follows. ■

We define some names for key steps for the algorithms that update the data structure. The CAS steps on lines 103 and 118 are called *flag* CAS steps, and the CAS on line 117 is called a *mark* CAS. A *freeze* CAS step is either a flag CAS or a mark CAS. An *abort* CAS occurs on line 112 and a *try* CAS on line 113. A *child* CAS occurs on line 85 or 87. Lines 124 and 126 are called *commit writes* and *abort writes*, respectively.

Any step performed inside a call to HELP($infp$) is said to *belong to* the Info object that $infp$ points to, including the steps performed inside the call to CAS-CHILD on line 123. The freeze CAS on line 103 is also said to belong to the Info object created on the previous line.

Lemma 8 *For each Info object I and each i, only the first freeze* CAS *on $I.nodes[i]$ that belongs to I can succeed.*

Proof: Let u be the node that $I.nodes[i]$ points to. All freeze CAS steps on u that belong to I use the same old value o for the CAS, and o is read from $u.update$ prior to the creation of I. If the first such freeze CAS fails, then the value of $u.update$ has changed from o to some other value before that first CAS. If the first freeze CAS succeeds, then it changes $u.update$ to a value different from o (since o cannot contain a pointer to I which was not created when o was read, and the new value does contain a pointer to o). Either way, the value of $u.update$ is different from o after the first freeze CAS, and it can never change back to o afterwards, by Lemma 7. Thus, no subsequent freeze CAS on $I.nodes[i]$ that belongs to I can succeed. ■

We next show that the *update* field of a node can be changed only if the *state* field of the Info object it points to is COMMIT or ABORT.

Lemma 9 *Let v be any node. If a step changes $v.update$, then $v.update.info \rightarrow state \in \{$*COMMIT, ABORT*$\}$ in the configuration that precedes the step.*

Proof: The only steps that can change $v.update$ are successful freeze CAS steps belonging to some Info object I at line 103, 117 or 118. Consider any such step s. Since the freeze CAS succeeds, we have $v = I.nodes[i]$ for some i and the value of $v.update$ prior to the step is $I.oldUpdate[i]$. Let I' be the Info object that $I.oldUpdate[i].info$ points to. Prior to the creation of I (at line 102), the call of FROZEN on $I.oldUpdate[i]$ at line 97 returned FALSE. So, during the execution of line 90, $I'.state \in \{$COMMIT, ABORT$\}$. Once the state of I' is either COMMIT or ABORT, there is no instruction that can change it to $\perp$ or TRY. Thus, when s occurs, $I'.state \in \{$COMMIT, ABORT$\}$, as required. ■

Lemma 10 *If there is a child* CAS *or commit write that belongs to an Info object I, then there is no abort write or successful abort* CAS *that belongs to I.*

Proof: Suppose there is a child CAS or commit write that belongs to I. Let H be the instance of HELP that performed this step. At line 114 of H, $I.state$ was TRY. Thus, some try CAS belonging to I succeeded. Let *try* be this try CAS. Since there is no instruction that changes $I.state$ to $\perp$, this try CAS must have been the first among all abort CAS and try CAS steps belonging to I. Moreover, no abort CAS belonging to I can ever succeed.

It remains to show that no abort write belongs to I. To derive a contradiction, suppose there is such an abort write in some instance H' of HELP. $I.state$ was TRY when H' executed line 125 prior to doing the abort write. Since *try* is the *first* among all try or abort CAS steps belonging to I, *try* is no later than the execution of line 112 or 113 of H'. Since no other step can change $I.state$ to TRY, $I.state$ must have the value TRY at all times between *try* and the read by H' at line 125. Thus, H' reads $I.state$ to be TRY at line 114 and sets the local variable *continue* to TRUE. Since H' executes the abort write at line 126, H' must have set continue to FALSE at line 119 after reading some value I' different from I in $I.nodes[i] \rightarrow info$ for some i. Let r be this read step.

Since H performs a child CAS or commit write belonging to I, H must have read a pointer to I in $I.nodes[i] \rightarrow update$ at line 119. Thus, some freeze CAS $fcas$ belonging to I on $I.nodes[i]$ succeeded. By Lemma 8, $fcas$ is the first freeze CAS belonging to I on $I.nodes[i]$. So, $fcas$ is no later than the freeze CAS of H' on $I.nodes[i]$. However, $I.nodes[i] \rightarrow update.info \neq I$ when H' reads it on line 119. So a successful freeze CAS belonging to I' must have occurred between $fcas$ and r. By Lemma 9, $I.state \in \{\text{COMMIT}, \text{ABORT}\}$ when this successful freeze CAS occurs. This contradicts the fact that $I.state$ is still TRY when H' performs line 125. ■

Corollary 11 *Once an Info object's state field becomes* ABORT *or* COMMIT*, that field can never change again.*

Proof: No step can change the *state* field to $\perp$. It follows that no try CAS can successfully change the *state* field to TRY, once it has become COMMIT or ABORT. Lemma 10 says that there cannot be two steps in the same execution that set the *state* to ABORT and COMMIT, respectively. ■

We use the notation $\&X$ to refer to a pointer to object X.

Lemma 12 *At all times after a call H to* HELP*($\&I$) reaches line 127, the state field of the Info object I that $infp$ points to is either* ABORT *or* COMMIT*.*

Proof: $I.state$ is initially $\perp$. The first execution of line 112 or 113 belonging to I changes the state to ABORT or TRY, and the state can never be changed back to $\perp$. So, at all times after H has executed line 112 or 113, $I.state \neq \perp$. If the condition at line 122 or 125 of H evaluates to true, then H writes COMMIT or ABORT in $I.state$ at line 124 or 126, respectively. If both conditions evaluate to false, then $I.state$ is either COMMIT or ABORT at line 125. In all three cases, $I.state$ has been either COMMIT or ABORT at some time prior to H reaching line 127. The claim follows from Corollary 11. ■

Lemma 13 *Let I be an Info object other than the dummy Info object. Let C be any configuration. If either*

- *there is some node v, such that $v.update.info$ contains a pointer to I in C, or*
- *some process is executing* HELP*($\&I$) in C,*

then there was a successful freeze CAS *at line 103 belonging to I prior to C.*

Proof: We prove this by induction on the length of the execution that leads to configuration C. If C is the initial configuration, the claim is vacuously satisfied.

Now consider any other configuration C and assume the claim holds for all earlier configurations. It suffices to show that any step s that changes a node's *update* field or invokes HELP preserves the claim.

If s is an invocation of HELP at line 104 then it was clearly preceded by the freeze CAS at line 103. If s is an invocatino of HELP at line 54 or 140, then a pointer to I was read from a node's *update* field at line 52 or 139, respectively, so by the induction hypothesis, the claim holds. If HELP was called at line 98, a pointer to I appeared in a node's *update* field in an earlier configuration by Lemma 6. So, the claim again follows from the induction hypothesis.

If s is an execution of line 103 itself that stores I in some node's *update* field, the claim is obvious. If s is an execution of line 117 or 118 of HELP, then the claim follows from the induction hypothesis (since a process was executing HELP($\&I$) in the configuration preceding s). ■

We next show that the freeze CAS steps belonging to the same Info object occur in the right order.

Lemma 14 *Let I be an Info object. For each $i \geq 2$, a freezing* CAS *belonging to I on $I.nodes[i]$ can occur only after a successful freezing* CAS *belonging to I on $I.nodes[i-1]$.*

Proof: For $i = 2$, since the freezing CAS belonging to I on $I.nodes[2]$ occurs inside HELP, the claim follows from Lemma 13.

If $i > 2$, then prior to the freezing CAS on $I.nodes[i]$ at line 117 or 118, $I.nodes[i-1] \rightarrow update.info$ contains a pointer to I when line 119 is executed in the previous iteration of HELP's while loop. Only a successful freezing CAS on $I.nodes[i-1]$ belonging to I could have put that value there. ■

Lemma 15 *Let I be an Info object. A successful freeze* CAS *belonging to I cannot occur when $I.state =$* ABORT.

Proof: There are no freeze CAS steps of the dummy Info object, by the preconditions to HELP. Consider any other Info object I. When a freeze CAS at line 103 is performed, $I.state = \perp$. Consider a successful freeze CAS $fcas$ that belongs to I inside some call H to HELP. Then the test at line 114 of that call evaluated to true prior to $fcas$, so there is a successful try CAS that belongs to I. Thus, there is no successful abort CAS that belongs to I. It remains to show that no abort write belonging to I occurred before $fcas$.

To derive a contradiction, suppose there was an abort write belonging to I prior to $fcas$. By Lemma 10, there is no commit write belonging to I. Consider the first abort write w belonging to I. Let H' be the call to HELP that performs w. Prior to w, any execution of line 114 would find $I.state =$ TRY. Thus, H' set *continue* to FALSE at line 119 when reading $I.nodes[i] \rightarrow update.info$ for some i. Let r be this read. By Lemma 14, this step is preceded by freeze CAS steps belonging to I on each of $I.nodes[1..i]$. By Lemma 8, $fcas$ cannot be a freeze CAS on any of these nodes, so $fcas$ is a freeze CAS on $I.nodes[j]$ for some $j > i$.

By Lemma 14, there is a successful freeze CAS $fcas'$ on $I.nodes[i]$ belonging to I before $fcas$. By Lemma 8, that CAS precedes the read r by H' of $I.nodes[i] \rightarrow update.info$. Since that read does not find a pointer to I in that field, some other CAS must have changed it between $fcas'$ and r. This contradicts Lemma 9, since r precedes w, the first time $I.state$ gets set to ABORT. ■

Definition 16 *We say that a node v is* frozen for an Info object I *if either*

- *$v.update$ contains* FLAG *and a pointer to I, and $I.state$ is either $\perp$ or* TRY*, or*
- *$v.update$ contains* MARK *and a pointer to I, and $I.state$ is not* ABORT.

Lemma 17 *1. If there is a successful flag* CAS *on node v that belongs to Info object I, then v is frozen for I at all configurations that are after that* CAS *and not after any abort* CAS*, abort write or commit write belonging to I.*

2. If there is a successful mark CAS *on node v that belongs to Info object I, then v is frozen for I at all configurations that are after that* CAS *and not after any abort write belonging to I.*

Proof: 1. It follows from Lemma 9 that $v.update$ cannot change after the successful flag CAS, until an abort CAS, abort write or commit write belonging to I.

2. If there is a successful mark CAS *mcas* belonging to I (at line 4.1), then the state of I was TRY at line 114. Thus, there is no successful abort CAS belonging to I. So, $v.update$ does not change until a commit write or an abort write belonging to I occurs, by Lemma 9. We consider two cases.

   If there is an abort write belonging to I, then there is no commit write belonging to I, so v remains frozen for I in all configurations that are after *mcas* but not after any abort write belonging to I.

   If there is no abort write belonging to I, then the state of I is never set to ABORT. It remains to show that no freeze CAS ever changes $v.update$ after *mcas* changes it to $\langle$MARK, $\&I\rangle$. Note

that no info object I' can have $I'.oldUpdate[i] = \langle \text{MARK}, \&I \rangle$. If there were such an I', then before the creation of I' at line 102, the call to FROZEN($\langle \text{MARK}, \&I \rangle$) on line 97 would have had to return FALSE, meaning that $I.state = \text{ABORT}$, which is impossible. So, no freeze CAS belonging to any Info object I' can change $v.update$ from $\langle \text{MARK}, \&I \rangle$ to some other value. Thus, v remains frozen for I at all times after *mcas*.

■

Corollary 18 *Let v be a node and I be an Info object. If, in some configuration C, $v.update.type =$* MARK *and $v.update.info$ points to I and $I.state =$* COMMIT *then v remains frozen for I in all later configurations.*

Proof: Prior to C there must be a mark CAS that sets $v.update$ to $\langle \text{MARK}, \&I \rangle$. Since $I.state =$ COMMIT, there is no abort write belonging to I, by Lemma 10. So the claim follows from Lemma 17. ■

5.2.3 Behaviour of Child CAS steps

Next, we prove a sequence of lemmas that describes how child pointers are changed. In particular, we wish to show that our freezing scheme ensures that the appropriate nodes are flagged or marked when a successful child CAS updates the tree data structure. Once again, these lemmas are similar to previous work [14, 7], but are included for the sake of completeness.

Lemma 19 *No two Info objects have the same value in the newChild field.*

Proof: Each Info object is created at line 102 of the EXECUTE routine, and no call to EXECUTE creates more than one Info object. Each call to EXECUTE (at line 164 or 190) passes a node that has just been newly created (at line 163 or 185, respectively) as the argument that will become the *newChild* field of the Info object. ■

Lemma 20 *The following are true for every Info object I other than the dummy Info object.*

1. *A successful child* CAS *belonging to I stores a value that has never been stored in that location before.*
2. *If no child* CAS *belonging to I has occurred, then no node has a pointer to $I.newChild$ in its child or prev fields.*

Proof: We prove the lemma by induction on the length of the execution. In an execution of 0 steps, the claim is vacuously satisfied, since there are no Info objects other than the dummy Info object. Suppose the claim holds for some finite execution. We show that it holds when the execution is extended by one step s.

If s creates an Info object (at line 102) of the EXECUTE routine, the node *newChild* was created at line 163 or 185 prior to the call to EXECUTE at line 164 or 190. Between the creation of the node and the creation of the Info object, a pointer to the node is not written into shared memory.

If s creates a new node, it is the execution of line 162, 163 or 185. We must show that none of these nodes contain pointers to $I.newChild$ in their child or *prev* fields, for any I whose first child CAS has not yet occurred. Line 162 creates a leaf whose *prev* field is $\perp$. Line 163 sets one child pointer to *newSibling*, which does not appear in any shared-memory location prior to line 163. The other child pointer and the *prev* field are set to nodes that were obtained from earlier calls to READCHILD and hence read from a *prev* or child field earlier. By induction hypotheses refnewChild-is-new, they cannot be $I.newChild$ for any Info object I whose first child CAS has not occurred. Similarly, when the node is created on line 185, its *prev* and child fields are set to values that were read from *prev* or child fields of other nodes, so the same argument applies.

If s is the first child CAS belonging to I, claim 1 follows from induction hypothesis 2.

If s is not the first child CAS belonging to I, we prove that it is not successful. To derive a contradiction, suppose some earlier child CAS s' belonging to I also succeeded. Both s and s' perform CAS($location, old, new$) steps with identical arguments. Thus *location* stores the value *old* in the configurations just before s' and s (since both CAS steps succeed). By Lemma 4.11, $old \neq new$. So, between s' and s, there must be some child CAS that changes *location* from *new* back to *old*. This violates part 1 of the inductive hypothesis.

If s is a child CAS belonging to some other Info object $I' \neq I$, then it does not write a pointer to $I.newChild$ into any node, by Lemma 19. ■

Corollary 21 *Only the first child* CAS *belonging to an Info object can succeed.*

Proof: Since all child CAS steps belonging to the same Info object try to write the same value into the same location, only the first can succeed, by Lemma 20.1. ■

Lemma 22 *The first child* CAS *belonging to an Info object I occurs while all nodes in $I.nodes$ are frozen for I, including the node $I.par$ to which the child* CAS *is applied.*

Proof: Since there is a child CAS belonging to I, there is no abort write or successful abort CAS belonging to I, by Lemma 10. Prior to the call to CAS-CHILD on line 123 that performed the successful child CAS, the local variable *continue* was true at line 122. This means that a freeze CAS belonging to I succeeded on each entry of $I.nodes[i]$, including $I.par$, by Lemma 4.11. By Lemma 17, these nodes remain frozen for I in all configurations that are after that freeze CAS and not after a commit write belonging to I. The first child CAS that belongs to I is before the first commit write belonging to I. So, the nodes in $I.nodes$ (including $I.par$) are frozen for I when this child CAS occurs. ■

The following lemma shows that marking a node is permanent, if the attempt of the update that marks the node succeeds.

Lemma 23 *If there is a child* CAS *belonging to an Info object I, then for all i, $I.mark[i] \rightarrow update = \langle \text{MARK}, \&I \rangle$ in all configurations after the first such child* CAS.

Proof: By Lemma 22, the claim is true in the configuration immediately after the first child CAS belonging to I. To derive a contradiction, suppose the *update* field of $I.mark[i]$ is later changed. Consider the first such change. This change is made by a successful freezing CAS belonging to some Info object I'. Before I' is created at line 102, FROZEN($\langle \text{MARK}, \&I \rangle$) returns FALSE at line 97, so $I.state = \text{ABORT}$. This contradicts Lemma 10. ■

The next lemma shows that if at some time the *update* field of a node v has the value $I.oldupdate[i]$ for some Info object I and at some later time v is still frozen for I then a child pointer of v can change between these times only by a successful child CAS that belongs to I. (Thus, the freezing works as a 'lock' on the child pointers of the node.)

Lemma 24 *Let I be an Info object and let v be the node that $I.nodes[i]$ points to, for some i. If $v.update = I.oldUpdate[i]$ in some configuration C and $I.info \rightarrow state \in \{\text{COMMIT}, \text{ABORT}\}$ in C, and v is frozen for I in a later configuration C', then the only step between C and C' that might change a child field of v is a successful child* CAS *belonging to I.*

Proof: Since $v.update = I.oldUpdate[i]$ at configuration C, and $v.update = \langle *, I \rangle$ at configuration C', there is a successful freeze CAS $fcas$ that belongs to I on v between C and C'. This freeze CAS uses $I.oldUpdate[i]$ as the expected value of $v.update$. So, by Lemma 7, $v.update = I.oldUpdate[i]$ at

all configurations between C and $fcas$, and $v.update = \langle *, I \rangle$ at all times between $fcas$ and C'. Let I' be the Info object that $I.oldUpdate[i].info$ points to.

By Corollary 21 and Lemma 22, any successful child CAS on v between C and C' must belong to either I' or I. To derive a contradiction, suppose there is such a successful child CAS that belongs to I'. Then by Lemma 10, there is no abort CAS or abort write that belongs to I'. By Lemma 21, this successful child CAS is the first child CAS of I', which is before the first commit write belonging to I'. Thus, $I'.state \notin \{\text{COMMIT}, \text{ABORT}\}$ in C because C is before the successful child CAS, contradicting the hypothesis of the lemma. ■

Lemma 25 *For any Info object I, the first child* CAS *that belongs to I succeeds.*

Proof: Let v be the node that $I.nodes[1]$ points to and let u be the node that $I.oldChild$ points to. The Info object I is created at line 102 of the EXECUTE routine. Before EXECUTE is called at line 164 or 190, there is a call to VALIDATELEAF on line 157 or 179, respectively. VALIDATELEAF calls VALIDATELINK, which returns TRUE. This VALIDATELINK reads a value from $v.update$ that is ultimately stored in $I.oldUpdate[1]$ and then checks on line 53 that $v.update.state \notin \{\perp, \text{TRY}\}$ when $v.update$ was read on line 52. Let C be the configuration after this read. After C, on line 57, the value u is read from a child field of v.

Let C' be the configuration just before the first child CAS belonging to I. By Lemma 22, v is frozen for I in C'. So, by Lemma 24, there is no change to v's child fields between C and C'. Moreover, u is read from a child field of v during this period, and the first child CAS of I uses u as the old value, so it will succeed. ■

5.2.4 Tree Properties

In this section, we use the lemmas from the previous sections to begin proving higher-level claims about our particular data structure, culminating in Lemma 34, which proves that SEARCHES end up at the correct leaf, and Lemma 36, which proves that all versions of the tree are BSTs.

Our data structure is persistent, so it is possible to reconstruct previous versions of it. Consider a configuration C. For any Internal node v whose sequence number is at most ℓ, we define the *version-ℓ left (or right) child of v* to be the node that is reached by following the left (or right) child pointer of v and then following its *prev* pointers until reaching the first node whose *seq* field is less than or equal to ℓ. (We shall show that such a node exists.) We define $D_\ell(C)$ as follows. The nodes of $D_\ell(C)$ is the set of all existing nodes in C and the edges go from nodes to their version-ℓ children; $T_\ell(C)$ is the subgraph of $D_\ell(C)$ containing those nodes that are reachable from the *Root* in $D_\ell(C)$. We use the notation $T_\infty(C)$ to represent the graph of nodes reachable from the *Root* by following the current child pointers. We shall show that $T_\ell(C)$ is a binary search tree rooted at *Root*.

Definition 26 *We say a node is* inactive *when it is first created. If the node is created at line 163 or 185, it becomes active when a child* CAS *writes a pointer to it for the first time, and it remains active forever afterwards. If the node is created at line 161 or 162, then it becomes active when a child* CAS *writes a pointer to its parent for the first time, and it remains active forever afterwards. The nodes that are initially in the tree are always active.*

Definition 27 *An ichild* CAS *is a child* CAS *belonging to an Info object that was created by an* INSERT *and a dchild* CAS *is a child* CAS *belonging to an Info object that was created by a* DELETE.

Lemma 28 *1. If a node is inactive, then there is no pointer to it in the prev field of any node or in a child field of an active node.*

2. The first argument of each call to READCHILD *and* SCANHELPER *is an active node.*

3. *No call to* ReadChild *or* Search *returns an inactive node.*

4. *For each Info object I, $I.nodes$ contains only active nodes.*

Proof: We prove the claim by induction on the length of the execution. The claim is vacuously satisfied for an execution of length 0. Assume the claim holds for some execution. We prove that it holds when the execution is extended by one step s.

1. When the *prev* field of a node is set at line 163, it points to a node returned by the Search on line 156, so it is active by inductive hypothesis 3. When the *prev* field of a node is set at line 185, it points to a node returned by the ReadChild on the previous line, which is active by inductive hypothesis 3.

   If s is a successful child CAS that changes a child pointer to point to a node v, v is active after the child CAS, by definition. If v was created at line 163, its children become active at the same time as v. If v was created at line 185, any children it has were copied from the children fields of an active node by induction hypothesis 1, so they were already active when v was created.

2. If s is a call to ReadChild on line 39, the first argument is either the root node, which is active, or the result of a previous call to ReadChild, which is active by inductive hypothesis 3. If s is a call to ReadChild on line 182, the first argument was returned by Search on line 178, so it is active by inductive hypothesis 3. If s is a call to ReadChild on line 141 to 144, then the first argument is the first argument of the call to ScanHelper, so it is active by inductive hypothesis 2.

   If s is a call to ScanHelper on line 132, the first argument is the root node, which is active. If s is a call to ScanHelper on line 141 to 144, the first argument was returned by a call to ReadChild, which was active by inductive hypothesis 3.

3. Suppose s is the return statement of a ReadChild. When that function was called, the first argument was an active node, by inductive hypothesis 2. The node returned by ReadChild is reached from that node by following child and prev pointers, so it follows from inductive hypothesis 1 that the resulting node is active too.

   Suppose s is the return statement of a Search. Each node returned is either the root, which is active, or obtained as the result of a ReadChild at line 39 during the Search, which is active by inductive hypothesis 3.

4. Suppose s is a step that creates an Info object I at line 102 of Execute. If Execute was called at line 164, then the elements of $I.nodes$ were returned by the Search on line 156, so they are active by inductive hypothesis 3. If Execute was called at line 190, then the elements of $I.nodes$ were returned by the Search on line 156 or the ReachChild on line 182, so they are active by inductive hypothesis 3.

■

The following Lemma shows that the effect of a child CAS step is as shown in Figure 1.

Lemma 29 *Consider a successful child* CAS *step s that belongs to some Info object I. Let C and C' be the configurations before and after s. Then,*

1. *In C, $I.oldChild \rightarrow update = \langle \text{Mark}, \&I \rangle$.*

2. *In C, $I.newChild$ is inactive.*

3. *If s is an ichild* CAS *created by an* INSERT*(k) then $I.newChild$ is an internal node and its two children in C' are both leaves, one of which has the same key as $I.oldChild$ and the other has the key k.*

4. *If s is a dchild* CAS *created by a* DELETE*(k) operation then $I.oldChild$ is an Internal node and in configuration C:*

   - *one of its children is $I.nodes[3]$, which is a leaf containing the key k, and*
   - *the other child is $I.nodes[4]$, which has the same key and children as $I.newChild$, and*
   - *both of the children of $I.oldChild$ have $\langle$MARK, $\&I\rangle$ in their update fields.*

Proof: By Corollary 21, s is the first child CAS belonging to I.

1. By Lemma 4.11, $I.oldChild$ is in $I.mark$, which is a subset of $I.nodes$. So, by Lemma 22, $I.oldChild$ is frozen for I at C and it must have been a mark CAS that froze the node.

2. Note that $I.newChild$ was created at line 163 if s is an ichild CAS, or at line 185 if s is a dchild CAS. By Lemma 19, s is the first child CAS that writes a pointer to $I.newChild$, so this node becomes active for the first time in C'.

3. $I.newChild$ was created at line 163, with its children satisfying the claim, and the children pointers cannot be changed before $I.newChild$ becomes active at C', by Lemma 28.4.

4. Since s is a dchild CAS, I was created by an EXECUTE routine called at line 190 of a DELETE(k) operation. $I.oldChild$ is copied from the local variable p of that DELETE. $I.oldUpdate[2]$ was read from the *update* field of $I.oldChild$ inside the call at line 179. Since that call to VALIDATELEAF returned $\langle$TRUE, $I.oldUpdate[2]\rangle$, $I.oldChild.info$ was found to be an Info object that was in state ABORT or COMMIT. Subsequently, the two child fields of $I.oldChild$ were read (inside the same call to VALIDATELEAF and at line 183) and were seen to be equal to l and *sibling*. By Lemma 24, these are still the children of $I.oldChild$ in C since $I.oldChild = I.nodes[2]$ is frozen for I in C, by Lemma 22. By the exit condition at line 36 of the SEARCH called at line 178, l is a leaf. Furthermore, $l \rightarrow key = k$, since the test at line 181 evaluated to FALSE.

   The key and children of $I.newChild$ are copied from *sibling*. If *sibling* is a leaf, then $I.newChild$ is also a leaf, so there is nothing further to prove. If *sibling* is an internal node, it remains to prove that the children of *sibling* do not change between the time they are copied at line 185 and C. This is because the call to VALIDATELINK (at line 187) read $I.oldUpdate[4]$ from $sibling \rightarrow update$ and then sees that the Info object that field points to is in state COMMIT or ABORT. Subsequently the children of *sibling* are seen to be the two children of $I.newChild$ inside the calls to VALIDATELINK at line 187 and 188. By Lemma 24, these are still the children of *sibling* in C since $sibling = I.nodes[4]$ is frozen for I in C, by Lemma 22.

   Both l and *sibling* are included in $I.nodes$. By Lemma 22 they are both frozen for I at configuration C. Since they are also in $I.mark$, they were frozen for I by a mark CAS, so their update fields are $\langle$MARK, $\&I\rangle$.

■

By Observation 1, no step changes a *prev* pointer of an existing node. The only step that changes a child field of a node is a successful child CAS. Thus, the following lemma provides a complete description of how T_i can be changed by any step. It also characterizes which nodes are in different tree versions T_i: roughly speaking, if a node is flagged, then it is still in all versions of the tree, but if it is marked for removal, it will be in all versions of the tree if the corresponding child CAS has not yet occurred, but it will only be in old versions after the child CAS has removed it.

Lemma 30 *The following statements hold.*

1. *For each successful child* CAS *that belongs to some Info object I and takes the system from configuration C to C', the following statements are true.*
   (a) *For all $i < I.seq$, $T_i(C) = T_i(C')$.*
   (b) *If I was created by an* INSERT*(k), then for all $i \geq I.seq$, $T_i(C')$ is obtained from $T_i(C)$ by replacing the leaf $I.oldChild$ by $I.newChild$, which is an internal node whose children are two leaves with keys $I.oldChild \rightarrow key$ and k. (If $I.oldChild$ is not in $T_i(C)$, then this replacement has no effect on T_i.)*
   (c) *If I was created by a* DELETE*(k), then for all $i \geq I.seq$, $T_i(C')$ is obtained from $T_i(C)$ by replacing the internal node $I.oldChild$ and its two children (which are a leaf containing k and a node sibling) by a copy of $I.newChild$, whose key is sibling.key and whose children are the same as sibling's children. (If $I.oldChild$ is not in $T_i(C)$, then this replacement has no effect on T_i.)*
2. *For every configuration C', and for each node v that is active in C', and for all $i \geq v.seq$, the following statements are true.*
   (a) *If $v.update.type =$* FLAG *in C' then v is in $T_i(C')$.*
   (b) *If $v.update = \langle$*MARK*$, \&I\rangle$ in C' and no child* CAS *that belongs to I has occurred before C', then v is in $T_i(C')$.*
   (c) *If $v.update = \langle$*MARK*$, \&I\rangle$ in C' and $i < I.seq$, then v is in $T_i(C')$.*

Proof: We prove the claim by induction on the length of the execution. First consider an execution of 0 steps. Claim 1 is satisfied vacuously. In the initial configuration C_0, all nodes are active, flagged with the dummy Info object, have sequence number 0, and are in $T_i(C_0)$ for all i, so claim 2 is true.

Now, suppose the claim holds throughout some finite execution. We prove the claim holds for any extension of that execution by a single step s.

1. Claim 1 for all successful child CAS steps prior to s follows from induction hypothesis 1. So it suffices to prove claim 1 holds for s if s is a successful child CAS belonging to some Info object I.
   (a) When I is created, $I.newChild$ is given the sequence number $I.seq$. Thus, when s swings a child pointer from $I.oldChild$ to $I.newChild$ it does not affect T_i for $i < I.seq$, since $I.newChild \rightarrow prev = I.oldChild$, by Lemma 4.11.
   (b) Suppose I was created by an INSERT(k) operation. Consider any $i \geq I.seq$. The step s changes a child pointer of some node p from $I.oldChild$ to $I.newChild$. By Lemma 28.4, p is active in C, so we can apply induction hypothesis 2 to it. By Lemma 22, p is frozen for I in C. Since p is not in $I.mark$, $p.update.type =$ FLAG. Moreover, $i \geq I.seq \geq p.seq$ by Invariant 5. So, by induction hypothesis 2a, p is in $T_i(C)$. Claim 1b follows from Lemma 29.3, since $I.newChild \rightarrow seq = I.seq \leq i$.
   (c) Suppose I was created by a DELETE(k) operation. Consider any $i \geq I.seq$. The step s changes a child pointer of some node gp from $I.oldChild$ to $I.newChild$. By Lemma 28.4, $gp = I.nodes[1]$ is active in C, so we can apply induction hypothesis 2 to it. By Lemma 22, gp is frozen for I in C. Since gp is not in $I.mark$, $gp.update.type =$ FLAG. Moreover, $i \geq I.seq \geq gp.seq$ by Invariant 5. So, by induction hypothesis 2a, gp is in $T_i(C)$. Claim 1c follows from Lemma 29.4, since $I.newChild \rightarrow seq = I.seq \leq i$.
2. Induction hypothesis 2 establishes the claim for all configurations prior to the final step s, so it suffices to prove the claim for the configuration C' after s. Let C be the configuration before s. Let v be any node that is active in C' and let $i \geq v.seq$.

(a) Suppose $v.update.type = \text{FLAG}$ in C'. We consider four cases.

- Suppose s is the child CAS that makes v active. Then, by Lemma 22 the node p whose child pointer is modified by s is flagged in C. Let I be the Info object that s belongs to. By induction hypothesis 2a, p is in $T_i(C)$ since $i \geq v.seq = I.seq \geq p.seq$ by Lemma 5. So, the node v is in $T_i(C')$ since there is now a path of child pointers from p to v of nodes whose sequence numbers are $v.seq$.
- Suppose v is active in C and s is a successful flag CAS on v. Let I be the Info object that s belongs to. Let up be the value stored in $v.update$ in C.
  If $up.type = \text{FLAG}$, then by induction hypothesis 2a, v was in $T_i(C)$, so it is in $T_i(C')$. Now suppose $up = \langle \text{MARK}, \&I' \rangle$ for some Info object I'. Since s belongs to I, $v = I.nodes[j]$ for some j and $up = I.oldUpdate[j]$. Prior to the creation of I at line 102, the call to FROZEN(up) at line 97 returned FALSE. So, the test at line 90 found $I'.state = \text{ABORT}$. By Lemma 10, there is no child CAS belonging to I'. So by induction hypothesis 2b, v is in $T_i(C)$, so it is also in $T_i(C')$.
- Suppose v is active in C and s is a successful child CAS. If $i < I.seq$, then $T_i(C) = T_i(C')$ (by claim 1a proved above), so claim 2a follows from induction hypothesis 2a. Now suppose $i \geq I.seq$. If s is an ichild CAS, then by claim 1b, proved above, the only node that s removes from T_i is $I.oldChild$, which is marked for I in C and is therefore not v (since v is flagged in C). If s is a dchild CAS, then by claim 1c, proved above, the only nodes that s removes from T_i are $I.oldChild$ and its children. By Lemma 29.4, these nodes are the three nodes in $I.mark$. So by lemma 22, they are marked in C' and are therefore not equal to v. In either case, claim 2a follows from induction hypothesis 2a.
- Suppose v is active in C and s is any other step. Then the truth of claim 2a follows from induction hypothesis 2a.

(b) Suppose that $v.update = \langle \text{MARK}, \&I \rangle$ in C' and no child CAS belonging to I has occurred before C'. Then, v is active in C since, immediately after the child CAS that makes v active, v is flagged for the dummy object. We consider three cases.

- Suppose s is a successful mark CAS on v. Then this mark CAS belongs to I since $v.update = \langle \text{MARK}, \&I \rangle$ in C'. Let up be the value stored in $v.update$ in configuration C. If $up.type = \text{FLAG}$, then claim 2b follows from induction hypothesis 2a. Now suppose $up = \langle \text{MARK}, \&I' \rangle$ for some Info object I'. Prior to creating I at line 102, FROZEN(up) at line 97 returned FALSE. Thus, $I'.state$ was ABORT. By Lemma 10, there is no child CAS belonging to I'. So, v is in $T_i(C')$ by inductive hypothesis 2b.
- Suppose s is a successful child CAS. This child CAS must belong to some Info object $I' \neq I$, since we assumed that no child CAS of I occurs before C'. The argument that v is in $T_i(C')$ is identical to the argument for the third case of 2a, above.
- Suppose s is any other step. Then claim 2b follows from induction hypothesis 2b.

(c) Suppose that $v.update = \langle \text{MARK}, \&I \rangle$ in C' and $i < I.seq$. We consider four cases.

- Suppose s is a successful mark CAS on v. The argument that v is in $T_i(C')$ is identical to the argument for the first case of 2b, above.
- Suppose s is a successful child CAS that belongs to I. By Corollary 21, there is no child CAS belonging to I before C. By induction hypothesis 2b, v is in $T_i(C)$. By claim 1a, proved above, $T_i(C) = T_i(C')$. So, v is in $T_i(C')$.
- Suppose s is a successful child CAS that belongs to some Info object $I' \neq I$. The argument that v is in $T_i(C')$ is identical to the argument for the third case of 2a, above.
- Suppose s is any other step. Then the truth of claim 2c follows from induction hypothesis 2c.

■

Corollary 31 *Let v be a node that is active in some configuration C. Then, for every $i \geq 0$, if v is in the left (or right) subtree of a node v' with key k within tree $T_i(C')$ for some later configuration C', then v was in the left (or right, respectively) subtree of a node with key k within tree $T_i(C)$.*

Proof: This follows immediately from Lemma 30.1. ■

Given a binary tree (which may or may not be a BST), we define the *search path for a key k* to be the path that begins at the root and, at each node, passes to the left or right child, depending on whether k is less than the key in the node or not.

Lemma 32 *If, for each $i \geq 0$, a node v is on the search path for key k in $T_i(C)$ for some configuration C and is still in $T_i(C')$ for some later configuration C', then v is on the search path for k in $T_i(C')$.*

Proof: This follows immediately from Corollary 31. ■

Lemma 33 *A call to* ReadChild*$(p, left, seq)$ returns the version-seq left (or right) child of the node pointed to by p at the time line 45 is executed if left is* True *(or* False*, respectively).*

Proof: This follows immediately from the fact that *prev* fields of nodes never change (by Observation 1). ■

Whenever Search(k, seq) reads a left (or right) child field of a node v on line 45 then we say that the Search *visits* the version-*seq* left (or right, respectively) child of v. (Notice that the time a node is visited is earlier than the time that local variable ℓ of Search points to this node.) We also say that a Search *visits* the root when it executes line 35.

Lemma 34 *Consider any instance S of* Search*(k, seq) that terminates, and let $v_1, \ldots, v_k$ be the nodes visited by S (in the order they are visited). There exist configurations $C_1, C_2, \ldots, C_k$ such that*

1. *C_1 is after the search is invoked,*
2. *for $i > 1$, C_{i-1} is before or equal to C_i,*
3. *v_i is on the search path for k in $T_{seq}(C_i)$,*
4. *C_i is before the step where S visits v_i, and*
5. *C_i is the last configuration that satisfies both (3) and (4).*

Proof: Since v_1 is the root node, which is visited when S executes line 35, let C_1 be the configuration before S executes line 35. This satisfies all claims (including 2, vacuously).

Let $1 < i \leq k$ and suppose C_{i-1} has already been defined to satisfy all of the claims. Let C' be the configuration before S visits v_i by reading a child pointer of v_{i-1}. Note that C_{i-1} is before C' by induction hypothesis 4. We first show that v_i is on the search path for k in T_{seq} at some configuration between C_{i-1} and C' by considering two cases.

Case 1 (v_{i-1} is in $T_{seq}(C')$). Then, by induction hypothesis 3 and Lemma 32, v_{i-1} is on the search path for k in $T_{seq}(C')$. So, v_i is also on the search path for k in $T_{seq}(C')$.

Case 2 (v_{i-1} is not in $T_{seq}(C')$). Let C'' be the last configuration between C_{i-1} and C' when v_{i-1} was in $T_{seq}(C'')$. By Lemma 32, v_{i-1} is on the search path for k in $T_{seq}(C'')$. The step after C'' must be a child CAS that removes v_{i-1} from T_{seq}. By Lemma 23, v_{i-1} is marked at all times after C''. By Lemma 21 and 22, the child pointers of v_{i-1} are never changed after C''. Since *prev* pointers of nodes never change either, the version-*seq* children of v_{i-1} never change after C''. Thus, v_i is already the

version-*seq* child of v_{i-1} at configuration C'' since v_i is the version-*seq* child of v_{i-1} at C' after C'' by Lemma 33. Thus, v_i is on the search path for k in $T_{seq}(C'')$.

Thus, in either case, there is a configuration between C_{i-1} and S's visit to v_i when v_i is on the search path for k in T_{seq}. Let C_i be the last such configuration. The claims follow. ■

Invariant 35 *Let C be any configuration and let $j \leq i$. Suppose the search path for a key k in $T_j(C)$ includes a node v and $v \in T_i(C)$. Then the search path for k in $T_i(C)$ also includes v.*

Proof: The claim is true for the initial configuration C_0, since $T_j(C_0) = T_i(C_0)$. We show that every step preserves the invariant. The only step that changes a tree or search path is a successful child CAS. Consider a successful child CAS belonging to some Info object I. It changes a child pointer from $I.oldChild$ to $I.newChild$. We consider three cases.

- Suppose $I.newChild \rightarrow seq > i$. Then by Lemma 4.11, neither T_i nor T_j change, so the invariant is preserved.
- Suppose $j \leq I.newChild \rightarrow seq \leq i$. Let C and C' be the configurations before and after the successful child CAS.

  If the child CAS is a dchild CAS, then by Lemma 30, T_j is not affected, while in T_i, a parent x and its children y and leaf z are replaced by a copy y' of y, so that x, y and z are no longer in $T_i(C')$. Thus, any search path that passed through x in $T_i(C)$ will now instead pass through the new node y' in $T_i(C')$. If the search path continued to a child of y in $T_i(C)$, it will continue to the same child of y' in $T_i(C')$. Thus, the invariant is preserved.

  If the child CAS is an ichild CAS, then by Lemma 30, T_j is not affected, while in T_i, a leaf x is replaced by an internal node with two leaf children. The old leaf is no longer in the tree $T_i(C')$. Thus, all search paths in T_i are unaffected, except those that pass through x, but since x is not in $T_i(C)$, the invariant is still true for C'.
- Suppose $I.newChild \rightarrow seq \leq j$. Then applies an identical change to both T_i and T_j, so the invariant is preserved.

■

Invariant 36 *For every configuration C and every integer $i \geq 0$, $T_i(C)$ is a BST.*

Proof: The claim is true in the initial configuration. The only steps that can modify T_i are successful child CAS steps, so we show that each successful child CAS preserves the invariant. Let I be the Info object that this child CAS belongs to and let $j = I.seq$. If $i < j$ then the child CAS does not affect T_i, by Lemma 30.1a. So suppose $i \geq j$.

First, consider a dchild CAS. By Lemma 30.1c, the change to T_i preserves the invariant.

Now, consider an ichild CAS. I was created by an INSERT(k) operation. The change that this ichild CAS can make to T_i is described by Lemma 30.1b: it replaces a leaf l with key k' by an internal node with two children whose keys are k and k'. By Lemma 34, l was on the search path for k in T_j in some configuration during the SEARCH(k, j) at line 156 of the INSERT. By Lemma 22, l is marked for I when the child CAS occurs. By Lemma 30.2b, l is still in T_j in the configuration prior to the child CAS. By Lemma 32, l is still on the search path for k in T_j in that configuration. By Invariant 35, l is also on the search path for k in T_i in that configuration. Thus, the change to T_j, as described by Lemma 30.1b preserves the BST invariant because the key k is being inserted at the correct location in T_j. ■

5.2.5 Linearizability

Finally, we are ready to prove that the implementation is linearizable. We do this by defining linearization points for all operations and proving Lemma 42, which describes how the current state of the data structure reflects the abstract set that would be obtained by performing all of the operations that have been linearized so far atomically at their linearization points. This connection between the states of the actual data structure and the abstract set also allows us to show that the results of all operations are consistent with this linearization.

We first show that HELP returns an appropriate response that indicates whether the update being helped has succeeded.

Lemma 37 *Consider any call H to* HELP *that is called with a pointer to an Info object I.*

1. *If H returns* TRUE *then there is a unique successful child* CAS *that belongs to I, and that child* CAS *occurs before H terminates.*

2. *If H returns* FALSE *then there is no successful child* CAS *that belongs to I.*

3. *If H does not terminate then there is at most one successful child* CAS *that belongs to I.*

Proof: 1. Suppose H returns TRUE. Then, $I.state$ = COMMIT at line 127. So some call to HELP($\&I$) performed a commit write at line 124 prior to H's execution of line 127. Prior to that, the same call to HELP performed a child CAS belonging to I. By Lemma 25, the first such child CAS succeeds. By Lemma 21, there is exactly on successful child CAS belonging to I.

2. Suppose H returns FALSE. By Lemma 12, when H reaches line 127, the $I.state$ must be ABORT or COMMIT. Since H returns FALSE, $I.state$ is ABORT at line 127. By Lemma 10, there is no child CAS that belongs to I.

3. This claim follows immediately from Lemma 21.

■

Next, we use the preceding Lemma to argue that each update returns an appropriate response, indicating whether the update has had an effect on the data structure.

Lemma 38 *Consider any call U to* INSERT *or* DELETE.

1. *If U does not terminate then there is at most one successful child* CAS *that belongs to any Info object created by U. If there is such a child* CAS*, it belongs to the Info object created in the last iteration of U's while loop.*

2. *If U returns* TRUE *then there is exactly one successful child* CAS *that belongs to any Info object created by U, and it belongs to the Info object created in the last iteration of U's while loop.*

3. *If U returns* FALSE *then there is no successful child* CAS *that belongs to any Info object created by U.*

Proof: For each iteration of U's while loop except the last, either EXECUTE is not called or EXECUTE returns FALSE. If EXECUTE returns FALSE, then either EXECUTE does not perform the first freezing CAS successfully at line 103 or the call to HELP returns FALSE. If the first freezing CAS does not succeed, no process can call HELP on the Info object created in this iteration of U. If HELP returns FALSE, there is no child CAS belonging to the Info object created in this iteration of U's loop, by Lemma 37. Thus, in all cases, there is no child CAS belonging to an Info object created in this iteration of U's loop.

The final iteration of U's loop can create at most one Info object, which has at most one successful child CAS, by Lemma 21. This establishes claim (1) of the lemma.

If U returns TRUE, then U's call to EXECUTE on line 164 or 190 returns true. This means that the call to HELP on line 104 of EXECUTE returns true. By Lemma 37, there is exactly one successful child CAS that belongs to the Info object created in the final iteration of U's loop. This completes the proof of claim (2).

If U returns FALSE, then either EXECUTE is not called at line 164 or 190, or that call to EXECUTE returns FALSE. By the same argument as in the first paragraph of this proof, there is no child CAS associated with the Info object created in the final iteration of U's while loop. ■

Next, we describe how operations of an execution are linearized. For the remainder of the proof, we fix an execution α.

If there is a successful child CAS that belongs to an Info object I created by an INSERT or DELETE operation, we linearize the operation at the first freeze CAS belonging to I (at line 103). There is at most one such successful child CAS, by Lemma 38 and if such a child CAS exists, it is preceded by a freezing CAS, by Lemma 22, so this defines a unique linearization point for each update operation that has a successful child CAS. In particular, this defines a linearization point for every update operation that returns TRUE and some that do not terminate, but it does not define a linearization point for any update that returns FALSE, by Lemma 38.

We linearize each INSERT that returns FALSE, each DELETE that returns FALSE and each FIND that terminates in the operation's last call to VALIDATELEAF at line 157, 179 or 76, respectively. More specifically, we linearize the operation when *pupdate* is read at line 66 of that call to VALIDATELEAF.

For each completed RANGESCAN operation, we define its sequence number to be the value it reads from *Counter* at line 130. We linearize every RANGESCAN operation with sequence number i at the step that the *Counter* value changes from i to $i+1$ with ties broken in an arbitrary way. Note that this step is well-defined and occurs during the execution interval of the RANGESCAN: after the RANGESCAN reads i from *Counter*, some process must increment *Counter* from i to $i+1$ no later than the RANGESCAN's own increment at line 131.

In the following, we define an update operation to be imminent if its linearization point has occurred, but it has not yet made the necessary change to the data structure.

Definition 39 *An update operation is called* imminent *in a configuration C of execution α if, for some Info object I created by the update,*

- *there is a freezing* CAS *belonging to I before C,*
- *there is no child* CAS *belonging to I before C, and*
- *there is a child* CAS *belonging to I after C.*

The following lemma is a consequence of the way that update operations must freeze nodes in order to apply changes.

Lemma 40 *In any configuration C, there cannot be two imminent updates with the same key.*

Proof: To derive a contradiction, suppose there are two update operations op_1 and op_2 with the same key that are both imminent in C. Let I_1 and I_2 be the two Info objects that satisfy definition 39. Let gp_1, p_1, l_1 and gp_2, p_2, l_2 be the results of the last SEARCH performed by the two operations prior to creating I_1 and I_2, respectively.

$I_1.nodes$ includes p_1 and $I_2.nodes$ includes p_2. By Lemma 24, l_1 is the child of p_1 at configuration C. Similarly, l_2 is the (same) child of p_2 at configuration C. By Lemma 34, p_1 and l_1 were all on the search path for k in T_∞ at some time before C. By Lemma 30.2, they are still on the search path

for k in the configuration prior to the successful child CAS of I_1. So, by Lemma 32, they are on the search path for k in C. A similar argument shows that p_2 and l_2 are on the search path for k in C. So, $l_1 = l_2$ and $p_1 = p_2$.

Since p_1 appears in both I_1 and I_2 there must be a successful freezing CAS belonging to each of I_1 and I_2 on this node, by Lemma 22. Let $fcas_1$ and $fcas_2$ be the steps that freeze p_1 for I_1 and I_2, respectively. Without loss of generality, assume $fcas_1$ occurs before $fcas_2$. Then, op_2 reads a value up from $p_1.update$ and stores the result in $I.oldUpdate$ *after* $fcas_1$; otherwise $fcas_2$ would fail, by Lemma 7. After op_2 reads this field, it gets the result FALSE from FROZEN(up) at line 97 (otherwise the attempt would be aborted before I_2 is created at line 102). Thus, $I_1.state$ must be ABORT or COMMIT when FROZEN checks this field. By Lemma 10, $I_1.state$ cannot be ABORT because there is a child CAS that belongs to I_1. Thus, there is a commit write belonging to I_1 prior to op_2's creation of I_2. By the code, there is a child CAS belonging to I_1 prior to the creation of I_2. This contradicts the fact that the first child CAS of I_1 occurs after C but the first freezing CAS belonging to I_2 occurs before C. ■

The following lemma shows will be used to argue about the linearization point of a FIND or an unsuccessful update operation, using the fact that VALIDATELEAF has returned true.

Lemma 41 *If a call* VALIDATELEAF(gp, p, l, k) *returns* $\langle$TRUE$, gpupdate, pupdate\rangle$ *then in the configuration C immediately before it reads $p.update$ at line 66, the following statements hold.*

1. *Either ($k < p.key$ and $p.left = l$) or ($k \geq p.key$ and $p.right = l$).*
2. *$p.update = pupdate$ and $pupdate$ is not frozen.*
3. *If $p \neq Root$, either ($k < gp.key$ and $gp.left = p$) or ($k \geq gp.key$ and $gp.right = p$).*
4. *If $p \neq Root$, $gp.update = gpupdate$ and $gpupdate$ is not frozen.*

Proof: Since VALIDATELEAF returns TRUE, its calls to VALIDATELINK return TRUE.

1. Consider the call to VALIDATELINK at line 64 of VALIDATELEAF. At line 52, $p.update = pupdate$. Later, $p.update = pupdate$ at C. By Lemma 7, $p.update$ was equal to $pupdate$ throughout that period. Node p was not frozen at line 53, so no changes to p's children occurred between that time and C, by Lemma 22. Claim (1) was true when line 57 was performed during that interval, so it is still true at C.

2. Since VALIDATELEAF returns TRUE, $p.update = pupdate$ when it is read at line 66 just after configuration C. Moreover, $pupdate$ was not frozen during the call to VALIDATELINK at line 64 before C, so it is still not frozen in C, by Corollary 11.

3. Suppose $p \neq Root$. Consider the call to VALIDATELINK at line 65 of VALIDATELEAF. At line 52, $gp.update = gpupdate$. Later, $gp.update = gpupdate$ at the read of $gp.update$ on line 66, which is after C. By Lemma 7, $gp.update$ was equal to $gpupdate$ throughout that period (including at C). Node gp was not frozen at line 53, so no changes to gp's children occurred between that time and C, by Lemma 22. Claim (3) was true when line 57 was performed during that interval, so it is still true at C.

4. Before C, $gp.update = gpupdate$ at line 52 of the call to VALIDATELINK on line 65. Since VALIDATELEAF returns TRUE, $gp.update = gpupdate$ when it is read at line 66 after configuration C. By Lemma 7, $gp.update = gpupdate$ at configuration C. Moreover, $gpupdate$ was not frozen during the call to VALIDATELINK at line 65 before C, so it is still not frozen in C, by Corollary 11.

■

Now, we are ready to establish the connection between the state of the shared data structure and the abstract set that it represents (according to the operations that have been linearized so far). For any configuration C of execution α, let

$$\begin{aligned} L(C) &= \{k : \text{there is a leaf of } T_\infty(C) \text{ with key } k\} \\ I_{ins}(C) &= \{k : \text{there is an imminent } \textsc{Insert}(k) \text{ in } C\} \\ I_{del}(C) &= \{k : \text{there is an imminent } \textsc{Delete}(k) \text{ in } C\} \\ Q(C) &= (L(C) \cup I_{ins}(C)) - I_{del}(C) \end{aligned}$$

Let $S(C)$ be the set of keys that would result if all update operations whose linearization points are before C were performed atomically in the order of their linearization points.

Lemma 42 *For all configurations C in execution α,*

1. $Q(C) = S(C) \cup \{\infty_1, \infty_2\}$,
2. $I_{ins}(C) \cap L(C) = \{\}$,
3. $I_{del}(C) \subseteq L(C)$, *and*
4. *If a* Find, Insert *or* Delete *operation that terminates in α is linearized in the step after configuration C, the output it returns is the same as if the operation were done atomically on a set in state $S(C)$.*

Proof: We prove the claim holds for all states and linearization points in a prefix of the execution, by induction on the length of the prefix.

For the base case, consider the prefix of 0 steps. In the initial configuration C, we have $Q(C) = \{\infty_1, \infty_2\}$, $I_{ins}(C) = I_{del}(C) = S(C) = \{\}$. There are no linearization points, so claim 4 holds vacuously.

Assume the claim is true for a prefix α'. We prove that it holds for $\alpha' \cdot s$ where s is the next step of α. Let C and C' be the configurations before and after s. We consider several cases.

- Suppose s is the first freezing CAS of an Info object that has a child CAS later in α and the Info object is created by an $\textsc{Insert}(k)$ operation. This is the linearization point of the Insert. So, $S(C') = S(C) \cup \{k\}$. We have $L(C') = L(C)$, $I_{ins}(C') = I_{ins}(C) \cup \{k\}$ and $I_{del}(C') = I_{del}(C)$. By Lemma 40, $k \notin I_{del}(C')$, so $Q(C') = Q(C) \cup \{k\}$. Thus, s preserves claims 1 and 3.

  Let gp, p and l be the three nodes returned by the last Search at line 156 of the Insert. By Lemma 34, p and its child l were on the search path for k in T_∞ in some earlier configuration. Since p is flagged for the Insert in C', it follows from Lemma 30.2a that p is still in the tree at C'. By Lemma 24, its child is still l at C'. Thus, l is still on the search path for k in C', by Lemma 32. Since the test on line 159 evaluated to False, $l.key \neq k$. By Lemma 36, the tree $T_\infty(C')$ is a BST, so k does not appear anywhere else in it. Thus, $k \notin L(C') = L(C)$. This ensures claim 2 is preserved in C'.

  By Lemma 40 applied to C', there is no imminent $\textsc{Insert}(k)$ in C. So, $k \notin Q(C)$. By the induction hypothesis, $k \notin S(C)$. So, an $\textsc{Insert}(k)$ performed on a set in state $S(C)$ would return True. By Lemma 37 the Insert$k$ linearized at s also returns True, establishing claim 4.

- Suppose s is the first freezing CAS of an Info object that has a child CAS later in α and the Info object is created by a $\textsc{Delete}(k)$ operation. This is the linearization point of the Delete. So, $S(C') = S(C) - \{k\}$. We have $L(C') = L(C)$, $I_{ins}(C') = I_{ins}(C)$ and $I_{del}(C') = I_{del}(C) \cup \{k\}$. So $Q(C') = Q(C) - \{k\}$. Thus, s preserves claim 1 and 2.

Let gp, p and l be the three nodes returned by the last SEARCH at line 178 of the DELETE. By Lemma 34, these three nodes were on the search path for k in T_∞ in some earlier configuration. The node gp is flagged for the DELETE in C' and, by Lemma 24, p is still the child of gp and l is still the child of p in C'. It follows from Lemma 30.2a that gp is still in the tree at C'. Thus, l is still on the search path for k in C', by Lemma 32. Since the test on line 181 evaluated to FALSE, $l.key = k$. Thus, $k \in L(C') = L(C)$. This ensures claim 3 is preserved in C'.

By Lemma 40 applied to C', there is no imminent DELETE(k) in C. So, $k \in Q(C)$. By the induction hypothesis, $k \in S(C)$. So, a DELETE(k) performed on a set in state $S(C)$ would return TRUE. By Lemma 37 the DELETEk linearized at s also returns TRUE, establishing claim 4.

- Suppose s is the first child CAS of an INSERT(k) operation. This is not the linearization point of any operation, so $S(C') = S(C)$. Furthermore, claim 4 follows from the induction hypothesis. By Lemma 29, $L(C') = L(C) \cup \{k\}$. By definition of imminent and Lemma 40, $I_{ins}(C') = I_{ins}(C) - \{k\}$. Furthermore $I_{del}(C') = I_{del}(C)$. So, $Q(C') = Q(C)$ and $S(C') = S(C)$, so claim 1, 2 and 3 are preserved in C'.

- Suppose s is the first child CAS of a DELETE(k) operation. This is not the linearization point of any operation, so $S(C') = S(C)$. Furthermore, claim 4 follows from the induction hypothesis. By Lemma 29, $L(C') = L(C) - \{k\}$. By definition of imminent and Lemma 40, $I_{del}(C') = I_{del}(C) - \{k\}$. Furthermore $I_{ins}(C') = I_{ins}(C)$. So, $Q(C') = Q(C)$ and $S(C') = S(C)$, so claim 1, 2 and 3 are preserved in C'.

- Suppose s is the linearization point of a FIND(k) that returns TRUE or a INSERT(k) that returns FALSE. This linearization point is at the read of $p.update$ on line 66 of the final VALIDATELEAF of the operation, which returns TRUE. By Lemma 41, gp and p are the grandparent and parent of l, and neither are frozen. This means that l is in $T_\infty(C)$ and hence $k \in L(C)$. Moreover, there is no imminent DELETE(k) (since then gp would be frozen) so $k \notin I_{del}(C)$. Hence, $k \in Q(C)$ and $k \in S(C)$ by the induction hypothesis. So, $S(C') = S(C)$, since a FIND does not affect the abstract set and an INSERT(k) would have no effect. Also, $Q(C') = Q(C)$, so $S(C') = Q(C')$. Moreover, a FIND(k) done atomically on the set $S(C)$ would return TRUE and a INSERT(k) done atomically on the set $S(C)$ would return FALSE.

- Suppose s is the linearization point of a FIND(k) that returns FALSE or a DELETE(k) that returns FALSE. This linearization point is at the read of $p.update$ on line 66 of the final VALIDATELEAF of the operation, which returns TRUE. By Lemma 41, gp and p are the grandparent and parent of l, and neither are frozen. This means that l is in $T_\infty(C)$ and hence $k \notin L(C)$, since T_∞ is a BST by Lemma 36 and l is on the search path for k in $T_\infty(C)$. Moreover, there is no imminent INSERT(k) (since then p would be frozen) so $k \notin I_{ins}(C)$. Hence, $k \notin Q(C)$ and $k \notin S(C)$ by the induction hypothesis. So, $S(C') = S(C)$, since a FIND does not affect the abstract set and an DELETE(k) would have no effect. Also, $Q(C') = Q(C)$, so $S(C') = Q(C')$. Moreover, a FIND(k) or DELETE(k) done atomically on the set $S(C)$ would return FALSE.

■

Let G be the directed graph consisting of all nodes, where there is an edge from node u to node v if v was a child of u at some time during the execution.

Lemma 43 *G is acyclic.*

Proof: Lemma 29 implies that a child CAS does not set up a new path between two nodes that were active before the child CAS unless there was already a path between them. So, each child CAS preserves the truth of the lemma. ■

The following Lemma states that any call to ScanHelper (that satisfies certain preconditions) will output the right set of keys. It will be used to prove that RangeScan's output is correct.

Lemma 44 *Let seq be an integer. Suppose a completed invocation S to* ScanHelper*$(node, seq, a, b)$ satisfies the following preconditions in the configuration C before it is invoked.*

- *node is in $T_{seq}(C)$,*
- *no proper ancestor of node in $T_{seq}(C)$ is frozen in C for a successful Info object with sequence number that is at most seq,*
- *node is not permanently marked in C for an Info object whose sequence number is at most seq, and*
- *Counter $>$ seq in C.*

Let C' be the configuration before Counter is incremented from seq to seq $+ 1$. Then a key k is in the set returned by S iff

1. *$k \in [a, b]$,*
2. *node is on the search path for k in $T_{seq}(C)$,*
3. *either k appears in some leaf of the subtree of $T_{seq}(C')$ rooted at node or there is a successful* Insert*(k) with sequence number less than or equal to seq whose child CAS occurs after C', and*
4. *there is no successful* Delete*(k) with sequence number less than or equal to seq whose child CAS occurs after C'.*

Proof: Consider the subgraph G_{seq} of G consisting of nodes whose sequence numbers are less than *seq*. G_{seq} is finite since *Counter* $>$ *seq* at all times after C_{seq}, so only finitely many updates have sequence number at most *seq*. By Lemma 43, G_{seq} is acyclic. So, we prove the claim by induction on the length of the longest path from *node* to a sink in G_{seq}.

Base Case: If *node* is a sink in G_{seq}, then it is a leaf node.

($\Rightarrow$): Suppose S returns $\{k\}$. By line 137, k is the key of *node* and $k \in [a, b]$. So claim 1 is satisfied. Since *node* is in $T_{seq}(C)$ and $T_{seq}(C)$ is a BST by Lemma 36, *node* is on the search path for k in $T_{seq}(C)$, so claim 2 is satisfied.

Case 1: If $T_{seq}(C')$ contains a leaf with key k, claim 3 is satisfied. If there is a Delete(k) with sequence number at most *seq* that is imminent in C', then the child CAS must be completed before C since no proper ancestor of *node* is frozen in C for the Delete; but k cannot be re-inserted into T_{seq} after C', due to Lemma 40 applied to configuration C', contradicting the assumption that *node* is in $T_{seq}(C)$ and contains k. Thus, claim 4 is satisfied.

Case 2: If $T_{seq}(C')$ does not contain a leaf with key k, (since C is after C') there must have been an Insert(k) that added a leaf with key k to T_{seq}. That insertion must have sequence number at most *seq* (since otherwise it would not change T_{seq}, by Lemma 30). Thus, claim 3 is satisfied. Moreover, by Lemma 40, there cannot be a Delete(k) with sequence number at most *seq* whose child CAS occurs after C'. Thus, claim 4 holds.

($\Leftarrow$): Assume statments 1 to 4 are true for some key k. We must show that k is the key of *node* (and hence is returned by S, since $k \in [a, b]$ by statement 1). We argue that k is in a leaf of $T_{seq}(C)$. By statement 3, we can consider two cases.

Case 1: If k is in a leaf of the subtree of $T_{seq}(C')$ rooted at *node*, then k is the key of *node* since *node* is a leaf. By statement 4, k is a leaf of $T_{seq}(C)$.

Case 2: If there is a successful Insert(k) with sequence number at most *seq* whose child CAS occurs after C'. Then, its child CAS must occur before C (because no ancestor of *node* in $T_{seq}(C)$ is frozen for the Insert). So, by statement 4, k is in a leaf of $T_{seq}(C)$.

In either case, $T_{seq}(C)$ contains a leaf with key k. Since *node* is a leaf on the search path for k of $T_{seq}(C)$, and $T_{seq}(C)$ is a BST by Lemma 36, *node* must contain k.

Induction Step: Now suppose *node* is an Internal node. Assume the claim is true for calls to ScanHelper on nodes that are successors of *node* in G_{seq}. We prove that it is true for a call on *node*.

First, we argue that the recursive calls to ScanHelper satisfy the conditions of the lemma, so that we can apply the induction hypothesis to them. Let S_1 be a recursive call to ScanHelper inside S at line 141 to 144. Let C_1 be the configuration before S_1 is invoked. Let $node_1$ be the node argument of S_1. By hypothesis, none of *node*'s proper ancestors in $T_{seq}(C)$ are frozen with an Info object whose sequence number is less than or equal to *seq* in C. By handshaking, no update with sequence number at most *seq* can freeze its first node after C and succeed. So by Lemma 30, the path in T_{seq} from the root to *node* never changes after C. At some time during line 140, *node* is not frozen for an in-progress Info object, by Lemma 12. So *node*'s version-*seq* children do not change after this, and at configuration C_1 *node* is in $T_{seq}(C_1)$ and $node_1$ is *node*'s version-*seq* child, so $node_1$ is also in $T_{seq}(C_1)$, as required.

Each proper ancestor of *node* was not frozen in C for a successful Info object with sequence number at most *seq*. If any of those ancestors became frozen after C with an Info object with sequence number at most *seq*, then that Info object is doomed to abort due to handshaking. Line 140 ensures *node* is not temporarily frozen (i.e., for an in-progress Info object) with sequence number at most *seq*, and handshaking ensures that it will never become so afterwards. Since none of *node*'s ancestors is temporarily flagged in C (with a sequence number at most *seq*) and *node* is not permanently marked in C, it follows that *node* never gets permanently marked after C by an Info object with sequence number at most *seq*.

Similarly, because none of $node_1$'s ancestors is flagged at C_1 by an Info object with sequence number at most *seq*, $node_1$ cannot be permanently marked by an Info object with sequence number at most *seq* at C_1.

This completes the proof that the conditions of the Lemma are met for the recursive calls to ScanHelper, so we can apply the induction hypothesis to them.

($\Rightarrow$): Suppose k is returned by S. We must prove that the 4 numbered claims are true for k. The key k is returned by one of the recursive calls S' on line 141–144. Since S' returns k, $k \in [a, b]$ by the induction hypothesis, so claim 1 is satisfied. By the induction hypothesis, the version-*seq* child of *node* upon which S' is called is on the search path for k in T_{seq} so *nodeistoo*. Similarly, claims 3 and 4 follow from the fact that they are satisfied for the recursive call S'.

($\Leftarrow$): Now suppose k is some key that satisfies claims 1 to 4. If $k < node.key$, the four claims are satisfied for the version-*seq* left child of *node*, and there is a recursive call on that child in line 142 or 144, since $a \leq k < node.key$. If $k \geq node.key$, the four claims are satisfied for the version-*seq* right child of *node*, and there is a recursive call on that child in line 141 or 143, since $b \geq k \geq node.key$. Thus, one of the recursive calls returns k, and so does S. ■

Theorem 45 *The implementation is linearizable.*

Proof: It follows from Lemma 42 and 44 that each terminated operation returns the same value that it would if operations were performed atomically in the linearization ordering. ■

5.2.6 Progress

The remaining results show that RangeScans are wait-free and all other operations are non-blocking.

Lemma 46 *Calls to* ReadChild *are wait-free.*

Proof: Whenever a node is created, its *prev* pointer is set to a node that already exists. Thus, there can be no cycles among *prev* pointers. ■

Theorem 47 RangeScan*s are wait-free.*

Proof: Let $\ell \geq 0$. We prove that no call to ScanHelper with parameter $seq = \ell$ can take infinitely many steps. Let G_ℓ be the subgraph of G consisting of nodes whose *seq* field is equal to ℓ. Note that G_ℓ is acyclic since G is acyclic and finite, since the RangeScan increments *Counter* from ℓ to $\ell + 1$ and only nodes created by iterations of the while loops of update operations that read *Counter* before this increment can belong to G_ℓ.

We prove the claim by induction on the maximum length of any path from *node* to a sink of G_ℓ:

Base case: if *node* is a sink of G_ℓ, then it must be a leaf, so termination is immediate.

Inductive step: ScanHelper$(node, \ell, a, b)$ calls ScanHelper on nodes that are successors of *node* in G_ℓ, which terminates by the induction hypothesis, and ReadChild, which terminates by Lemma 46.

Then the claim follows, since RangeScan just calls ScanHelper. ■

Theorem 48 *The implementation is non-blocking.*

Proof: To derive a contradiction, suppose there is an infinite execution where only a finite number of operations terminate. Eventually, no more RangeScan operations take steps, by Lemma 47, so the *Counter* variable stops changing. Let ℓ be the final value of *Counter*. Since there is at most one successful child CAS belonging to each update operation, there is a point in the execution after which there are no more changes to child pointers.

Suppose there is at least one update that takes infinitely many steps. Let O be the set of update operations that each take infinitely many steps without terminating. Beyond some point, each Search performed by an operation in O repeatedly returns the same three nodes gp, p and l. If gp or p is frozen, the operation calls Help on the Info object causing that Info object's state to become Abort or Commit, by Lemma 37. So, eventually these three nodes can be frozen for updates in O. Consider a node v in G that is the p node of some Insert in O or the gp node of some Delete in O such that no other such node is reachable from v. (Such a v exists, since G is acyclic and finite.) One of the operations in O will eventually sucessfully perform its first freeze CAS on v, and then no other operation can prevent it from freezing the rest of its nodes, so the operation will terminate, a contradiction.

Now suppose there is no update that takes infinitely many steps. So, the operations that run forever are all Find operations. Let O be the set of these operations. Beyond some point, each Search performed by a Find in O will repeatedly return the same gp, p and l. Due to helping, these nodes will eventually be unfrozen, so the ValidateLeaf called by Find will return True and the Find will terminate, which is again a contradiction. ■

6 Open Questions

We believe that our approach can be generalized to work on many other concurrent data structures. Could it be used, for example, to provide RangeScans for Natarajan and Mittal's implementation of a non-blocking leaf-oriented BST [29], which records information about ongoing operations in the tree edges they modify? Or with Natarajan *et al.*'s wait-free implementation of a red-black tree [30], which is based on the framework of [40]? More generally, could we design a general technique similar to [7, 8] to support wait-free partial Scans on top of any concurrent tree data structure?

References

[1] A. Agarwal, Z. Liu, E. Rosenthal, and V. Saraph. Linearizable iterators for concurrent data structures. *CoRR*, abs/1705.08885, 2017.

[2] H. Attiya, R. Guerraoui, and E. Ruppert. Partial snapshot objects. In *Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 336–343, 2008.

[3] H. Avni, N. Shavit, and A. Suissa. Leaplist: Lessons learned in designing TM-supported range queries. In *Proc. 2013 ACM Symposium on Principles of Distributed Computing*, pages 299–308, 2013.

[4] A. Braginsky and E. Petrank. A lock-free B+tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 58–67, 2012.

[5] 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.

[6] T. Brown and H. Avni. Range queries in non-blocking k-ary search trees. In *Proc. 16th International Conference on Principles of Distributed Systems*, volume 7702 of *LNCS*, pages 31–45, 2012.

[7] T. Brown, F. Ellen, and E. Ruppert. Pragmatic primitives for non-blocking data structures. In *Proc. 32nd ACM Symposium on Principles of Distributed Computing*, pages 13–22, 2013.

[8] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *Proc. 19th ACM Symposium on Principles and Practice of Parallel Programming*, pages 329–342, 2014.

[9] B. Chatterjee. Lock-free linearizable 1-dimensional range queries. In *Proc. 18th International Conference on Distributed Computing and Networking*, pages 9:1–9:10, 2017.

[10] B. Chatterjee, N. Nguyen, and P. Tsigas. Efficient lock-free binary search trees. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing*, pages 322–331, 2014.

[11] L. Chen, Y. Gao, A. Zhong, C. S. Jensen, G. Chen, and B. Zheng. Indexing metric uncertain data for range queries and range joins. *VLDB Journal*, 26(4):585–610, 2017.

[12] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert. The amortized complexity of non-blocking binary search trees. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing*, pages 332–340, 2014.

[13] 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.

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

[15] P. Fatourou, Y. Nikolakopoulos, and M. Papatriantafilou. Linearizable wait-free iteration operations in shared double-ended queues. *Parallel Processing Letters*, 27(2):1–17, 2017.

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

[17] M. He and M. Li. Deletion without rebalancing in non-blocking binary search trees. In *Proc. 20th International Conference on Principles of Distributed Systems*, pages 34:1–34:17, 2016.

[18] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[19] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. 23rd International Conference on Distributed Computing Systems*, pages 522–529. IEEE, 2003.

[20] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, July 1990.

[21] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 161–171, 2012.

[22] Intel Threading Building Blocks documentation. https://www.threadingbuildingblocks.org/docs/help/reference/containers_overview.

[23] Java Platform Standard Edition 7 documentation. http://docs.oracle.com/javase/7/docs/index.html.

[24] P. Jayanti. An optimal multi-writer snapshot algorithm. In *Proc. 37th ACM Symposium on Theory of Computing*, pages 723–732, 2005.

[25] N. D. Kallimanis and E. Kanellou. Wait-free concurrent graph objects with dynamic traversals. In *Proc. 19th International Conference on Principles of Distributed Systems*, Leibniz International Proceedings in Informatics, 2015.

[26] E. M. Knorr, R. T. Ng, and V. Tucakov. Distance-based outliers: Algorithms and applications. *VLDB Journal*, 8(3-4):237–253, 2000.

[27] M. M. Michael. CAS-based lock-free algorithm for shared deques. In *Proc. 9th International Euro-Par Conference on Parallel Processing*, number 2790 in LNCS, pages 651–660. Springer, 2003.

[28] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[29] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *Proc. 19th ACM Symposium on Principles and Practice of Parallel Programming*, pages 317–328, 2014.

[30] A. Natarajan, L. Savoie, and N. Mittal. Concurrent wait-free red black trees. In *Proc. 15th International Symposium on Stabilization, Safety and Security of Distributed Systems*, volume 8255 of *LNCS*, pages 45–60, 2013.

[31] .NET framework class library documentation. http://msdn.microsoft.com/en-us/library/gg145045.aspx.

[32] Y. Nikolakopoulos, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A consistency framework for iteration operations in concurrent data structures. In *Proc. IEEE International Parallel and Distributed Processing Symposium*, pages 239–248, 2015.

[33] Y. Nikolakopoulos, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Of concurrent data structures and iterations. In *Algorithms, Probability, Networks and Games: Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occassion of his 60th Birthday*, volume 9295 of *LNCS*, pages 358–369. Springer, 2015.

[34] T. Palpanas. Data series management: The road to big sequence analytics. *SIGMOD Record*, 44(2):47–52, 2015.

[35] E. Petrank and S. Timnat. Lock-free data-structure iterators. In *Proc. 27th International Symposium on Distributed Computing*, number 8205 in LNCS, pages 224–238. Springer, 2013.

[36] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proc. 17th ACM Symposium on Principles and Practice of Parallel Programming*, pages 151–160, 2012.

[37] E. Rosenthal. Linearizable iterators. Manuscript available from https://cs.brown.edu/research/pubs/theses/masters/2016/rosenthal.eli.pdf.

[38] N. Shafiei. Non-blocking Patricia tries with replace operations. In *Proc. 33rd International Conference on Distributed Computing Systems*, pages 216–225, 2013.

[39] A. Spiegelman and I. Keidar. Dynamic atomic snapshots. In *Proc. 20th International Conference on Principles of Distributed Systems*, Leibniz International Proceedings in Informatics, 2016.

[40] J.-J. Tsay and H.-C. Li. Lock-free concurrent tree structures for multiprocessor systems. In *Proc. International Conference on Parallel and Distributed Systems*, pages 544–549, 1994.

[41] 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.