Lock-Free Augmented Trees

Panagiota Fatourou
FORTH ICS and University of Crete, Greece
faturu@ics.forth.gr

Eric Ruppert
York University, Canada
ruppert@eecs.yorku.ca

FORTH ICS TR 495 August 2025

Abstract

Augmenting an existing sequential data structure with extra information to support greater functionality is a widely used technique. For example, search trees are augmented to build sequential data structures like order-statistic trees, interval trees, tango trees, link/cut trees and many others.

We study how to design *concurrent* augmented tree data structures. We present a new, general technique that can augment a lock-free tree to add any new fields to each tree node, provided the new fields' values can be computed from information in the node and its children. This enables the design of lock-free, linearizable analogues of a wide variety of classical augmented data structures. As a first example, we give a wait-free trie that stores a set S of elements drawn from $\{1, \ldots, N\}$ and supports linearizable order-statistic queries such as finding the kth smallest element of S. Updates and queries take $O(\log N)$ steps. We also apply our technique to a lock-free binary search tree (BST), where changes to the structure of the tree make the linearization argument more challenging. Our augmented BST supports order statistic queries in $O(h)$ steps on a tree of height h. The augmentation does not affect the asymptotic running time of the updates.

For both our trie and BST, we give an alternative augmentation to improve searches and order-statistic queries to run in $O(\log |S|)$ steps (with a small increase in step complexity of updates). As an added bonus, our technique supports arbitrary multi-point queries (such as range queries) with the same time complexity as they would have in the corresponding sequential data structure.

1 Introduction

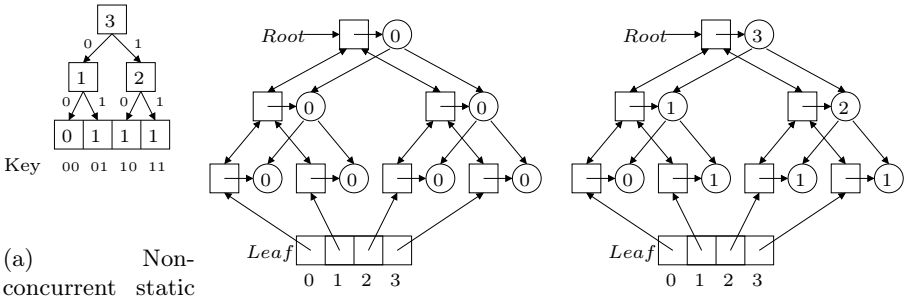
Augmentation is a fundamental technique to enhance the functionality of sequential data structures and to make them more efficient, particularly for queries.

Augmentation is sufficiently important to warrant a chapter in the algorithms textbook of Cormen et al. [16], which illustrates the technique with the most well-known example of augmenting a binary search tree (BST) to support many additional operations. By *augmenting* each node to store the size of the subtree rooted at that node, the BST can answer order-statistic queries, such as finding the j-th smallest element in the BST or the rank of a given element, *in sublinear time*. In a balanced BST, these queries take logarithmic time whereas a traversal of an unaugmented BST would take linear time to answer them.

More generally, each node of a (sequential) tree data structure can be augmented with any number of additional fields that are useful for various applications, provided that the new fields of a node can be computed using information in that node and its children. When applied to many standard trees, such as balanced or unbalanced BSTs, tries or B-trees, the augmentation does not affect the asymptotic time for simple updates, like insertions or deletions, but it can facilitate many other efficient operations. For example, a balanced BST of keys can be augmented for a RangeSum query that computes the sum of all keys within a given range in logarithmic time by adding a field to each node that stores the sum of keys in the node's subtree. (The sum can be replaced by any associative aggregation operator, such as minimum, maximum or product.) Similarly, a BST of key-value pairs can be augmented to aggregate the *values* associated with keys in a given range: each node should store the sum of values in its subtree. One can also filter values, for example to obtain the aggregate of all odd values within a range. More sophisticated augmentations can also be used. For instance, an interval tree stores a set of intervals in a balanced BST sorted by the left endpoints, where each node is augmented to store the maximum right endpoint of any interval in the node's subtree, so that one can determine whether any interval in the BST includes a given point in logarithmic time [16]. There are many other types of augmented trees, including one representing piecewise constant functions [10; 11, Section 4.5], measure trees [23], priority search trees [28] and segment trees [7, 8]. Section 3.5 gives another novel example of how to use tree augmentation. Augmented trees are also used as a building block for many other sequential data structures such as link/cut trees [35] and tango trees [17]. These structures have many applications in graph algorithms, computational geometry and databases.

We consider how to augment *concurrent* tree data structures. The resulting data structures are linearizable and lock-free and use single-word compare-and-swap (CAS) instructions. The technique we introduce is very general: as in the sequential setting, it can handle any augmentation to a lock-free tree data structure where the new fields can be computed using the data stored in the node and its children. Thus, it can be used to provide efficient, lock-free shared implementations of many of the sequential data structures mentioned above. Our augmentation does not affect the asymptotic running time of update operations. Moreover, we provide a way for queries to obtain a snapshot of the data structure so that they can simply execute the sequential code to answer the query.

For ease of presentation, we first illustrate the technique on top of a simple



(a) Non-concurrent static trie for the set $S = \{1, 2, 3\}$.

(b) Initialization of concurrent trie.

(c) Concurrent trie for set $S = \{1, 2, 3\}$.

Figure 1: Examples of the trie data structure when $U = \{0, 1, 2, 3\}$. Nodes are shown as squares, Versions are shown as circles containing their *sum* fields.

data structure that represents a dynamically-changing set S of keys drawn from the universe $U = \{0, \ldots, N-1\}$. The basic data structure is a *static* binary trie of height $\log_2 N$, where each key of U is assigned a leaf. Our technique mirrors this tree of nodes by a tree of Version objects, which store the mutable fields of the augmentation. See Figure 1c for an example where each node stores the number of elements present in its subtree to support order-statistic queries. Insertions and deletions of elements modify the appropriate leaf of the tree (and its Version), and then cooperatively propagate any changes to the Version objects stored in ancestors of that leaf until reaching the root. This cooperative approach ensures updates perform a constant number of steps at each node along this path, so they take $O(\log N)$ steps in total. Moreover, we design the updates so that the fields of Version objects (including their child pointers) never change. Thus, reading the root node's Version object provides a "snapshot" of the entire Version tree, which a query can then explore at its leisure, knowing that it will not be changed by any concurrent update operations. This allows any query operation that follows pointers from the root to be performed exactly as in a sequential version of the data structure, using the same number of steps. For example, order-statistic queries can be answered using $O(\log N)$ steps, and the size of S can be found in $O(1)$ steps. All operations are wait-free.

Then, we describe how to apply the technique to a BST. This encompasses additional complications because the structure of the tree changes dynamically as keys are inserted or deleted. We base our lock-free augmented BST on the lock-free BST of Ellen et al. [19], whose amortized step complexity is $O(h + c)$ per operation, where h is the height of the tree and c is the point contention (i.e., the maximum number of update operations running concurrently). Our augmentation does not affect this asymptotic step complexity of the lock-free update operations, and wait-free queries can again be performed using the same number of steps as in a sequential implementation.

In an augmented tree, each insertion or deletion must typically modify many tree nodes. For example, an insertion in an order-statistic tree must incre-

ment the count in all ancestors of the inserted node. In the concurrent setting, we must ensure that all of these changes appear to take place atomically, so that queries operate correctly. It is generally very difficult to design lock-free data structures where many modifications must appear atomic. Our proposed technique addresses this challenge in a rather simple way. However, the full proofs of correctness are fairly challenging.

Whether augmentation of the tree is needed or not, our technique also provides a simple way of taking a snapshot of the tree to answer queries that must examine multiple locations in the tree, such as a range query. Thus, in addition to supporting augmentation, our technique provides an alternative approach to other recent work on providing linearizable range queries on concurrent trees [3, 5, 12, 15, 21] or more general snapshots [31, 32, 38]. Ordinarily, our snapshots can be discarded when the query completes, but if desired, they can also be used to maintain past versions of the data structure. Many of these other approaches use multiversioning in a way that requires complex schemes for unlinking old, obsolete versions from the data structure to facilitate garbage collection (e.g., [6, 39]). The simplicity of our approach avoids this.

Similarly, our approach yields a constant-time algorithm for finding the number of keys in a lock-free tree. In contrast, a previous, more general method for adding a size query to any dynamic set [33] is substantially more complicated, and size queries take $\Omega(P)$ steps in a system of P processes.

2 Related Work

There is very little previous work on lock-free implementations of augmented trees. In a recent manuscript, Kokorin, Alistarh and Aksenov [27] describe a wait-free BST supporting order-statistic queries and range queries. They use a FIFO queue for each tree node. Before reading or writing a node, an operation must join the node's queue and help each operation ahead of it in the queue by performing that operation's access to the node and, if necessary, adding the operation to the queue of the node's child (or children). This adds $\Omega(Ph)$ to the worst-case step complexity per operation when there are P processes accessing a tree of height h. To handle order-statistic queries, each node stores the size of its subtree. They assume that all updates will succeed (e.g., a $\mathsf{Delete}(k)$ will always find k to be in the BST), so that an update can modify the *size* field of nodes on its way down a path in the tree. This top-down approach does not seem to generalize to other augmentations where new fields are generally computed bottom-up because the values of the fields of a node usually depend on the values in the node's children.

Sun, Ferizovic and Blelloch [36] discuss augmented trees in a parallel setting, but their focus is on processes sharing the work of a single expensive operation (like a large range query or unioning two trees), whereas our goal is to support multiple concurrent operations on the data structure.

The cooperative approach we use to propagate operations up to the root of the tree originates in the universal construction of Afek, Dauber and Touitou [1].

It has been used to build a variety of lock-free data structures [4, 20, 25, 29]. All of these applied the technique to a tournament tree with one leaf per process. A process adds an operation at its leaf, and processes move up the tree gathering larger batches of operations until the batch is applied to the data structure at the root of the tournament tree. Here, we instead apply the approach directly to the tree data structure itself to build larger and larger pieces of the updated tree until we reach the root, at which time we have constructed a new version of the data structure (without destroying any previous versions).

Jayanti [24] used the technique of [1] to implement an array $A[1..n]$ where processes can update an array element and query the value of some fixed function $f(A[1], \ldots, A[n])$, if f can be represented as an evaluation tree similar to a circuit (where leaves are elements of the array, each internal node represents some function of its children and the root represents f). Updates cooperatively propagate changes up the tree so that a query can read f's value from the root. Our trie has some similarities, but is much more general: instead of simply computing a function value, we construct a copy of the data structure that can be used for more complex queries. Our BST implementation goes further to remove the restriction that the shape of the tree being used for the propagation is fixed.

A different technique that also involves cooperatively building trees bottom-up also appears in Chandra, Jayanti and Tan's construction [14] of closed objects (where the effect of any pair of operations is equivalent to another operation). They build trees that represent batches of operations to keep track of the sequence of all operations applied to the closed object being implemented. In contrast, we directly build a representation of the implemented tree data structure.

Our work is on augmenting tree data structures with additional fields to support additional functionality. The main challenge is to make changes to several nodes required by an insert or delete appear atomic. As a byproduct, our technique for doing this also allows processes to take a snapshot of the tree, which can be used to answer arbitrary queries on the state of the tree. For example, it can be used on a BST to find all keys in a given range. A number of recent papers [9, 21, 31, 32, 38] use some form of multiversioning to add the ability to take a snapshot of the state of a concurrent data structure (but without addressing the problem of augmentation). Our approach applies only to trees, whereas some of the other work can be applied to arbitrary data structures, but we do get more efficient queries: a query in our scheme has the same step complexity as the corresponding query in a sequential implementation, whereas a query that runs on top of other multi-versioning schemes, such as that of [38], can take additional steps for every update to the tree that is concurrent with the query. Our approach is more akin to that of functional updates to the data structure that leave old versions accessible, as in the work on classical persistent data structures [18], but the novelty here is that the new versions are built cooperatively by many concurrent operations.

3 Augmented Static Trie

In this section, we illustrate our augmentation technique for the simple data structure that represents a set S of keys drawn from the universe $U = \{0, 1, \ldots, N-1\}$. For simplicity, assume N is a power of 2. A simple, classical data structure for S is a bit vector $B[0..N-1]$, where $B[i] = 1$ if and only if $i \in S$. Even in a concurrent setting, update operations (insertions and deletions of keys) can be accomplished by a single CAS instruction and searches for a key by a single read instruction.

To illustrate the technique, suppose we wish to support the following order-statistic queries.

- Select(k) returns the kth smallest element in S.
- Rank(x) returns the number of elements in S smaller than or equal to x.
- Predecessor(x) returns the largest element in S that is smaller than x.
- Successor(x) returns the smallest element in S that is larger than x.
- Minimum and Maximum return the smallest and largest element in S.
- RangeCount(x_1, x_2) returns the number of elements in S between x_1 and x_2.
- Size returns $|S|$, the number of elements in the set S.

In the *non-concurrent* setting we can build a binary tree of height $\log_2 N$ whose leaves correspond to the elements of the bit vector, as shown in Figure 1a. We augment each node x with a *sum* field to store the sum of the bits in x's descendant leaves, i.e., the number of elements of S in the subtree rooted at x. For a leaf, the *sum* field is simply the bit that indicates if that leaf's key is present in S. The *sum* field of an internal node can be computed as the sum of its children's *sum* fields. It is straightforward to see that Size queries can then be answered in $O(1)$ time and the other order-statistic queries can be answered in $O(\log N)$ time. We call this data structure a *static trie* because the path to the leaf for $i \in U$ is dictated by the bits of the binary representation of i, as in a binary trie [22; 26, Section 6.3]: starting from the root, go left when the next bit is 0, or right when the next bit is 1. Although the trie's *shape* is static, it represents a dynamically changing set S.

3.1 Wait-Free Implementation

The challenge of making the augmented trie concurrent is that each insertion or deletion, after setting the bit in the appropriate leaf, must update the *sum* fields of all ancestors of that leaf. All of these updates must appear to take place atomically. To achieve this, we use a modular design that separates the structure of the tree (which is immutable) from the mutable *sum* fields of the

nodes. This modularity means the same approach can be used to augment various kinds of lock-free trees.

We use Node objects to represent the tree structure. *Root* is a shared pointer to the root Node. To expedite access to the leaves, we use an array $Leaf[0..N-1]$, where $Leaf[i]$ stores a pointer to the leaf Node for key i. Each Node has a *version* field, which stores a pointer to a Version object that contains the current value of the Node's *sum* field. A Version object v associated with a node x also stores pointers *left* and *right* to the Version objects that were associated with x's children at the time when v was created. This way, the Version objects form a *Version tree* whose shape mirrors the tree of Nodes. See Figure 1b. Query operations are carried out entirely within this Version tree. To simplify queries, fields of Versions are immutable, so that when a query reads $Root.version$, it obtains a snapshot of the entire Version tree that it can later explore by following child pointers.

To see how updates work, consider an Insert(3) operation, starting from the initial state of the trie shown in Figure 1b. It must increment the *sum* field of the leaf for key i and of each Node along the path from that leaf to the root. Since Versions' fields are immutable, whenever we wish to change the data in the Version associated with a Node x, we create a *new* Version initialized with the new *sum* value for the Node, together with the pointers to the two Versions of x's children from which x's *sum* field was computed. Then, we use a CAS to attempt to swing the pointer in $x.version$ to the new Version. If the Insert(3) runs by itself, it would make the sequence of changes shown in Figure 4 as it works its way up the tree. The Insert is linearized when $Root.version$ is changed (Figure 4c). Prior to that linearization point, any query operation reading the root's *version* field gets a pointer to the root of the initial Version tree; after it, a query operation gets a pointer to a Version tree that reflects all the changes required by the Insert. A Delete(k) operation is handled similarly by decrementing the *sum* field at each Node along the path from k's leaf Node to the root.

Now, consider concurrent updates. Each update operation must ensure that the root's *version* pointer is updated to reflect the effect of the update. We avoid the performance bottleneck that this could create by having update operations *cooperatively* update Versions. At each Node x along the leaf-to-root path, the update reads the *version* field from both of x's children, creates a new Version for x based on the information in the children's Versions, and attempts to install a pointer to it in $x.version$ using a CAS. Following the terminology of [24], we call this procedure a *refresh*.This approach is cooperative, since a refresh of Node x by one update will propagate information from all updates that have reached either child of x to x. If an update's first refresh on x fails, it performs a second refresh. This is called a *double refresh* of x. We shall show that attempting a refresh twice at each Node suffices: if both of the CAS steps in an update's *double refresh* on a Node x fail, it is guaranteed that some other process has propagated the update's information to x.

Figure 2 describes the fields of our objects. Figure 3 provides pseudocode for the implementation. It is substantially simpler than previous lock-free tree data

```
1: type Node            ▷ used to store nodes of static trie structure
2:   | Node* left, right ▷ immutable pointers to children Nodes
3:   | Node* parent      ▷ immutable pointer to parent Node
4:   | Version* version  ▷ mutable pointer to current Version
5:
6: type Version         ▷ used to store a Node's augmented data
7:   | Version* left, right ▷ immutable pointers to children Versions
8:   | int sum          ▷ immutable sum of descendant leaves' bits
9:
```

Figure 2: Object types used in wait-free trie data structure.

structures for sets, even though it includes augmentation and provides atomic snapshots. In our code, if *ptr* is a pointer to an object O, $ptr.f$ denotes field f of O. A shared pointer *Root* points to the root Node of the binary tree with N leaves. We use an array $Leaf[0..N-1]$ where $Leaf[k]$ points to the leaf Node for key k.

A **Refresh**(x) reads the *version* field of x and its two children, creates a new Version for x based on information in the children's Versions, and then attempts to CAS the new Version into $x.version$. To handle different augmentations, one must only change the way Refresh computes the new fields. **Propagate**(x) performs a double Refresh at each node along the path from x to the root.

An **Insert**(k) first checks if the key k is already present in the set at line 15. If not, it uses a CAS at line 18 to change the leaf's Version object to a new Version object with *sum* field 1. If the CAS succeeds, the Insert will return true to indicate a successful insertion. If the key k is already present when the read at line 14 is performed or if the CAS fails (meaning that some concurrent operation has already inserted k), the Insert will return false. In all cases, the Insert calls Propagate before returning to ensure that the information in the leaf's Version is propagated all the way to the root.

The **Delete**(k) operation is very similar to an insertion, except that the operation attempts to switch the *sum* field of $Leaf[k]$ from 1 to 0.

Find and **Select** are given as examples of query operations in Figure 3. Each first takes a snapshot of the Version tree by reading $Root.version$ on line 41 or 47 and then executes the query's standard sequential code on that tree. Other queries can be done similarly. In particular, to ensure linearizability, queries should access the tree only via $Root.version$, not through the *Leaf* array.

3.2 Correctness

A detailed proof of correctness appears in Section A; we sketch it here, with references to claims that are formalized in the appendix. We first look at the structure of Version trees. Let U_x be the sequence of keys from the universe U that are represented in the subtree rooted at Node x of the tree, in the order they appear from left to right. In particular, $U_{Root} = \langle 0, 1, \ldots, N-1 \rangle$. We prove (in Invariant 1), by induction on the height of the Node x, that the Version

```
Initialization (refer to Figure 1b):
Node* Root ← root of a perfect binary tree of Nodes with N leaves.
For each Node x, x.version points to a new Version with fields sum ← 0, left ← x.left.version
                    and right ← x.right.version.
Node* Leaf[0..N − 1] contains pointers to the leaf Nodes of the binary tree.

function BOOLEAN(Insert)int k ▷ Add k to S; return true iff k was not already in S
    old ← Leaf[k].version
    result ← (old.sum = 0)
    if result then
        new ← new Version with sum ← 1, left ← Nil, and right ← Nil
        result ← CAS(Leaf[k].version, old, new)
    Propagate(Leaf[k].parent)
    return result

function BOOLEAN(Delete)int k ▷ Remove k from S; return true iff k was in S
    old ← Leaf[k].version
    result ← (old.sum = 1)
    if result then
        new ← new Version with sum ← 0, left ← Nil and right ← Nil
        result ← CAS(Leaf[k].version, old, new)
    Propagate(Leaf[k].parent)
    return result

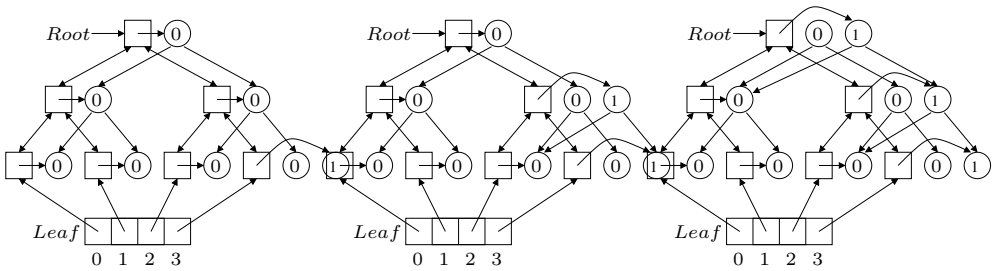
function BOOLEAN(Refresh)Node* x ▷ Try to propagate information to Node x from its children
    old ← x.version
    v_L ← x.left.version
    v_R ← x.right.version
    new ← new Version with left ← v_L, right ← v_R, sum ← v_L.sum + v_R.sum
    return CAS(x.version, old, new)

function (Propagate)Node* x ▷ Propagate updates from x's children up to root
    while x is not Nil do
        if not Refresh(x) then
            Refresh(x) ▷ Do a second Refresh if first one fails
        x ← x.parent

function BOOLEAN(Find)Key k ▷ Check if key k is in S
    v ← Root.version ▷ Start at the root
    for i ← 1.. log_2 N do ▷ Traverse path to leaf of Version tree
        if ith bit of binary representation of k is 0 then v ← v.left
        else v ← v.right
    return (v.sum = 1)

function INT(Select)j ▷ Return the jth smallest element in S
    v ← Root.version ▷ Start at the root
    i ← 1                ▷ Keep track of breadth-first index of v in tree
    if v.sum < j then return Nil▷ No such element in S
    else
        while v.left ≠ Nil do
            if v.left.sum ≥ j then ▷ Required element is in left subtree
                v ← v.left
                i ← 2i
            else       ▷ Required element is in right subtree
                v ← v.right
                i ← 2i + 1
                j ← j−v.left.sum ▷ Adjust rank of element being searched for
        return i−N ▷ Convert breadth-first index to value
```

Figure 3: Implementation of wait-free augmented trie.



(a) After the CAS on line 18 updates the *version* of the rightmost leaf Node.

(b) After the CAS on line 34 updates the right child of the root Node in the first iteration of the loop in Propagate.

(c) After the CAS on line 34 updates the root Node in the second iteration of the loop in Propagate.

Figure 4: Key steps of an Insert(3) into the initially empty set shown in Figure 1b.

tree rooted at $x.version$ is a perfect binary tree with $|U_x|$ leaves. Recall that the fields of Version objects are immutable, so we need only consider lines 17, 25 and 33, which create new Version objects. Invariant 1 can be easily proved because of the way the Version tree for x is constructed at line 33 by combining the Version trees for x's children. Line 33 also ensures that, for every internal Version v, $v.sum = v.left.sum + v.right.sum$ (Invariant 2). Since leaf Versions contain 0 or 1, according to lines 17 and 25, so $v.sum$ stores the sum of the bits stored in leaves in the subtree rooted at v.

The key goal of the correctness proof is to define linearization points for the update operations (insertions and deletions) so that, at all times, the Version tree rooted at *Root.version* accurately reflects all update operations linearized so far. Then, we linearize each query operation at the time it reads *Root.version* to take a snapshot of the Version tree. This will ensure that the result returned by the query is consistent with the state of the represented set S at the query's linearization point.

We consider an arbitrary execution in which processes perform operations on the trie. An execution is formalized as an alternating sequence of configurations and steps $C_0, s_1, C_1, s_2, \ldots$, where each configuration C_i describes the state of the shared memory and the local state of each process, and each s_i is a step by some process that takes the system from configuration C_{i-1} to C_i. A step is either a shared-memory access or a local step that affects only the process's local state.

Our goal is to define a linearization point (at a step of the execution) of each update operation so that for each configuration C, the Version tree rooted at *Root.version* is the trie that would result by sequentially performing all the operations that are linearized before C in their linearization order. Thus, the linearization point of an update operation should be the moment when the effect of the update has been propagated to the root Node, so that it becomes visible

to queries. To define these linearization points precisely, we define the *arrival point* of an update operation on a key k at each Node along the path from the leaf Node representing k up to the root Node. Intuitively, the arrival point of the update at Node x is the moment when the effect of the update is reflected in the Version tree rooted at $x.version$. Then, the linearization point is simply the arrival point of the update at *Root*. We must ensure these linearization points are well defined by showing that the double-refresh technique propagates each update all the way up to *Root* before the update terminates.

Definition 1, below, formally defines the arrival point of each Insert(k) or Delete(k) operation at Node x, where $k \in U_x$ using induction from the bottom of the tree to the top. If an Insert(k) sees that k is already in a leaf Node at line 14 , or if a Delete(k) sees that k is not present in a leaf Node at line 22, the arrival point of the operation is at that line. Otherwise the update performs a CAS on the leaf at line 18 or 26. If the CAS succeeds, the CAS is the update's arrival point at that leaf. Otherwise, we put the arrival point of the update at the leaf at a time when k's presence or absence would cause the update to fail. An update's arrival point at an internal Node is the first successful CAS by a Refresh that previously read the child after the update's arrival point at that child.

Definition 1. We first define the arrival point of an Insert(k) or Delete(k) operation *op* at $Leaf[k]$.

1. If *op* performs a successful CAS at line 18 or 26, then the arrival point of *op* is that CAS.

2. If *op* performs an unsuccessful CAS at line 18 or 26, then the arrival point of *op* is the first successful CAS on $Leaf[k].version$ after *op* read the old value of $Leaf[k].version$ at line 14 or 22. (Such a CAS must exist; otherwise *op*'s CAS would have succeeded.)

3. If *op* is an Insert that reads a Version with $sum = 1$ from $Leaf[k].version$ on line 14 or *op* is a Delete that reads a Version with $sum = 0$ from $Leaf[k].version$ on line 22, then the arrival point of *op* is *op*'s read at line 14 or 22.

If multiple operations' arrival points at a leaf Node are at the same successful CAS, we order them: first the operation that did the successful CAS, then all the other operations (ordered arbitrarily).

Next, we define the arrival point of an Insert(k) or Delete(k) *op* at an internal Node x with $k \in U_x$.

4. If $k \in U_{x.left}$, the arrival point of *op* is the first successful CAS on $x.version$ at line 34 of a Refresh that read $x.left.version$ at line 31 after the arrival point of *op* at $x.left$.

5. If $k \in U_{x.right}$, the arrival point of *op* is the first successful CAS on $x.version$ at line 34 of a Refresh that read $x.right.version$ at line 32 after the arrival point of *op* at $x.right$.

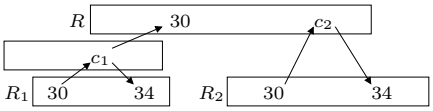


Figure 5: Call to Refresh in proof that a double refresh successfully propagates updates to a Node from its children. The horizontal axis represents time, and boxes indicate the interval between a routine's invocation and its response. Numbers refer to line numbers in the pseudocode. An arrow $s_1 \to s_2$ indicates step s_1 must precede step s_2.

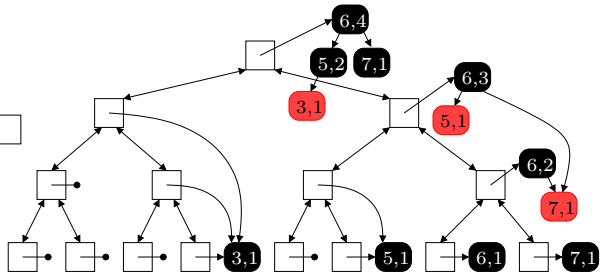


Figure 6: Augmenting the trie with red-black trees (RBTs) to speed up queries. $N = 8$ and $S = \{3, 5, 6, 7\}$. Squares are trie Nodes. Ovals are RBT nodes. Each RBT node has child pointers, and stores a key and a *size* field that represents the number of keys in the subtree. Black dots represent RBT nodes with *sum* 0.

If multiple operations' arrival points at an internal Node are at the same successful CAS, we order them as follows: first the operations on keys in $U_{x.left}$ in the order they arrived at $x.left$ and then the operations on keys in $U_{x.right}$ in the order they arrived at $x.right$.

For example, consider the Insert(3) depicted in Figure 4. Its arrival point at the leaf Node representing key 3 is the CAS that updates that leaf's *version* field, shown in Figure 4a. Its arrival point at the parent of this leaf is the CAS that updates the data structure as shown in Figure 4b. Its arrival point at the root is the CAS that updates the *Root.version* as shown in Figure 4c.

It follows easily from Definition 1 that arrival points of an update operation *op* are after *op* begins. (See Lemma 1.) If *op* terminates, we also show (in Lemma 2) that it has an arrival point at the root Node before it terminates. Recall that after *op*'s arrival point at a leaf, *op* calls Propagate, which does a double Refresh at each Node along the path from that leaf to the root. We show by induction that the double refresh at each node x along the path ensures *op* has an arrival point at x. The induction step follows immediately from Parts 4 and 5 of Definition 1 if one of *op*'s calls to Refresh(x) performs a successful CAS. So, suppose both of x's calls R_1 and R_2 to Refresh(x) fail their CAS. Then for each R_i, there must be a successful CAS c_i on $x.version$ between R_i's read of $x.version$ on line 30 and its CAS on line 34, as depicted in Figure 5. Although c_1 may store outdated information, the Refresh that performs c_2 must have read information from x's children after c_1, which is enough to ensure that *op* has an arrival point at x, by Parts 4 and 5 of Definition 1.

Our next goal is to prove a key invariant that, for each configuration C and Node x, the Version tree rooted at $x.version$ accurately reflects all of the updates whose arrival points at x are prior to C. In other words, it is a trie structure

(similar to the one shown in Figure 1a) that would result from performing all of those updates in the order of their arrival points at x. As a corollary, when we take x to be the root Node, we see that the Version tree rooted at $Root.version$ has a 1 in the leaf for key k if and only if k is in the set obtained by sequentially performing the linearized operations in order. Correctness of all query operations follows from this fact and the fact that *sum* fields in Version trees are accurate (Invariant 2, described above).

We sketch the proof of the key invariant. We make the argument separately for each key $k \in U_x$. We define $Ops(C, x, k)$ to be the sequence of update operations on key k whose arrival points at x precede configuration C, in the order of their arrival points. We must show that, in each configuration C, the leaf corresponding to key k in the subtree rooted at $x.version$ contains a 1 if and only if $Ops(C, x, k)$ ends with an Insert(k). (See Invariant 4.)

If x is the leaf for key k, we consider each step that can add arrival points at x. First, consider a CAS that flips the bit stored in $x.version$. If the CAS sets the bit to 1, it follows from Part 1 and Part 2 of Definition 1 that it is the arrival point of one or more Insert(k) operations, which preserves the invariant. Similarly, a CAS that sets the bit to 0 is the arrival point of one or more Delete(k) operations, which preserves the invariant. If the step is an Insert(k)'s read of $x.version$ when it has value 1 or a Delete(k)'s read of $x.version$ when it has value 0, it also preserves the invariant.

If x is an internal Node, the fact that the invariant holds at x can be proved inductively. The claim at x follows from the assumption that it holds at the children of x, since the invariant is phrased in terms of a single key and the sets of keys represented in the two subtrees of x are disjoint.

Finally, we prove that operations that arrive at a leaf are propagated up the tree in an orderly way, so that they arrive at the root in the same order. This is useful for showing that the update operations return results consistent with their linearization order (Lemma 4).

3.3 Complexity and Optimizations

Insert and Delete take $O(\log N)$ steps. Searches and the order-statistic queries listed at the beginning of Section 3 take $O(\log N)$ steps and are read-only. Size queries can be answered in $O(1)$ time by simply returning $Root.version.sum$. We could also augment the data structure so that each node stores the minimum element in its subtree to answer Minimum queries in $O(1)$ time. A range query that returns R elements can be done in $O(R(\log \frac{N}{R} + 1))$ time, since it visits at most R locations in the top $\log R$ levels of the Version tree and in the rest of the tree it visits $O(\log N - \log R)$ locations per returned element, for a total of $O(R(\log N - \log R + 1))$ locations. All operations are wait-free.

We assume a safe garbage collector, such as the one provided by Java, which deallocates objects only when they are no longer reachable. We now consider a very pessimistic worst-case bound on the amount of space used by objects that are still reachable. For each Node x, up to $O(\log N)$ different Versions belonging to x could be stored in the Version trees of each of x's ancestors.

Thus, the space used by all objects reachable by following pointers from *Root* is $O(N \log N)$. In addition to this, any old ongoing queries could have an old snapshot of a Version tree.

The Node tree is static and complete, so it can be represented using an array $Tree[1..2N-1]$ of pointers to Versions, where $Tree[1]$ is the root, and the children of the internal Node $Tree[i]$ are $Tree[2i]$ and $Tree[2i+1]$ [26, p. 144]. This saves the space needed for the *Leaf* array and parent and child pointers, since we can navigate the tree by index arithmetic rather than following pointers.

3.4 Variants and Other Applications

We described how to augment the trie with a *sum* field to facilitate efficient order-statistic queries. However, the method can be used for any augmentation where the values of a node's additional fields can be computed from information in the node and its children, by modifying line 33 to compute the new fields. Section 1 mentions some of the many applications where this can be applied.

Generalizing our implementation to d-ary trees is straightforward for any $d \geq 2$. The number of CAS instructions per update would be reduced to $2 \log_d N$, but the number of reads (and local work) per update would increase to $\Theta(d \log_d N)$. Order statistic queries could still be made to run in $\Theta(\log_2 N)$ time if each node stores prefix sums and uses binary search.

Instead of storing a set of keys $S \subseteq U$, a straightforward variant of our data structure can store a set of key-value pairs where each record has a unique key drawn from U. Instead of storing just one bit, a leaf's Version object would also store the associated value. A $\mathsf{Replace}(k, v)$ operation that replaces the value associated with key k with a new value v would update the appropriate leaf's $version$ field and call Propagate. If several $\mathsf{Replace}(k, *)$ operations try to update a leaf concurrently, one's CAS will succeed and the others will fail, and we can assign them all arrival points at the leaf at the time of the successful CAS, with the failed operations preceding the successful one.

Our approach can also provide lock-free *multisets* of keys drawn from U. Instead of storing a bit, the leaf for key k stores a Version whose *sum* field is the number of copies of k in the multiset. With CAS instructions, operations can be made lock-free if each $\mathsf{Insert}(k)$ or $\mathsf{Delete}(k)$ repeatedly tries to install a new Version k's leaf with its *sum* field incremented or decremented and then calls Propagate. If the leaf's *sum* field can be updated with a fetch&add, the updates can be made wait-free.

Without any modification, our trie supports multipoint queries, like range searches that return all keys in a given range, since reading $Root.version$ yields a snapshot of the trie. In fact, our technique has more efficient queries than some recent papers discussed in Section 2 that provide multipoint queries: in our approach, queries take the same number of steps as in a sequential implementation.

3.5 Improving Query Time to $O(\log n)$

The time to perform order-statistic queries on the set S can be improved from $O(\log N)$ to $O(\log |S|)$. To do this, we simply use a different augmentation. The *version* field of each Node x stores a pointer to the root of a red-black tree (RBT) that represents all the elements in the subtree of Nodes rooted at x. See Figure 6 for an example. A Refresh(x) updates $x.version$ by reading the RBTs stored in $x.left.version$ and $x.right.version$, joining them into one RBT (without destroying the smaller RBTs) and then using a CAS to store the root of the joined RBT in $x.version$. The algorithm to Join two RBTs in logarithmic time, provided that all elements in one are smaller than all elements in the other, is in Tarjan's textbook [37]. To avoid destroying the smaller RBTs when performing a Join, one can use the path-copying technique of Driscoll et al. [18]. Pseudocode is in Section B.

Each RBT node also has a *size* field storing the number of elements in the subtree rooted at that node. A query reads $Root.version$ to get a snapshot of a RBT containing all elements in the dynamic set. Order-statistic queries are answered in $O(\log n)$ steps using the *size* fields of the RBT.

There is a tradeoff: the time for updates increases to $O(\log N \log \hat{n})$, since a Join of two RBTs must be performed at each of $\log N$ Nodes of the Node tree during Propagate. Here, $\hat{n}$ denotes a bound on the maximum size the set could have under any possible linearization of the update operations. The elements in a RBT constructed by a Refresh on a non-root Node may never all be in the set simultaneously, so we must argue that the size of each such RBT is $O(\hat{n})$. Consider a Join(T_1, T_2) during a call R to Refresh(x). Without loss of generality, assume $|T_1| \geq |T_2|$. Let α' be the prefix of the execution up the time R reads T_1 from $x.left.version$. Suppose we modify α' by delaying R's read of $x.version$ until just before R reads $x.left.version$, and then appending to the execution all the steps needed to complete the Propagate that called R. This will ensure that all remaining CAS steps of the Propagate succeed and T_1 will be a subtree of the tree stored in $Root.version$. Thus, there must be some way to linearize α' so that all elements in T_1 are simultaneously in the represented set (since the modified execution is linearizable), so $|T_1| \leq \hat{n}$. Thus, the size of the RBT that R builds is $|T_1| + |T_2| \leq 2|T_1| \leq 2\hat{n}$.

4 Augmented Binary Search Tree

In this section, we illustrate our technique by augmenting a binary search tree (BST) that represents a set S of elements drawn from an *arbitrary* (ordered) universe U. We describe the augmentation for order-statistic queries, but as explained above, the same approach can be used for many other applications. In constrast to the augmented trie of Section 3, the time and space complexity of our augmented BST depend on $|S|$ rather than $|U|$.

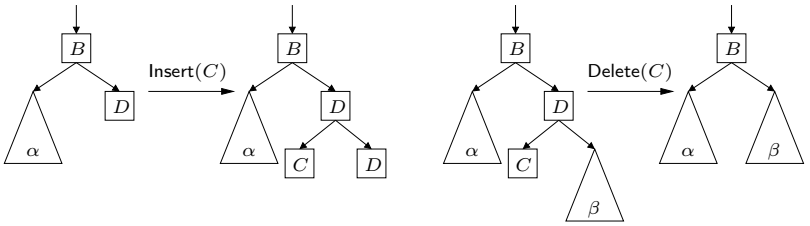


Figure 7: How updates modify a leaf-oriented BST. Here, α and β represent arbitrary subtrees.

4.1 Basic Lock-free BST

We base our augmented BST on the lock-free BST of Ellen et al. [19], so we first give a brief overview of how their BST works. The BST is leaf-oriented: keys of S are stored in the leaves; keys in internal nodes serve only to direct searches to the leaves. The BST property requires that all keys in the left subtree of a node x are smaller than x's key and all keys in the right subtree of x are greater than or equal to x's key. The tree nodes maintain child pointers, but not parent pointers. To simplify updates, the BST is initialized with three sentinel nodes: an internal node and two leaves containing dummy keys ∞_1 and ∞_2, which are considered greater than any actual key in U and are never deleted. A shared *Root* pointer points to the root node of the tree, which never changes.

An insert or delete operation starts at the root and searches for the leaf at which to apply its update. Updates are accomplished by simple modifications to the tree structure as shown in Figure 7. To coordinate concurrent updates to the same part of the tree, updates must flag a node before modifying one of its child pointers and remove the flag when the modification is done. Before removing an internal node from the tree, the operation must *permanently* flag it. Since only one operation can flag a node at a time, flagging a node is analogous to locking it. To ensure lock-free progress, an update that needs to flag a node that is already flagged for another update first *helps* the other update to complete and then tries again to perform its operation. When retrying, the update does not begin all over from the top of the tree; the update keeps track of the sequence of nodes it visited on a thread-local stack so that it can backtrack a few steps up the tree by popping the stack until reaching a node that is not permanently flagged for deletion, and then searches onward from there for the location to retry its update. Each update is linearized at the moment one of the changes shown in Figure 7 is made to the tree, either by the operation itself or by a helper.

The tree satisfies the BST property at all times. We define the *search path* for a key k at some configuration C to be the path that a sequential search for k would take if it were executed without interruption in C. Searches in the lock-free BST ignore flags and simply follow child pointers until reaching a leaf. A search for k may pass through nodes that get removed by concurrent updates,

but it was proved in [19] that each Node the search visits *was* on the search path for k (and by the way we linearize updates, it was thus also in the set represented by the BST) at some time during the search. A search that reaches a leaf ℓ is linearized when that leaf was on the search path for k.

4.2 Lock-free Augmentation

We now describe how to augment the lock-free BST of [19] with additional fields for each node, provided the fields can be computed from information in the node and its children. We again use the *sum* field, which supports efficient order-statistic queries, as an illustrative example. As in Section 3.1, we add to each tree Node x a new *version* field that stores a pointer to a tree of Version objects. This Version tree's leaves form a snapshot of the portion of S stored in the subtree rooted at x. In particular, the leaves of the Version tree stored in $Root.version$ form a snapshot of the entire set S. Each Version v stores a *sum* field and pointers to the Versions of x's children that were used to compute v's *sum*. Each Version associated with Node x also stores a copy of x's key to direct searches through the Version trees. Version trees will always satisfy the BST property, and the *sum* field of each Version v stores the number of keys in leaf descendants of v. See Figure 10 on page 34 for a formal description of the Node and Version object types. See Figure 8a for the initial state of the BST, including the sentinel Nodes. Pseudocode for the implementation is in Section C.

An **Insert** or **Delete** first runs the algorithm from [19] to modify the Node tree as shown in Figure 7. Figures 8b and 8c show the effects of the modification when Versions are also present. Then, the update calls Propagate to modify the *sum* fields of the Versions of all Nodes along the path from the location where the key was inserted or deleted to the root. As in Section 3.1, an update operation's changes to the *sum* field of all thes Nodes become visible at the same time, and we linearize the update at that time. If an Insert(k) reaches a leaf Node that already contains k, before returning false, it also calls Propagate to ensure that the operation that inserted the other copy of key k has been propagated to the *Root* (and therefore linearized). Similarly, a Delete(k) that reaches a leaf Node and finds that k is absent from S also calls Propagate before returning false.

The **Propagate** routine is similar to the one in Section 3.1. As mentioned in Section 4.1, each update uses a thread-local stack to keep track of the Nodes that it visits on the way from *Root* to the location where the update must be performed, so Propagate can simply pop these Nodes off the stack and perform a double Refresh on each of them. Some of the Nodes along the path may have been removed from the Node tree by other Delete operations that are concurrent with the update, but there is no harm in applying a double Refresh to those deleted Nodes.

As in Section 3.1, each **Refresh** on a Node x reads the Versions of x's children and combines the information in them to create a new Version for x, and then attempts to CAS a pointer to that new Version into $x.version$. There is one difference in the Refresh routine: because x's child pointers may be changed by

concurrent updates, Refresh reads x's child pointer, reads that child's *version* field, and then reads x's child pointer again. If the child pointer has changed, Refresh does the reads again, until it gets a consistent view of the child pointer and the *version* field of that child.

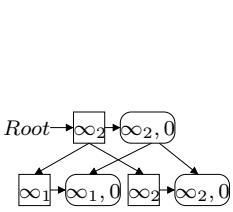
A **query operation** first reads *Root.version* to get the root of a Version tree. This Version tree is an immutable BST (with *sum* fields) whose leaves form a snapshot of the keys in S at the time *Root.version* is read. The query is linearized at this read. Then, the standard, sequential algorithm for an order-statistic query can be run on that Version tree. To ensure linearizability, searches are performed like other queries. This has the additional benefit of making searches wait-free, unlike the original BST of [19], where searches can starve. Complex queries, like range queries, can access any subset of Nodes in the snapshot. Our technique provides snapshots in a simpler way than [21] (later generalized by [38] to any CAS-based data structure), which keeps a list of previous timestamped versions of each child pointer. Our approach makes queries more efficient since they do not have to search back through version lists for an old version with a particular timestamp. It also avoids many of the problems of garbage collection, since old Versions are automatically disconnected from our data structure when a new Version replaces it. Unlike [21], our approach does not provide a snapshot of the Node tree: the shape of the Version tree may not match the shape of the Node tree at any time. Instead, our approach provides a snapshot of the *set of elements represented by the tree*.

4.3 Correctness

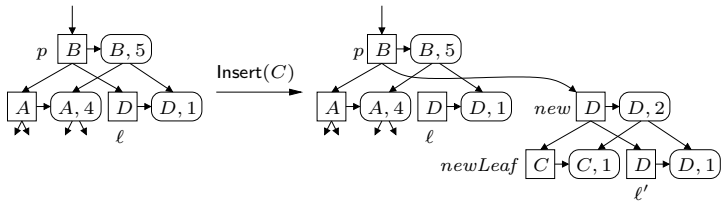
As in Section 3.2, we define arrival points of update operations at a Node to keep track of when the updates have been propagated to that Node. We again linearize updates at their arrival point at the root Node, and queries when they obtain a snapshot of the Version tree by reading *Root.version*. As in [19], sentinel Nodes as shown in Figure 8a ensure that the root Node never changes.

The proof again has two main parts: showing that every update operation has an arrival point at the root, and showing an invariant that in every configuration C, the Version tree rooted at a Node x is a legal (augmented) BST containing the set that would result from sequentially performing all operations that have arrival points at x at or before C, in the order of their arrival points. The former claim implies that the linearization respects the real-time order of operations. Applying the latter claim to the root shows that queries return results consistent with the linearization ordering.

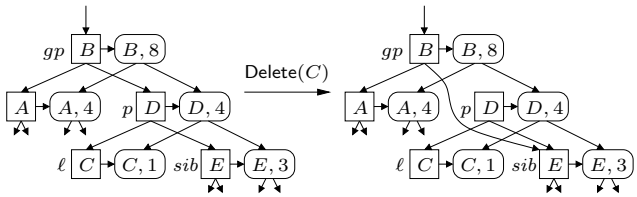
Although this high-level plan for the proof is similar to Section 3.2, updates' changes to the Node tree introduce some challenges. Firstly, we must ensure that updates are not "lost" if concurrent updates remove the Nodes to which they have propagated. This involves transferring arrival points from the removed Node x to another Node x', and this requires proving a number of claims about the arrival points that can be present at x and x' to ensure that transferring arrival points from x to x' does not change the set of keys that should be stored in the Version tree of x'. Secondly, in the original, unaugmented BST of [19],



(a) Initialization of augmented BST with sentinel Nodes.



(b) Addition of new Nodes and Versions for an Insert. In this example, the subtree rooted at A has four leaves. The data structure is shown after the three new Nodes have been added to the Node tree, but before the change has been propagated to Node B's Version.



(c) Change to the Node tree for a Delete. The subtrees rooted at A and E have four and three leaves, respectively. The data structure is shown after the Nodes C and D have been removed from the Node tree, but before the change has been propagated to Node B's Version.

Figure 8: Augmented BST data structure. Nodes are shown as squares and Version objects as ovals with *key* and *sum* fields shown.

an Insert(k) that reaches a leaf that already contains k returns false, but that leaf may no longer be in the tree when the Insert reaches it, so the linearization point of the Insert is retroactively chosen to be some time during the Insert when that leaf was present in the tree. We must do something similar in choosing the arrival point of failed updates at a leaf.

In this section, we define the arrival points (which in turn defines the linearization) and sketch some of the key arguments about them. A detailed proof of linearizability appears in Section D. For a configuration C, let T_C be the Node tree in configuration C: this is the tree of all Nodes reachable from *Root* by following child pointers. Since our augmentation does not affect the way the Node tree is handled, it follows from [19] that T_C is always a BST. The *search path for key k in C* is the root-to-leaf path in T_C that a BST search for key k would traverse. The following intuition guides our definition of arrival points: the arrival point of an update operation *op* on key k at a Node x should be the first time when both (a) x is on the search path for k and (b) the effect of *op* is reflected in the Version tree rooted at $x.version$. We also ensure that, for any configuration C, the Nodes at which an operation has arrival points defined will be a suffix of the search path for k in C.

A successful Insert(k), shown in Figure 8b, replaces a leaf ℓ containing some key k' by a new internal Node *new* with two children, *newLeaf* containing k, and ℓ', which is a new copy of ℓ. The CAS step that makes this change is the arrival point of the Insert(k) at *new* and $newLeaf$, since these Nodes' Version trees are initialized to contain a leaf with key k. There may also be many operations that had arrival points at ℓ before ℓ is replaced by ℓ' in the Node tree. For example, there may be an Insert(k'') followed by a Delete(k'') if ℓ is the end of the search path for k''. If these operations have not propagated to the root, we must ensure that this happens, so that they are linearized: we do not want to lose the arrival points of these operations when ℓ is removed from the Node tree. So, we transfer all arrival points of update operations at ℓ to *new* and the appropriate child of *new* (depending on whether the key of the update is less than $new.key$ or not).

Similarly, when a Delete(k) changes the Node tree as shown in Figure 8c, each operation with an arrival point at the deleted leaf ℓ (and the Delete(k) itself) is assigned an arrival point at ℓ's sibling *sib*, and at all of *sib*'s descendants on the search path for the operation's key. We shall show that operation's key cannot appear in the Version trees of any of those Nodes, so the Version trees of those Nodes correctly reflect the fact that the key has been deleted.

As mentioned above, if an Insert(k) returns false because it finds a leaf ℓ containing k in the tree, it was shown in [19] that ℓ *was* on the search path for k in some configuration C during the Insert. Since augmentation has no effect on updates' accesses to the Node, this is still true for the augmented BST. We choose C as the arrival point of the Insert at that leaf. Similarly, if a Delete(k) returns false because its search for k reaches a leaf ℓ that does not contain k, we choose a configuration C during the Delete when ℓ is on the search path for k in T_C as the arrival point of the Delete at ℓ.

When a Refresh updates the *version* field of a Node x, we assign arrival

points to all update operations that had arrival points at x's children before the Refresh read the *version* fields of those children, as in Section 3.2. This indicates that those operations have now propagated to x, and the Version tree in $x.version$ reflects the fact that those updates have been performed.

Definition 2 in Section D formalizes the arrival points described above. We then use this definition to prove that each update operation's arrival point at the root is between the update's invocation and response (refer to Section D.2). In particular, this reasoning has to argue that no operation gets "lost" as it is being propagated to the root if concurrent deletions remove Nodes to which it has been propagated. Recall that Propagate calls a double Refresh on every Node in the update operation's local *stack*, which remembers all of the internal Nodes visited to reach the leaf ℓ where the update occurs. We use the fact from [19] that Nodes can be removed from the path that leads from the root to ℓ, but no new Nodes can ever be added to it. (It is fairly easy to see that the changes to the Node tree shown in Figure 7 cannot add a new ancestor to ℓ.) Thus, Propagate calls a double Refresh on every ancestor of ℓ to propagate the update all the way to the root Node.

Then, we prove our main invariant that in every configuration C and in each Node $x \in T_C$, the leaves of the Version tree stored in $x.version$ contain exactly those keys that would be obtained by sequentially performing the operations with arrival points at x at or before C, in the order of their arrival points (see Invariant 8 in Section D.3.). This argument is complicated by the fact that the Node tree changes and arrival points are shifted from one Node to another. We make the argument by focusing on one key k at a time, and showing that k is in the Version tree if and only if the last operation on key k in the sequential execution is an Insert. We also show that the boolean responses these operations will return are consistent with this sequential ordering. Applying this invariant to the root Node, this implies updates return responses consistent with the linearization ordering.

Unlike the trie in Section 3, the subtree rooted at Node x may have a different shape than the Version tree rooted at $x.version$, since updates may have changed the Node tree since $x.version$ was stored. However, we prove (in Invariant 6) that for any configuration C and any Node $x \in T_C$, the keys of update operations with arrival points at Nodes in the left (or right) subtree of x are less than $x.key$ (or greater than or equal to $x.key$, respectively). Together with the main invariant mentioned above, this allows us to prove that all Version trees are legal BSTs. The correctness of the augmentation fields is trivial, since these fields are correct when an internal Version is created, and its fields are immutable. It follows that queries return results consistent with the linearization.

4.4 Complexity

The amortized time per operation on the unaugmented BST is $O(h + c)$, where h is the height of the Node tree and c is point contention [19]. Since we have not made any change to the way the Node tree is handled, we must just count the additional steps required for the augmentation. We argue that the amortized

time to perform a Propagate is also $O(h + c)$. The number of iterations of the loop in Propagate is bounded by the number of elements pushed on to the stack by the update, which in turn is bounded by the running time of the update in the original algorithm of [19]. Recall that a Refresh may have to reread child pointers repeatedly until it gets a consistent view of the child pointer and the child's *version* field. Rereading is necessary only if the child pointer changes between two successive reads. Thus, there are at most c re-reads caused by each change to a child pointer (namely by those Refresh operations running when the change happens). Moreover, there is at most one child pointer change for each update operation. Thus, the amortized running time per update operation remains $O(h + c)$. Since queries begin by taking a snapshot of the Version tree, queries are wait-free and can be accomplished in the same time that they would require in the sequential setting. For example, searches and order-statistic queries take $O(h)$ steps.

4.5 Extensions

The variants of the trie described in Section 3.4 apply equally to the BST.

The approach of Section 3.5 can be applied to our BST in exactly the same way so that, even though the Node tree is unbalanced, *Root.version* points to a *balanced* Version tree containing the elements of the set. This facilitates queries that can be done in the same time as in a sequential augmented balanced BST. For example, order-statistic queries can all be answered in $O(\log n)$ time where n is the size of the set. This does, however, increase the amortized time for update operations, which can be bounded using the argument of Section 3.5 by $O((h + c) \log \hat{n})$, where $\hat{n}$ is a bound on the size of the set under any possible linearization of the execution.

5 Future Work

Our technique provides lock-free implementations of many tree data structures based on augmented trees supporting insertions, deletions, and arbitrarily complex queries.

Although we base our augmented BST on [19], we believe our technique could also be applied to the similar lock-free BST design of Natarajan, Ramachandran and Mittal [30] or other lock-free trees. It would be interesting to apply it to a balanced tree such as the lock-free chromatic BST of [13] or to a self-balancing concurrent tree such as the CB Tree [2]. In particular, this would require ensuring the Propagate routine works correctly with rotations used to rebalance the tree. The technique may also be applicable to trees that use other coordination mechanisms, such as locks (e.g., [30]).

Could our technique be extended to obtain lock-free implementations of sequential augmented data structures that require more complex updates (such as the insertion of a pair of keys)? In the sequential setting, examples of such data structures include link/cut trees [35] and segment trees [7, 8]. Shafiei [34]

described a mechanism for making multiple changes to a tree appear atomic, but it would require additional work to find a suitable way to generalize our Propagate routine with her approach.

References

[1] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proc. 27th ACM Symposium on Theory of Computing*, pages 538–547, New York, NY, USA, 1995.

[2] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert Endre Tarjan. The CB tree: a practical concurrent self-adjusting search tree. *Distributed Computing*, 27(6):393–417, 2014.

[3] Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. In *Proc. 23rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 14–27, 2018.

[4] Shalom Asbell and Eric Ruppert. A wait-free deque with polylogarithmic step complexity. In *Proc. 27th International Conference on Principles of Distributed Systems*, volume 286 of *LIPIcs*, pages 17:1–17:22, 2023.

[5] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. *ACM Trans. Parallel Comput.*, 7(3), June 2020.

[6] Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. Space and time bounded multiversion garbage collection. In *Proc. 35th International Symposium on Distributed Computing*, volume 209 of *LIPIcs*, pages 12:1–12:20, 2021.

[7] J. L. Bentley. Solutions to Klee's rectangle problems. Technical report, Carnegie-Mellon University, Pittsburgh, PA, 1977.

[8] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[9] Guy E. Blelloch and Yuanhao Wei. VERLIB: concurrent versioned pointers. In *Proc. 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 200–214, 2024.

[10] Prosenjit Bose, Marc J. van Kreveld, Anil Maheshwari, Pat Morin, and Jason Morrison. Translating a regular grid over a point set. *Computational Geometry*, 25(1–2):21–34, 2003.

[11] Peter Brass. *Advanced Data Structures*. Cambridge University Press, 2008.

[12] Trevor Brown and Hillel 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.

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

[14] Tushar Deepak Chandra, Prasad Jayanti, and King Tan. A polylog time wait-free construction for closed objects. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 287–296, 1998.

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

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 17. MIT Press, fourth edition, 2022. Chapter on augmenting data structures (in particular RBTs).

[17] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic optimality–almost. *SIAM Journal on Computing*, 37(1):240–251, 2007.

[18] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[19] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing*, pages 332–340, 2014. References to lemmas are to the full version, available online from `https://users.ics.forth.gr/~faturu/BSTproof.pdf`.

[20] Panagiota Fatourou and Nikolaos D. Kallimanis. The RedBlue family of universal constructions. *Distributed Computing*, 33(6):485–513, 2020.

[21] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 275–286, 2019.

[22] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[23] Gaston H. Gonnet, J. Ian Munro, and Derick Wood. Direct dynamic structures for some line segment problems. *Computer Vision, Graphics and Image Processing*, 23(2):178–186, 1983.

[24] Prasad Jayanti. f-arrays: implementation and applications. In *Proc. 21st ACM Symposium on Principles of Distributed Computing*, pages 270–279. ACM, 2002.

[25] Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *Proc. 25th International Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 3821 of *LNCS*, pages 408–419, 2005.

[26] Donald E. Knuth. *The Art of Computer Programming, Volume 3*. Addison-Wesley, second edition, 1998.

[27] Ilya Kokorin, Dan Alistarh, and Vitaly Aksenov. Wait-free trees with asymptotically-efficient range queries. Manuscript available from https://arxiv.org/abs/2310.05293, 2023.

[28] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[29] Hossein Naderibeni and Eric Ruppert. A wait-free queue with polylogarithmic step complexity. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 124–134, 2023.

[30] Aravind Natarajan, Arunmoezhi Ramachandran, and Neeraj Mittal. FEAST: a lightweight lock-free concurrent binary search tree. *ACM Transactions on Parallel Computing*, 7(2), May 2020.

[31] Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 368–384, 2022.

[32] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proc. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 151–160, 2012.

[33] Gal Sela and Erez Petrank. Concurrent size. *Proc. of the ACM on Programming Languages*, 6(OOPSLA2):345–372, 2022.

[34] Niloufar Shafiei. Non-blocking patricia tries with replace operations. *Distributed Computing*, 32(5):423–442, 2019.

[35] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.

[36] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. PAM: parallel augmented maps. In *Proc. 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 290–304, 2018.

[37] Robert Endre Tarjan. *Data Structures and Network Algorithms*, chapter 4.2. SIAM, Philadelphia, USA, 1983.

[38] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021.

[39] Yuanhao Wei, Guy E. Blelloch, Panagiota Fatourou, and Eric Ruppert. Practically and theoretically efficient garbage collection for multiversioning. In *Proc. 28th ACM Annual Symposium on Principles and Practice of Parallel Programming*, pages 66–78, 2023.

Appendix

A Proof of Correctness for the Wait-Free Static Trie

In this appendix, we give a full proof of linearizability for the augmented trie of Section 3. Recall that U_x is the sequence of keys from the universe U that are represented in the leaves of the subtree of Nodes rooted at Node x in the order they appear from left to right. More formally, we can define U_x inductively from the bottom of the tree to the top.

$$\begin{aligned} U_{Leaf[k]} &= \langle k \rangle && \text{if } 0 \leq k \leq N-1 \\ U_x &= U_{x.left} \cdot U_{x.right} && \text{if } x \text{ is an internal Node} \end{aligned}$$

We start with a straightforward invariant that, for any Node x, the subtree of Versions stored rooted at $x.version$ has the same shape as the subtree of Nodes rooted at x.

Invariant 1. *For any Node x, the tree of Version objects rooted at $x.version$ is a perfect binary tree with $|U_x|$ leaves.*

Proof. The initialization of our data structure ensures that this is true in the initial configuration. We argue that every update to the *version* field of a $Node$ preserves the invariant. If a CAS on line 18 or 26 updates the *version* field of a leaf Node, it installs a pointer to a Version object created on line 17 or 25 with no children, so the invariant is preserved. If a CAS on line 34 installs a new Version v in the *version* field of an internal Node x at height h, v's left and right subtrees T_L and T_R are read from $x.left.version$ and $x.right.version$ at lines 31 and 32. Assuming the claim holds at all times before the CAS, T_L and T_R are perfect binary trees with $|U_{x.left}| = 2^{h-1}$ and $|U_{x.right}| = 2^{h-1}$ leaves, respectively, so v is the root of a perfect binary tree with $2^h = |U_x|$ leaves. □

The following invariant that *sum* fields are correctly computed is trivial, since it is satisfied whenever an internal Version is created at line 33, and fields of Versions are immutable.

Invariant 2. *For every internal Version v, $v.sum = v.left.sum + v.right.sum$.*

A.1 Linearization Respects Real-Time Order

For $k \in U$, let $Path(k)$ be the path of Nodes in the tree from $Leaf[k]$ up to the root. In this section, we show that arrival points of update operations on key k at all Nodes along $Path(k)$ are during the execution interval of the update. In particular, this means each update's linearization point is during the update, since we define the linearization point to be the arrival point at the root Node. This ensures that if one update ends before another begins, the first appears earlier in the linearization, as required by the definition of linearizability.

The first easy lemma shows the arrival points are after the invocation of the update.

Lemma 1. *The arrival point of an update operation on a key k at a node x in $Path(k)$ is after the start of the update.*

Proof. Consider an update operation U that is either an Insert(k) or a Delete(k) operation. Let $x_0, \ldots, x_m$ be the Nodes of $Path(k)$. We prove the claim holds for all x_i by induction on i. For the base case, it follows from parts 1 to 3 of Definition 1 that U's arrival point at $x_0 = Leaf[k]$ is after the start of U. For the induction step, if the claim holds for x_{i-1}, it follows from parts 4 and 5 of Definition 1 that it holds for x_i, since U must arrive in x_{i-1} before it can arrive in x_i. □

The next few results lead to Lemma 2, which states that every update operation has an arrival point at the root Node before it terminates. The following observation is an immediate consequence of parts 4 and 5 of Definition 1.

Observation A.1. *For any call to Refresh(x) on an internal Node x that performs a successful CAS at line 34, each operation whose arrival point at a child of x is before the Refresh(x) performs line 30 has an arrival point at x no later than the CAS of that Refresh.*

Definition 1 ensures that the arrival point of an update operation at a Node is unique if it exists. The following Lemma shows that the arrival point of a completed update on key k exists at all Nodes along $Path(k)$. To prove this, we observe that an update operation with key k calls Propagate, which performs a double Refresh on all the ancestors of $Leaf[k]$. We show in Lemma 2 that the update has an arrival point at each ancestor before the end of the double Refresh on it. It is proved using the following simple observation.

Observation A.2. *If two calls to Refresh on the same Node both perform successful CAS steps, then one performs the read on line 30 after the CAS on line 34 of the other.*

Proof. Let x be an internal Node. Only a successful CAS on line 34 can update $x.version$. Whenever this happens, a pointer to a Version object that was newly created on line 33 is stored in $x.version$. Thus, the value stored in $x.version$ was never stored there before. It follows that there cannot be a successful CAS between the time $x.version$ is read on line 30 of a Refresh(x) and the time that Refresh performs a successful CAS on line 34. The claim follows. □

Lemma 2. *Each completed update operation has an arrival point at the root Node before the end of the update.*

Proof. Consider an Insert(k) or Delete(k) operation U. Let $x_0, \ldots, x_m$ be $Path(k)$.

We show by induction that for $0 \leq i \leq m$, the update has an arrival point at x_i before U's invocation of Propagate has completed i iterations of the loop in lines 36–39.

For the base case, it follows from parts 1 to 3 of Definition 1 that U's arrival point at $Leaf[k]$ is before the time U calls Propagate on line 19 or 27. So, the claim holds after 0 iterations of the loop.

For the induction step, let $i > 0$ and assume the claim holds after $i-1$ iterations of the loop. The ith iteration of the loop does a double Refresh on x_i. If either invocation of Refresh(x_i) at line 37 or 38 of the ith loop iteration performs a successful CAS then the claim follows from Theorem A.1. Otherwise, let R_1 and R_2 be the two invocations of Refresh(x_i), which both perform an unsuccessful CAS. Then, there must have been two successful CAS steps c_1 and c_2 by other processes between line 30 and 34 of R_1 and R_2, respectively. (See Figure 5.) By Theorem A.2, the invocation R of Refresh(x_i) that performed c_2 must have read $x_i.version$ at line 30 *after* c_1, which is after the beginning of R_1. By the induction hypothesis, U has an arrival point at x_{i-1} before the beginning of R_1 and therefore before R performs line 30. Thus, by Theorem A.1 applied to R, U has an arrival point at x_i that is no later than R's CAS, which is before the end of R_2. The claim follows for x_i. □

Lemma 2 shows that each completed update has a well-defined linearization point before it terminates. Together with Lemma 1, this means that the linearization point of each update is during its execution interval.

A.2 Linearization is Consistent with Responses

The goal of this section is to prove that operations in the concurrent execution return the same results that they would if they were performed sequentially in their linearization order. To do this, we prove Invariant 4, which says that for every Node x, the Version tree rooted at $x.version$ stores the set of keys that would be present after performing the operations with arrival points at x, in the order of their arrival points.

The following Lemma is helpful for proving Invariant 4 for leaf Nodes. It shows that groups of operations that arrive at a leaf at the same time are of the same type.

Lemma 3. *Let k be any key. If the arrival points of several operations at the leaf Node x for key k are at the same step, then the operations are either all Insert(k) operations or all Delete(k) operations.*

Proof. Multiple operations can be assigned the same arrival points at the leaf Node x only at a successful CAS step on $x.version$. Consider a successful CAS s performed at line 18 of an Insert(k) operation I. (The argument if the CAS is performed by an Delete is similar.) Then, parts 1 and 2 of Definition 1 specify that s is the arrival point at x of I and some update operations that fail their CAS on $x.version$. Since s succeeds, the value of $x.version$ in the configuration C before s is the same value that I read on line 14. Since I's test on line 15 was true, $x.version.sum$ must have been 0 in C. Let U be another update operation whose arrival point at x is s. We argue that U must be an Insert(k). By part 2 of Definition 1, s is the first successful CAS on $x.version$ after U reads $x.version$

on line 14 or 22. Thus, U reads the same Version from $x.version$ as is still there in C. If U were a Delete operation, the test on line 23 would return false and U would not perform its CAS on line 26. □

For any configuration C, key k and Node x in $Path(k)$, let $Ops(C, x, k)$ be the sequence of update operations of the form Insert(k) or Delete(k) with arrival points at x prior to C, in the order of their arrival points at x.

The following lemma is useful for showing that updates on a particular key are linearized in the same order that they arrive at the leaf for that key, which will allow us to show that the results returned by operations are consistent with their linearization order.

Invariant 3. *Let C be any configuration. If Node x is a non-root ancestor of the leaf Node for key k, then $Ops(C, x.parent, k)$ is a prefix of $Ops(C, x, k)$.*

Proof. In the initial configuration C_0, $Ops(C_0, x, k)$ and $Ops(C_0, x.parent, k)$ are empty sequences, so the claim holds. Assuming the invariant holds at some configuration C, we show that it holds at the next configuration C'. If a step extends the sequence of operations with arrival points at x, the invariant is clearly preserved. Consider a step that extends the sequence of operations with arrival points at $x.parent$. Since $Ops(C, x.parent, k)$ is a prefix of $Ops(C, x, k)$ and Parts 4 and 5 of Definition 1 imply that $Ops(C', x.parent, k)$ is obtained by appending to $Ops(C, x.parent, k)$ all operations of $Ops(C, x, k)$ that do not already appear in $Ops(C, x.parent, k)$ (in the order they appear in $Ops(C, x, k)$), we have $Ops(C', x.parent, k) = Ops(C, x, k) = Ops(C', x, k)$ so the invariant is satisfied in C'. □

Corollary A.3. *Let C be any configuration. If x is the leaf Node for key k, then $Ops(C, Root, k)$ is a prefix of $Ops(C, x, k)$.*

Proof. This follows from applying Invariant 3 repeatedly along the path from x to the root Node. □

Now we are ready to prova a key invariant for correctness. Invariant 1 ensures that the leaves referred to in Invariant 4, below, are well-defined, and that Invariant 4 fully specifies what values the *sum* fields of all of the leaves in Version trees have.

Invariant 4. *The following holds in every configuration C. For each Node x, and $1 \leq i \leq |U_x|$, if k is the ith key in U_x then the ith leaf in the tree of Version objects rooted at $x.version$ has sum 1 if and only if $Ops(C, x, k)$ is non-empty and its last operation is an Insert(k).*

Proof. In the initial configuration C_0, $Ops(C_0, x, k)$ is empty and all Versions are initialized with *sum* field equal to 0.

Assume the claim holds in all configurations before a step s to prove that it holds in the configuration after s. Let C be the configuration immediately before s and C' be the configuration immediately after s. The truth of the invariant can be affected by s only if s is a successful CAS step on the *version*

field of a Node or the arrival point of some operation(s) at a Node. We consider all such steps in several cases.

Suppose s is a read step at line 14 that is the arrival point of an $\mathsf{Insert}(k)$ at k's leaf Node x, in accordance with part 3 of Definition 1. Then, $x.version.sum = 1$ in both C and C'. Because the invariant holds in C, and $Ops(C', x, k) = Ops(C, x, k) \cdot \mathsf{Insert}(k)$, the invariant holds in C' at x.

The argument if s is a read step at line 22 that is the arrival point of a $\mathsf{Delete}(k)$ at k's leaf is symmetric.

Suppose s is a successful CAS on the *version* field of a leaf Node x at line 18. By Lemma 3, the operations whose arrival points are at s are all $\mathsf{Insert}(k)$ operations. Thus, $Ops(C', x, k)$ is obtained from $Ops(C, x, k)$ by appending one or more $\mathsf{Insert}(k)$ operations to the end of the sequence. Moreover, $x.version.sum = 1$ in C', so the claim holds for x in C'.

The argument if s is a successful CAS on the *version* field of a leaf x at line 26 is symmetric.

Now, suppose s is a successful CAS of some call R of $\mathsf{Refresh}(x)$ that stores the Version v in the *version* field of an internal Node x. According to parts 4 and 5 of Definition 1, this CAS may be the arrival point in x of some number of updates involving keys in U_x.

We first consider the case where k is in $U_{x.left}$. Prior to R's CAS, R reads some Version v_L from $x.left.version$ on line 31. Let C_L be the configuration immediately after this read. Line 33 sets $v.left = v_L$. It follows from Theorem A.2 that any call to $\mathsf{Refresh}(x)$ that performs a successful CAS on $x.version$ strictly before s does its CAS, and that CAS is before C_L. Hence, any such $\mathsf{Refresh}$ also reads $x.left.version$ on line 31 before C_L. It follows from part 4 of Definition 1 and Invariant 3 that $Ops(C', x, k) = Ops(C_L, x.left, k)$. Moreover, the ith key of U_x is also the ith key of $U_{x.left}$ and the ith leaf of the tree rooted v is the ith leaf of the tree rooted at v_L. So, the last operation in $Ops(C', x, k)$ is an $\mathsf{Insert}(k)$ if and only if the last operation in $Ops(C_L, x.left, k)$ is an $\mathsf{Insert}(k)$, by our assumption that the claim holds in configuration C_L before s. The latter statement is true if and only if the *sum* field of the ith leaf of the subtree rooted at v_L is 1, which is true if and only if the *sum* field of the ith leaf of the subtree rooted at v is 1. This proves that the claim holds at x in C' because s stores v in $x.version$.

The argument for the $(i - |U_{x.left}|)$th key in $U_{x.right}$, which is the ith key in U_x, is similar. □

Lemma 4. *Each completed update operation returns the same result as it would if all updates were performed sequentially in their linearization order.*

Proof. Consider a completed $\mathsf{Insert}(k)$ operation I. (The argument for a Delete operation is similar.) Let C be the configuration after I's arrival point at the root Node and let x be the leaf Node corresponding to key k. I should return false if and only if the last update operation on k preceding it in the linearization order is an $\mathsf{Insert}(k)$. Since operations are linearized according to their arrival points at the root Node, this is equivalent to saying that the last operation

before I in $Ops(C, Root, k)$ is an Insert. By Theorem A.3, this is equivalent to saying that the last operation before I in $Ops(C, x, k)$ is an Insert. So, we must show that I returns false if and only if the last operation before I in $Ops(C, x, k)$ is an Insert.

First, suppose I reads a Version whose *sum* is 1 at line 14. Then, I returns false. Moreover, in the configuration C_1 before I's arrival point at x, $x.version.sum = 1$, so by Invariant 4 the last operation in $Ops(C_1, x, k)$ is an Insert(k). Thus, since I is the only operation whose arrival point at x is in the step after C_1 (by part 3 of Definition 1), the last operation in $Ops(C, x, k)$ before I is an Insert(k), as required.

For the remainder of the proof, suppose I reads a Version whose *sum* is 0 at line 14.

Now, suppose I performs an unsuccessful CAS on line 18. In this case, I also returns false. According to Definition 1, I is among several updates whose arrival points are at one successful CAS, and I is not the first in this group. By Lemma 3, they are all Insert(k) operations, so the last operation before I in $Ops(C, x, k)$ is an Insert, as required.

Finally, suppose I performs a successful CAS on line 18. In this case, I returns true. Let C_2 be the configuration before that CAS. Since the CAS succeeds, the value of $x.version$ in C is the same as it was when I read it at line 14 and C. So, in C, $x.version.sum = 0$. Thus, $Ops(C_2, x, k)$ is either empty or ends with a Delete operation. Since I is the first operation whose arrival point at x is at the successful CAS, the last operation before I in $Ops(C, x, k)$ is not an Insert, as required. □

It remains to show that query operations can be linearized. We assume that a query begins by reading a Version v from $Root.version$, and then performs the same code on the tree of Versions rooted at v as it would in the sequential data structure. The linearization point of the query is when it reads $Root.version$. It follows from Invariant 4 that the leaves of the tree rooted at v accurately reflect the contents of the represented set S at the query's linearization point. By Invariant 2, all Versions in the Version tree rooted at v have *sum* fields that accurately reflect the *sum* fields of the leaves, so queries on this tree will return results consistent with their linearization points.

B Pseudocode for Faster Queries in Wait-free Trie

Here, we provide some more detailed information about the modification outlined in Section 3.5 to improve the time for searches and other order statistic queries to $O(\log |S|)$. These modifications apply equally to the trie of Section 3 and the BST of Section 4. The type definitions are given in Figure 9. The only difference for the Node type is that the *version* field now stores a pointer to (the root of) a RBT. We remark that the RBT is a standard node-oriented RBT where keys are stored in internal nodes as well as leaves. RBT nodes are represented using the RNode type. Each RBT node is augmented with a *sum*

```
200: type Node          ▷ used to store nodes of static trie structure
201:     Node* left, right ▷ immutable pointers to children Nodes
202:     Node* parent   ▷ immutable pointer to parent Node
203:     Rnode* version ▷ mutable pointer to RBT

204:
205: type Rnode         ▷ used to store a node of an RBT; replaces Version objects
206:     Rnode* left, right ▷ immutable pointers to children
207:     int sum        ▷ immutable number of keys in subtree rooted at Rnode
208:     U key          ▷ immutable key needed for searching RBT
209:     {red, black} colour ▷ immutable colour used for balancing RBT

210:
211: function BOOLEAN(Refresh)Node* x ▷ Try to propagate information to Node x from its children
212:     old ← x.version
213:     v_L ← x.left.version
214:     v_R ← x.right.version
215:     if v_L.sum = 0 then new ← v_R
216:     else if v_R.sum = 0 then new ← v_L
217:     else new ← Join(v_L, v_R) ▷ Non-destructive join of two RBTs into new RBT
218:     return CAS(x.version, old, new)
```

Figure 9: Modified type definitions and Refresh routine to support faster queries.

field that stores the number of elements in the subtree rooted at that node to facilitate order-statistic queries. All fields of an RBT node are immutable.

The initialization is the same as in Figure 3, except that all Node's *version* field can initially point to a single dummy Rnode with *sum* field 0 and Nil child pointers. The Insert and Delete routines are identical to Figure 3, except that lines 17 and 25 create a new RNode instead of a Version object. Line 17 must fill in *new*'s additional fields $key \leftarrow k$ and $colour \leftarrow$ black. The modified Refresh routine is shown in Figure 9. It uses a non-destructive implementation of the standard (sequential) Join algorithm to combine two RBTs into one. The Propagate routine is identical to Figure 3. Queries are done by reading *Root.version* and running standard (sequential) RBT query algorithms on it.

C Pseudocode for Lock-Free Augmented BST

Here, we give more details about how to augment the lock-free BST of Ellen et al. [19]. Type definitions are given in Figure 10. High-level pseudocode for Insert and Delete are given in Figure 11. These are mostly the same as in [19], except for the addition of calls to Propagate and the creation of Version objects used to initialize the *version* fields of Nodes created by Insert. Consequently, we do not give all the details of these routines; see EFHR14 for the detailed pseudocode. The new routines for handling Versions and some example queries are given in Figure 12.

An **Insert**(k) searches for k in the BST of Nodes and arrives at a leaf Node ℓ

```
100: type Node          ▷ used to store nodes of static trie structure
101:  | U key           ▷ immutable key of Node
102:  | Node* left, right ▷ mutable pointers to children Nodes
103:  | Version* version ▷ mutable pointer to current Version
104:  | Info* info      ▷ for coordinating updates; irrelevant to our augmentation
105:
106: type Version       ▷ used to store a Node's augmented data
107:  | U key           ▷ immutable key of Node this Version belongs to
108:  | Version* left, right ▷ immutable pointers to children Versions
109:  | int sum         ▷ immutable sum of descendant leaves' bits
110:
```

Figure 10: Object types used in lock-free augmented BST data structure.

containing some key k'. If $k' = k$, the value k is already in the BST, so the Insert does not need to modify the tree and will eventually return false. Otherwise, the Insert attempts to replace the leaf ℓ by a new internal Node whose key is $\max(k, k')$ with two new leaf children whose keys are $\min(k, k')$ and $\max(k, k')$. There are also some additional steps required to coordinate updates to the same part of the tree, and those steps may cause the attempt to fail, in which case the Insert tries again by backtracking up the tree and then searching down the tree for the correct place to try inserting the node again. The details of the inter-process coordination are not important to the augmentation. Before attempting to add the three new Nodes to the tree, the Insert creates a new Version object for each of them with fields filled in as shown in Figure 8b. To facilitate backtracking after an unsuccessful attempt, the Insert keeps track of the sequence of internal Nodes visited on the way to the location to perform the insertion in a *thread-local* stack. When an attempt of the Insert succeeds, it calls Propagate on the newly inserted internal Node and returns true. Propagate uses the thread's local stack to revisit the Nodes along the path from the root to the insertion location in reverse order, performing a double Refresh on each Node, as in Section 3. If the Insert terminates after finding the key is already present in a leaf Node, it calls Propagate on that leaf Node, to ensure that the operation that inserted that leaf Node has been linearized, and then returns false.

A **Delete**(k) has a very similar structure. It first searches for k in the BST of Nodes and arrives at a leaf Node ℓ. If ℓ does not contain k, then the Delete does not need to modify the tree and returns false after calling Propagate. Otherwise, the Delete uses a CAS to attempt to remove both ℓ and its parent from the tree. (See Figure 8c.) Again, there are some additional steps required to coordinate updates to the same part of the tree, which may cause the Delete's attempt to fail and retry, but the details are irrelevant to the augmentation. When an attempt of the Delete succeeds, it calls Propagate to perform a double refresh along a path to the root, starting from the internal Node whose child pointer is changed (i.e., the Node that was formerly the grandparent of the deleted leaf ℓ) and returns true.

The **Refresh**(x) routine is similar to the one in Figure 3. Because the struc-

```
109: Initialize the data structure as shown in Figure 8a, where Root is a shared pointer
110: function Boolean(Insert)Key k
111:     let stack be an empty thread-local stack
112:     push Root on to stack
113:     loop
114:         do a BST search for k from top Node on stack, pushing visited internal Nodes on stack
115:         let ℓ be the leaf reached by the search
116:         if ℓ.key = k then
117:             Propagate(stack)
118:             return false ▷ k is already in the tree
119:         let p be the top Node p on stack ▷ p was ℓ's parent during the search
120:         let new be a new internal Node whose children are a new leaf Node with key k and a
121:             new Leaf with ℓ's key. Each of the three new Nodes has a pointer to a new Version
122:             object with the same key as the Node. The leaf Versions have sum 1 (or 0 if the key
123:             is ∞1 or ∞2) and new.sum = new.left.sum + new.right.sum. (See Figure 8b.)
124:         attempt to change p's child from ℓ to new using CAS
125:         if attempt fails due to another update then
126:             help complete the update that caused the attempt to fail
127:             backtrack by popping stack until a node that is not marked for deletion is popped,
128:                 helping complete the deletion of each marked Node that is popped
129:         else ▷ new was successfully added to tree
130:             Propagate(stack)
131:             return true
132: function Boolean(Delete)Key k
133:     let stack be an empty thread-local stack
134:     push Root on to stack
135:     loop
136:         do a BST search for k from top Node on stack, pushing visited internal Nodes on stack
137:         let ℓ be the leaf reached by the search
138:         if ℓ.key ≠ k then
139:             Propagate(stack)
140:             return false ▷ k is not in the tree
141:         pop Node p from stack ▷ p was ℓ's parent during the search
142:         let gp be the top Node on stack ▷ gp was p's parent during the search
143:         attempt to change gp's child from p to ℓ's sibling using CAS
144:         if attempt fails due to another update then
145:             help complete the update that caused the attempt to fail
146:             backtrack by popping stack until a node that is not marked for deletion is popped,
147:                 helping complete the deletion of each marked Node that is popped
148:         else ▷ deletion removed k's Node from tree
149:             Propagate(stack)
150:             return true
```


Figure 11: Pseudocode for augmented BST. The code for updates is given at a high level. For details, see [19]. Changes to Insert and Delete to support augmentation is shaded.

```
152: function BOOLEAN(Refresh)Node* x ▷ Try to propagate information to Node x from its children
153:     old ← x.version
154:     repeat                ▷ Get a consistent view of x.left and x.left.version
155:         x_L ← x.left
156:         v_L ← x_L.version
157:     until x.left = x_L
158:     repeat                ▷ Get a consistent view of x.right and x.right.version
159:         x_R ← x.right
160:         v_R ← x_R.version
161:     until x.right = x_R
162:     new ← new Version with left ← v_L, right ← v_R, sum ← v_L.sum + v_R.sum
163:     return CAS(x.version, old, new)

164: function (Propagate)Stack* stack ▷ Propagate updates starting at top Node on stack
165:     while stack is not empty do
166:         pop Node x off of stack
167:         if not Refresh(x) then
168:             Refresh(x)▷ Do a second Refresh if first one fails

169: function BOOLEAN(Find)k ▷ Returns true if k is in the set, or false otherwise
170:     v ← Root.version
171:     while v.left ≠ Nil do
172:         if k < v.key then v ← v.left
173:         else v ← v.right
174:     return (v.key = k)

175: function U(Select)j ▷ Returns set's jth smallest element or Nil if no such element
176:     v ← Root.version
177:     if j > v.sum then ▷ Set contains fewer than j elements
178:         return Nil
179:     repeat                ▷ Loop invariant: desired element is jth in tree rooted at v
180:         if j ≤ v.left.sum then
181:             v ← v.left
182:         else
183:             j ← j − v.left.sum
184:             v ← v.right
185:     until v is a leaf
186:     return v.key

187: function INT(Size)
188:     return Root.version.sum
```

Figure 12: Pseudocode augmented BST, continued. We include Find, Select and Size as three examples of queries that use the augmentation.

ture of the BST's Node tree can change, the repeat loops ensure that the Refresh gets a consistent view of x's child pointer and the contents of that child's *version* field. The other difference is that line 162 stores $x.key$ in the *key* field of the new Version. The **Propagate** routine is identical to the one given in Figure 3, except that we cannot use parent pointers on line 166. Instead, an update operation keeps track of the sequence of Nodes that it traversed from the root to reach a node x and then does a double Refresh on each of them in reverse order (from x to the root).

A query operation is performed on a snapshot of the Version tree obtained by reading $Root.version$. This includes the **Find** operation, which simply performs a search on the Version tree as it would in a sequential BST. As an additional bonus, our Find operation is wait-free, unlike the original lock-free BST [19], where Find operations may starve.

D Correctness of Augmented BST

We now give a detailed proof of correctness for the augmented BST of Section 4. Throughout this section we consider an execution α of the implementation and show that it is linearizable.

D.1 Facts About the Unaugmented BST

We first summarize some facts from [19] about the original, unaugmented, lock-free BST that will be useful for our proof. Since our augmentation does not affect the Node tree, these facts remain true in the augmented BST.

In [19], the mechanism to coordinate updates ensures that a status field of an internal Node is *flagged* whenever one of the Node's child pointers is changed and that the Node's status field is permanently flagged before the Node is removed from the tree. (In [19], permanent flags are called "marks".) The following result is a consequence of this fact.

Lemma 5 (Lemma 14 and 17 of [19])**.** *A Node's child pointer can change only when the Node is not permanently flagged for deletion and it is reachable.*

Recall that T_C is the Node tree in configuration C and the search path for a key k in C is the path that would be taken through T_C by a sequential BST search for k in T_C. We say a Node is reachable in C if it appears in T_C.

The modifications that can occur in the Node tree are shown in Figure 7. Neither type of modification can ever give a Node a new ancestor (although a deletion may remove an ancestor). The next lemma is a consequence of this.

Lemma 6 (Lemma 19 of [19])**.** *If a Node is on the search path for key k in one configuration and is still reachable in some later configuration, then it is still on the search path for k in the later configuration.*

When an update operation on key k searches for the location to modify in the tree, it ignores flags. This means that it may pass through Nodes that have

been deleted by concurrent operations. The following lemma shows that even if this happens, each visited Node *was* on the search path for k at some time during the search. This is crucial for the correctness of the unaugmented BST, and it is also useful for proving that our augmented tree can be linearized.

Lemma 7 (Lemma 20 of [19]). *If an Insert(k) or Delete(k) visits a Node x during its search for the location of key k, then there was a configuration between the beginning of the operation and the time it reaches x when x was on the search path for k in the Node tree.*

An important fact is that all updates to the Node tree preserve the BST property.

Lemma 8 (Lemma 21 of [19]). *In all configurations C, T_C satisfies the BST property.*

D.2 Linearization Respects Real-Time Order

In this section, we show that Propagate succeeds in assigning arrival points to an update operation at each reachable Node it calls a double Refresh on, and that arrival point is during the execution interval of the operation. In particular, if the call to Propagate completes, the update is assigned an arrival point at the root Node that is during the operation. Since this is used as the linearization point of the update, and each query is also assigned a linearization points during the query, it follows that the linearization respects the real-time order of operations.

We first formalize the definition of arrival points described in Section 4.3.

Definition 2. We first define the arrival point of each unsuccessful update at a leaf Node.

1. Consider an Insert(k) that arrives at a leaf Node ℓ with key k or a Delete(k) that arrives at a leaf Node ℓ that does not contain k. When searching for the location of k, the update arrives at ℓ by reading a child pointer of some Node p. The update operation's arrival point at ℓ is the last configuration C such that (1) C precedes the update's read of ℓ from a child pointer of p and (2) p is on the search path for k in C. (C exists by Lemma 7.)

We define the remaining arrival points inductively. Assuming the arrival points are defined for a prefix of the execution α, we describe the arrival points associated with the next step s of the execution.

We first consider steps that change the Node tree.

2. Consider a step s that changes the Node tree to perform an Insert(k). Then s replaces a leaf ℓ of the Node tree by an internal Node *new* with two leaf children, *newLeaf* which contains k, and ℓ', which contains the same key as ℓ (see Figure 8b). The step s is the arrival point at *new* of all operations whose arrival points at ℓ precede s (in the order they arrived at ℓ). Step s is also the arrival point for each of these operations at one

of *newLeaf* or ℓ' (in the same order): those operations on keys less than *new.key* go to the left child of *new*, and the rest go to the right child. Finally, the last arrival point placed at s is for Insert(k) at both *newLeaf* and *new*.

3. Consider a step s that changes the Node tree to perform a Delete(k). Then s removes a leaf ℓ containing k and ℓ's parent p from the Node tree (see Figure 8c) by changing the appropriate child pointer of ℓ's grandparent gp from p to ℓ's sibling *sib*. The step s is the arrival point of the Delete(k) at *sib* and at every descendant of *sib* on the search path for k. Furthermore, for each update operation that has an arrival point at l, s is the arrival point of the operation at *sib* and at every descendant of *sib* that is on the search path for the key of the operation. If s is the arrival point of multiple operations at a Node, they are ordered as they were ordered at ℓ, with the Delete(k) last.

Finally, we consider a successful CAS s performed by a Refresh R.

4. Consider a Refresh(x) R that performs a successful CAS s on $x.version$ at line 163. Let x_L be the left child of x read by R's last execution of line 155. Step s is the arrival point at x of all operations that have an arrival point at x_L prior to R's last read at line 156 and do not already have an arrival point at x prior to s.

5. Consider a Refresh(x) R that performs a successful CAS s on $x.version$ at line 163. Let x_R be the right child of x read by R's last execution of line 159. Step s is the arrival point at x of all operations that have an arrival point at x_R prior to R's last read at line 160 and do not already have an arrival point at x prior to s.

If multiple operations' arrival points at an internal Node x are at the same successful CAS on $x.version$, we order them as follows: first the operations described in part 4 in the order of their arrival points at x_L and then the operations described in part 5 in the order of their arrival points at x_R.

For operations whose arrival points are defined by Part 1, the associated response is false. The Insert(k) whose arrival point is defined by Part 2 and the Delete(k) whose arrival point is defined in Part 3 get an associated response of true. Whenever arrival points are copied from one Node to another by any of Parts 2 to 5, the associated responses are copied as well.

As in Section A, the arrival points define a sequence of operations *Ops* at each Node. Now, we also attach a boolean response to each of the operations in the *Ops* sequence as we define the arrival points.

Definition 3. For each configuration C and Node x, let $Ops(C, x)$ be the sequence of update operations with arrival points at x that are at or before C, in the order of their arrival points at x. Each operation in the sequence is annotated with a boolean response. Let $Ops^*(C, x)$ be the sequence of update

operations with arrival points at x that are strictly before C, in the order of their arrival points at x. Let $Ops(C, x, k)$ be the subsequence of $Ops(C, x)$ consisting of operations with argument k.

Since the Refresh routine works in the same way as in Section 3, the following result has an identical proof to Theorem A.2.

Observation D.1. *If two calls to Refresh on the same Node both perform successful CAS steps, then one performs the read on line 153 after the CAS on line 163 of the other.*

The following claims are used to prove that no update is "lost" by the propagation technique if a Node where it has arrived is deleted from the tree.

Invariant 5. *For any configuration C and any internal Node x, each operation in $Ops(C, x)$ is also in $Ops(C, y)$ for some child y of x in C. Moreover, the set of operations in the Ops sequences of the children of x can only grow.*

Proof. The invariant holds vacuously in the initial configuration because no operations have arrival points. Assume that the claims hold up to some configuration C. We show that they hold up to the next configuration C'.

Part 1 of Definition 2 adds update operations to the *Ops* sequence only of leaf Nodes, so it trivially preserves the claims.

Part 2 of Definition 2 ensures that when an insertion occurs, all operations added to the new internal Node's *Ops* sequence are also added to the *Ops* sequence of one of its children (and that child is a leaf). (See Figure 8b.) Moreover, no violation of the invariant is created at the Node whose child pointer changes, since all arrival points of the replaced leaf are transferred to the new internal Node that replaces it.

Part 3 of Definition 2 ensures that when a deletion occurs, all operations added to any internal Node's *Ops* sequence are also added to one of its children's *Ops* sequence. (See Figure 8c.) Moreover, no violation of the invariant is created at the Node whose child pointer changes, since all arrival points of the old child came from one of its children (by the induction hypothesis) and all of these are transferred to the new child.

Consider the arrival points added to a Node x by Parts 4 and 5 of Definition 2 when the CAS of a Refresh operation updates $x.version$. All of the newly added operations had arrival points at x's children when their *version* fields were previously read by the Refresh, by the induction hypothesis. By the second claim of the induction hypothesis, these operations still have arrival points at one of x's children at or before C (even if x's children have changed), so the first claim is satisfied in C'. □

Next, we show that a successful Refresh propagates operations up the Node tree.

Lemma 9. *Suppose Node y is a child of Node x at a configuration C and that an update operation op has an arrival point at y at or before C. If Refresh(x) reads $x.version$ at line 153 after C and performs a successful CAS on line 163 then op has an arrival point at x at or before the CAS.*

Proof. Assume y is the left child of x; the case where it is the right child is symmetric. Let y' be the left child of x that the Refresh reads at line 155. By the second claim of Invariant 5, *op* has an arrival point at y' before this read. By Part 4 of Definition 2, *op* has an arrival point at x at or before the successful CAS of the Refresh. □

Now, we show that a double Refresh propagates operations, whether one of them performs a successful CAS or not. The following proof is similar to the induction step in the proof of Lemma 2.

Lemma 10. *Suppose Node y is a child of Node x at a configuration C and that an update operation op has an arrival point at y before C. If a process executes the double Refresh at lines 167–168 on x after C then op has an arrival point at x at or before the end of the double Refresh.*

Proof. If either Refresh has a successful CAS, the claim follows from Lemma 9. If the CAS steps performed by both calls R_1 and R_2 to Refresh(x) fail, then there must have been two successful CAS steps c_1 and c_2 on $x.version$ during R_1 and R_2, respectively. (See Figure 5.) By Theorem D.1, the invocation R of Refresh(x) that performed c_2 must have read $x.version$ at line 153 *after* c_1, which is after the beginning of R_1. Thus, by Lemma 9 applied to R, *op* has an arrival point at x no later than c_2, which is before the end of R_2. □

In Lemmas 11 to 14, we consider an update operation *op* on a key k and show that it has an arrival point at the root before it terminates. Let $x_1, \ldots, x_m$ be the Nodes on the local stack (from the newest pushed to the oldest) when *op* calls Propagate. The first arrival point of *op* at any Node is defined by Part 1, 2 or 3 of Definition 2. Let C_0 be the first configuration at or after this arrival point. Since x_1 is on the stack, it is an internal Node. Let x_0 be the left child of x_1 in C_0 if $k < x_1.key$ or the right child of x_1 otherwise. In Lemma 14, we wish to show by induction on i that before *op*'s Propagate completes its double Refresh on x_i, *op* has an arrival point at x_i. We first prove the base case in Lemma 11 by showing that *op* has an arrival point at x_0 before Propagate is called. Lemmas 12 and 13 are useful for the induction step of Lemma 14, which shows that if a double Refresh is called after *op* has an arrival point at x_{i-1}, then before the double Refresh completes, *op* will have an arrival point at x_i.

Lemma 11. *Part 1, 2 or 3 of Definition 2 defines the arrival point of op at x_0 at or before C_0 and C_0 is before op's call to Propagate. Moreover, x_0 is on the search path for k in C_0.*

Proof. We consider several cases.

First, suppose *op* is an Insert(k) whose test on line 116 returns true because it found a leaf ℓ that contains k. Part 1 of Definition 2 defines the arrival point of *op* at ℓ to be the latest configuration before the Insert reads ℓ as the child of x_1 in which x_1 is on the search path for k. By definition, C_0 is that configuration. Either C_0 is immediately before the read of x_0 from one of x_1's children or C_0 is an earlier configuration immediately before the step that removes x_1 from the

BST. In the latter case, x_1 is marked in C_0, so its child pointers cannot change thereafter (by Lemma 5), so x_0 is already the child of x_1 in the search path for k in C_0. Either way, ℓ is the next Node after x_1 on the search path for k in C_0, so $x_0 = \ell$, and C_0 is the arrival point of *op* at x_0. Finally, C_0 is during *op*'s search, by Lemma 7, so it is before *op* calls Propagate on line 117.

The claim can be proved similarly if *op* is a Delete(k) whose test on line 138 returns true because it found a leaf ℓ that does not contain k: in this case, x_0 is again ℓ.

Now suppose *op* is an Insert(k) that successfully adds a leaf containing k to the Node tree. Then, C_0 is the configuration immediately after the successful CAS that adds k to the BST. This step changes a child pointer of x_1, the top Node on *op*'s stack, to add three new Nodes to the Node tree. So, x_0 is the new internal Node created on line 120 of *op*. The successful CAS is the arrival point of the Insert at x_0, by Part 2 of Definition 2, and C_0 is the configuration after this CAS. This CAS may be performed by *op* itself or a helper, but, either way, it must be done before the Insert(k) reaches line 130, since it is the linearization point of the Insert in the unaugmented BST. It remains to show that x_0 is on the search path for k in C_0. By Lemma 7, x_1 was on the search path for k in some earlier configuration. By Lemma 5, x_1 is still reachable when the CAS occurs. By Lemma 6, this means that x_1 is still on the search path for k when the CAS occurs, so x_0 is on the search path for k in the configuration C_0 after this CAS.

Finally, suppose *op* is a Delete(k) that successfully removes a leaf containing k from the Node tree. Then, C_0 is the configuration immediately after the successful CAS that deletes k from the BST. This step changes a child pointer of x_1, the top Node on *op*'s stack, to the sibling of the deleted leaf. So, x_0 is this sibling. The CAS is the arrival point of the Delete at x_0, by Part 3 of Definition 2. For the same reason as for Inserts, this CAS is prior to *op*'s call to Propagate on line 149. The argument that x_0 is on the search path for k in C_0 is also similar: x_1 is on the search path for k in C_0, so x_0 is too. □

Since Nodes $x_1, \ldots, x_m$ were on the stack, they are all internal Nodes. This ensures that the Node x in the following claim is well-defined.

Lemma 12. *Consider any configuration C at or after C_0. Let x be the first node in the search path for k in C that is not in $\{x_1, \ldots, x_m\}$. Then, op is in $Ops(C, x)$.*

Proof. The claim is true in C_0, since Lemma 11 ensures x_0 is on the search path for k in C_0 and *op* is in $Ops(C_0, x_0)$.

As an induction hypothesis, assume the claim holds in some configuration C to prove that it holds in the next configuration C'. Let P be the search path for k in C and P' be the search path in C'. Let x and x' be the first Nodes on P and P', respectively, that are not in $\{x_1, \ldots, x_m\}$. If $x = x'$, then the claim trivially follows from the induction hypothesis. So, assume for the rest of the proof that $x \neq x'$. The step from C to C' must have changed the child pointer of some Node x_i to x'.

If this step changes the Node tree to perform an insertion, x_i's child before the CAS was a leaf and therefore is not in the set $\{x_1, \ldots, x_m\}$ of internal Nodes. Thus, that leaf must have been x, the first Node on P that is not in $\{x_1, \ldots, x_m\}$. By the induction hypothesis, *op* is in $Ops(C, x)$. By the second claim of Invariant 5, *op* is in $Ops(C', x')$.

If the step changes the Node tree to perform a deletion, the step removes a leaf Node ℓ and ℓ's parent p from the Node tree by changing a child pointer of x_i from p to ℓ's sibling x'. So, prior to the CAS, x must have been p (if $p \notin \{x_1, \ldots, x_m\}$) or ℓ (otherwise). We first argue that *op* is in either $Ops(C, x')$ or $Ops(C, \ell)$. If $x = p$, then *op* is in $Ops(C, p)$ by the induction hypothesis, and in one of $Ops(C, \ell)$ or $Ops(C, x')$ by Invariant 5. If $x = \ell$, then *op* is in $Ops(C, \ell)$ by the induction hypothesis. Finally, we argue that *op* is in $Ops(C', x')$. This is trivial if *op* is in $Ops(C, x')$. If *op* is in $Ops(C, \ell)$ then Part 3 of Definition 2 ensures that it is in $Ops(C', x')$. □

Next, we consider the nodes $x_1, \ldots, x_m$ on the stack on which *op*'s Propagate will perform a double Refresh. In the proof of Lemma 20 in [19], it is shown that there is a sequence of configurations $D_{m-1}, \ldots, D_1$ where D_i is at or before D_{i-1} for all i, such that x_i and x_{i-1} are consecutive Nodes on the search path for k in D_{i-1}. Intuitively, these configurations can be defined inductively as follows. D_{m-1} is when x_{m-1} is read from a child of the permanent root node x_m. Once D_i is defined, we consider two cases: if x_i is still on the search path for k when its child pointer to x_{i-1} is read during the search, then let D_{i-1} be this read; otherwise, let D_{i-1} be the configuration before x_i was deleted from the search path. In the latter case, x_i's child already pointed to x_{i-1} at D_{i-1} because x_i's child pointers cannot be changed after it is deleted, by Lemma 5.

Lemma 13. *In any configuration at or after D_i, if x_i is still in the search path for k, then x_i's ancestors are a subset of $x_{i+1}, \ldots, x_m$.*

Proof. The claim holds vacuously for the root Node x_m. Assume the claim holds for x_{i+1} to prove it for x_i. In the configuration D_i, x_{i+1} and x_i are consecutive Nodes on the search path for k. Since this configuration is after D_{i+1}, the ancestors of x_{i+1} are a subset of $\{x_{i+2}, \ldots, x_m\}$. Thus the claim holds for x_i in D_i. Any step that modifies the Node tree after D_i preserves the claim because it can only splice out a Node (for a deletion) or replace a leaf. (See Figure 7.) □

The next lemma implies that each completed update operation has a linearization point, which is before the update terminates.

Lemma 14. *Suppose an update operation op calls Propagate, which does a double Refresh on Nodes $x_1, x_2, \ldots, x_m$. Then op has an arrival point at x_i before the end of the ith iteration of the loop in Propagate.*

Proof. We prove by induction on i that for $0 \leq i \leq m$, *op* has an arrival point at x_i before the end of i iterations of the loop in Propagate. When $i = 0$, this follows from Lemma 11.

For the induction step, assume the claim holds for $x_0, \ldots, x_{i-1}$ and consider the ith iteration of the loop in Propagate. If one of $x_0, \ldots, x_{i-1}$ is a child of x_i at the beginning of the ith iteration of the loop, then the claim follows from Lemma 10. Otherwise, by Lemma 13, the child of x_i at the beginning of the ith iteration of the loop is the first Node on the search path for k that is not in $\{x_1, \ldots, x_m\}$. By Lemma 12, *op* has arrived at that Node before the ith iteration of the loop, so the claim follows from Lemma 10. □

It follows from Lemma 14 that *op* has an arrival point at the root Node before it terminates. Now it remains to show that the arrival point at the root Node is after *op* is invoked, which is quite easy.

Lemma 15. *The arrival point of an operation at any Node is after the operation is invoked.*

Proof. The first arrival point of an Insert at any Node is defined either by Part 1 or Part 2 of Definition 2. (All other parts define arrival points of Inserts that have already arrived at other Nodes.) For Part 2, the arrival point is when the tree is updated to perform the Insert. In [19], this modification to the tree is used as the linearization point of the Insert, so it was shown in [19, Lemma 27] that it is during the Insert. For Part 1, the arrival point is after the start of the Insert, by definition. The claim follows for Insert operations.

The proof for Delete operations is similar. □

Recall that the linearization point of an update operation is its arrival point at the root Node. It follows from Lemmas 14 and 15 that the linearization point of each update operation is within the execution interval of that operation.

D.3 Linearization is Consistent with Responses

In a configuration C, we say a Node is *reachable* if there is a path of child pointers from the root Node to it.

Invariant 6. *In every configuration C, for every internal Node x that is reachable in C,*

- *for every Node x_L that was in x's left subtree at or before C, every operation in $Ops(C, x_L)$ has a key less than $x.key$, and*
- *for every Node x_R that was in x's right subtree at or before C, every operation in $Ops(C, x_R)$ has a key greater than or equal to $x.key$.*

Proof. Initially, the invariant holds vacuously, since all *Ops* sequences are empty.

Assume the invariant holds in all configurations up to C and let s be a step from configuration C to another configuration C'. Consider any internal Node x that is reachable in C'. To prove the claim holds for x in C', we need only consider a step s if it modifies the subtree of the Node tree rooted at x, or if it modifies an *Ops* sequence of a Node in that subtree. We prove the first claim

for the left subtree of x; the arguments for the right subtree are symmetric in all cases.

First, consider a successful CAS step s that modifies the left subtree of x to perform an Insert(k). This step s changes a child pointer of a reachable Node p from a leaf ℓ (where ℓ is in the left subtree of x) to a new Node *new* that has two leaf children ℓ' and *newLeaf*. (See Figure 8b.) By Part 2 of Definition 2, the step s is the arrival point at both *new* and the appropriate child of *new* of all operations whose arrival points at ℓ precede s. By the induction hypothesis, since ℓ was in the left subtree of x in C, all of these operations had keys less than $x.key$. Step s is also the arrival point at *newLeaf* and *new* of the Insert of $newLeaf.key = k$. Since the Node tree is always a BST by Lemma 8, it follows that $k < x.key$.

Next, consider a successful CAS step s that modifies the left subtree of x to perform a Delete(k). This step removes a leaf ℓ and its parent p (which is in the left subtree of x) by changing a child pointer of p's parent gp, which is reachable, from p to ℓ's sibling *sib*. (See Figure 8c.) By Part 3 of Definition 2, s is the arrival point at *sib* and some of its descendants of operations in $Ops(C, \ell)$. Since ℓ is in the left subtree of x in C, the induction hypothesis ensures that all of these operations have keys less than $x.key$. Step s is also the arrival point of the Delete(k) at *sib* and some of its descendants. Since the Node tree is always a BST (by Lemma 8) and the key k appeared in Node ℓ in the left subtree of x, it follows that $k < x.key$.

Next, consider a successful CAS step s on the *version* field of some Node y that was in x's left subtree at some time before C. This step is performed by a Refresh(y). Let y_L and y_R be the Nodes that the Refresh reads from y's child fields at lines 155 and 159, respectively. If y is still reachable when y_L is read from its left child field, then y is in the left subtree of x at that time, since the only types of changes to the Node tree that can happen are those shown in Figure 7. So, y_L is also in the left subtree of x at that time. If y is not reachable when y_L is read from its left child field, then y_L was the left child of y when y was deleted (since y's child fields cannot change after it is deleted, by Lemma 5). So, y_L was in the left subtree of x just before y was deleted. Either way, y_L was in the left subtree of x before C. The argument for y_R is symmetric, so we can apply the induction hypothesis to both y_L and y_R. Step s is the arrival point at y of operations that have an arrival point at y_L or y_R prior to the reads at line 156 or 160, respectively. By the induction hypothesis, all such operations have keys less than k.

Finally we consider arrival points that are at configuration C' itself (and not at step s). By Part 1 of Definition 2, C' can be the arrival point of an Insert(k) operation at a leaf ℓ if its search arrived at ℓ containing k, and ℓ is reachable in C'. If ℓ was in the left subtree of x before C', then it still is in the left subtree of x because x and ℓ are reachable in C'. By the BST property of the Node tree (Lemma 8), $k < x.key$. Similarly, C' can be the arrival point at a leaf ℓ of a Delete(k) if its search arrived at a leaf ℓ that does not contain k, and ℓ is reachable in C'. If ℓ was in the left subtree of x before C', then it still is in the left subtree of x because x and ℓ are reachable in C'. So, by the BST property

of the Node tree, $\ell.key < x.key$. □

Leaf nodes satisfy the following claim because it holds whenever a leaf Node is created (on line 162) and the fields of leaf Nodes and Version objects are never changed.

Observation D.2. *For each leaf Node x, $x.version$ always points to a Version with key $x.key$ and no children.*

Invariant 7. *For all configurations C and all Nodes x that are reachable in C, if x_L and x_R are the left and right child of x in C, then*

- *for keys $k < x.key$, $Ops(C, x, k)$ is a prefix of $Ops(C, x_L, k)$, and*
- *for keys $k \geq x.key$, $Ops(C, x, k)$ is a prefix of $Ops(C, x_R, k)$.*

Proof. Initially, the invariant holds vacuously, since all *Ops* sequences are empty. We must show that the invariant is preserved by any change to *Ops* sequences of a Nodes, and by any change to the structure of the Node tree.

We first consider changes to the *Ops* sequences of Nodes. Adding an arrival point to a leaf Node clearly preserves the invariant. Part 1 of Definition 2 only adds arrival points to leaf Nodes. Whenever part 2 or part 3 of Definition 2 adds an arrival point of an operation to an internal Node, it also adds adds an arrival point of the operation to the appropriate child of that Node, so the invariant is preserved. It remains to check that a successful CAS step of a Refresh(x) that updates $x.version$ preserves the invariant at x. Parts 4 and 5 of Definition 2 may append operations to $Ops(C, x, k)$. By Invariant 6, the new operations can only come from x_L if $k < x.key$ or x_R if $k \geq x.key$. Thus, $Ops(C, x, k)$ simply becomes a longer prefix of $Ops(C, x_L, k)$ or $Ops(C, x_R, k)$.

Next, we show that changes to the structure of the Node tree preserve the claim. Consider a successful CAS step that modifies the Node tree to perform an Insert operation. It changes a child pointer of a reachable Node p from a leaf ℓ to a new Node *new*. (See Figure 8b.) Since the *Ops* sequence of ℓ is transferred to *new* by Part 2 of Definition 2, this change preserves the claim.

Now consider a successful CAS step that modifies the Node tree to perform a Delete operation. Let C be the configuration before the step and C' be the configuration after. This step removes a leaf ℓ and its parent p by changing a child pointer of p's parent gp, which is reachable, from p to ℓ's sibling *sib*. Assuming the claim holds in C, we must show that $Ops(C', gp, k)$ is a prefix of $Ops(C', sib, k)$ for all k. Without loss of generality, assume ℓ is p's left child and p is gp's right child, as shown in Figure 8c. (The other cases are argued in exactly the same way.)

First, consider keys $k \geq p.key \geq gp.key$. Since the claim holds in C, we have $Ops(C, gp, k)$ is a prefix of $Ops(C, p, k)$, which is a prefix of $Ops(C, sib, k)$. Thus, $Ops(C', gp, k) = Ops(C, gp, k)$ is a prefix of $Ops(C, sib, k) = Ops(C', sib, k)$ as required.

Now, consider keys k satisfying $gp.key \leq k < p.key$. Since the claim holds in C, $Ops(C, gp, k)$ is a prefix of $Ops(C, p, k)$, which is a prefix of $Ops(C, \ell, k)$.

So, $Ops(C, gp, k)$ is a prefix of $Ops(C, \ell, k)$. Moreover, $Ops(C, sib, k)$ is empty, by Invariant 6. Part 3 of Definition 2 ensures that the step transfers the arrival points at ℓ to sib, so we have $Ops(C', sib, k) = Ops(C, \ell, k)$. So, we have $Ops(C', gp, k) = Ops(C, gp, k)$ is a prefix of $Ops(C, \ell, k) = Ops(C', sib, k)$, as required. □

The following crucial invariant shows that in every configuration C, the Version tree stored in each Node x accurately reflects all the operations that have arrival points at x at or before C: the set of keys that would be obtained by performing the operations in $Ops(C, x)$ sequentially is exactly the set of keys that are actually stored in the leaves of the Version tree that $x.version$ points to in configuration C. When we talk about the set of keys in the leaves, we include only keys from U; we exclude the dummy leaves containing ∞_1 and ∞_2.

Invariant 8. *In every configuration C, for every Node x that is reachable in C,*

(1) the Version tree rooted at $x.version$ is a BST whose leaves contain exactly the keys that would be in a set after performing $Ops(C, x)$ sequentially, and

(2) the responses recorded in $Ops(C, x)$ are consistent with executing the operations sequentially.

Proof. In the initial configuration, the Version tree of each Node contains no keys (see Figure 8a), and all *Ops* sequences are empty, so the claims hold vacuously.

Assume the claims hold for the prefix of the execution up to some configuration C. We consider the next step s after C, and show the claims hold in the resulting configuration C'. We prove this in two parts: first, we show that a Node x's Version tree in C' contains the keys that should be present after doing the sequence of updates $Ops^*(C', x)$ (i.e., that the Version tree reflects the operations whose arrival points at x are at step s), and then show that all operations whose arrival points at x are at the configuration C' have no effect on the set of keys that should be stored in the Version tree rooted at x (i.e., that they are updates that should return false). We need only consider a step s that changes the data structure or adds an arrival point to some Node x (so that $Ops(C', x) \neq Ops(C, x)$). We consider several cases.

Case 1 Consider a successful CAS step s that performs an Insert(k) operation. This step changes a child pointer of a reachable Node p from a leaf ℓ to a new Node *new* that has two leaf children ℓ' and *newLeaf*, which contains k. (See Figure 8b.) Let k' be the key of ℓ and ℓ'. By Theorem D.2, ℓ's Version tree in C contains a single key k'. By induction hypothesis (1), performing $Ops(C, \ell)$ results in the set $\{k'\}$. Hence, for any key $k'' \neq k'$, $Ops(C, \ell, k'')$ does not end with an Insert. We must check that the claim is satisfied at each of *new*, *newLeaf* and ℓ'. (For any other reachable Node, the claim is true by the induction hypothesis.)

By Part 2 of Definition 2, $Ops^*(C', new) = Ops(C, \ell) \cdot \langle \mathsf{Insert}(k) : \mathsf{true} \rangle$. So, performing this sequence of operations would yield the set with two elements $\{k, k'\}$ (since $k \neq k'$ because the CAS would not be performed if the test on line 116 were true). This set matches the keys in the Version tree rooted at $new.version$ in C', and the $\mathsf{Insert}(k)$ would indeed return true in the sequential execution.

The operations of $Ops(C, \ell)$ are split among the Ops^* sequences of the leaves $newLeaf$ and ℓ' according to whether their keys are less than $new.key$ or not. But since, for any $k'' \neq k'$, $Ops(C, \ell, k'')$ does not end with an Insert, we see that performing $Ops^*(C', \ell')$ yields the set $\{k'\}$ and performing $Ops^*(C', newLeaf)$ (which also includes an $\mathsf{Insert}(k)$ at the end) yields the set $\{k\}$. These sets match the keys in the Version trees rooted at $\ell'.version$ and $newLeaf.version$, respectively. Claim (2) at ℓ' and $newLeaf$ also follow immediately from induction hypothesis (2) applied to ℓ.

Case 2 Consider a successful CAS step s that performs a $\mathsf{Delete}(k)$ operation. This step removes a leaf ℓ containing key k and its parent p by changing a child pointer of p's parent gp, which is reachable, from p to ℓ's sibling sib. (See Figure 8c.) Suppose ℓ is the left child of p. (The other case is symmetric.) This step also adds arrival points of operations in $Ops(C, \ell)$ to Nodes in the subtree rooted at sib, according to Part 3 of Definition 2. We must show that adding these new operations to the Ops sequences of Nodes in sib's subtree does not change the set of keys that should be stored in their Version trees. In other words, we must show that for each such Node z, the set of keys after doing $Ops(C, z)$ sequentially is the same as the set of keys after doing $Ops^*(C', z)$ sequentially. We do this by arguing about each key k' separately.

By Theorem D.2, $\ell.version$ is the root of a single-node Version tree containing only the key k. Thus, by the induction hypothesis, performing the operations in $Ops(C, \ell)$ sequentially yields the set $\{k\}$. Hence, performing the operations in the sequence $\sigma = Ops(C, \ell) \cdot \langle \mathsf{Delete}(k) : \mathsf{true} \rangle$ yields an empty set and the $\mathsf{Delete}(k)$ would return true in this sequential execution. Let k' be any key that appears in the sequence σ. Since performing the operations of σ sequentially yields the empty set, the last operation on k' in σ must be a $\mathsf{Delete}(k')$.

The step s adds arrival points of all updates in σ with key k' to sib and each of sib's descendants along the search path for k'. Let z be one of those Nodes. Since an operation with key k' had an arrival point at ℓ, which is the left child of p in C, we must have $k' < p.key$ by Invariant 6. Thus, since z is in the right subtree of p in C, no operations on k' can appear in $Ops(C, z)$, again by Invariant 6. By the induction hypothesis applied to z, this means that the Version tree rooted at $z.version$ does not contain k'. Thus, claim (1) is satisfied with respect to k', since $Ops^*(C', z, k')$ ends with a $\mathsf{Delete}(k')$. Moreover, by induction hypothesis (2), the operations in $Ops(C, \ell, k')$ are annotated with the response they would get if they were

executed sequentially. Since there are no operations on k' in $Ops(C, z)$, performing the operations of $Ops(C, \ell, k')$ after $Ops(C, z)$ would yield the same responses, so claim (2) is also satisfied with respect to key k'.

Case 3 Consider a successful CAS step s performed by line 163 of an execution R of Refresh(x). For any Node other than x, the claims are true by the induction hypothesis. Let x_L and x_R be the children of x read by R's last execution of line 155 and 159, respectively. Let C_L and C_R be the configurations before R's last reads of $x_L.version$ and $x_R.version$ on line 156 and 160. Let T_L and T_R be the Version trees whose roots are read by those two reads. Let T' be the tree whose root is stored in $x.version$ by step s.

By the construction on line 162, the root of T' is a Version with key $x.key$ and its left and right subtrees are T_L and T_R. By the induction hypothesis, T_L is a BST whose leaves contain exactly the keys that would be in a set after performing $Ops(C_L, x_L)$ sequentially. Similarly, T_R is a BST whose leaves contain exactly the keys that would be in a set after performing $Ops(C_R, x_R)$ sequentially. It follows from Invariant 6, that all keys in T_L are less than $x.key$ and all keys in T_R are greater than or equal to $x.key$. Thus, T' is a BST.

Let k be any key. To prove claim (1), we show that k appears in a leaf of T' if and only if $Ops^*(C', x, k)$ ends with an Insert(k). Assume $k < x.key$. (The case where $k \geq x.key$ has a symmetric argument.) Then, $Ops^*(C', x, k) = Ops(C_L, x, k) \cdot \tau \cdot \phi$, where τ is the sequence of operations on key k whose arrival points at x are between C_L and C, and ϕ is the sequence of operations on key k whose arrival points are at the CAS step s. Since the CAS on $x.version$ at line 163 of the Refresh R succeeds, $x.version$ does not change between R's read of $x.version$ at line 153 and the CAS. In particular, it does not change between C_L and C. Thus, the arrival points of operations in τ can only be those defined by Part 3 of Definition 2. It was argued in Case 2, above, that whenever the arrival point of an operation is added to a Node by Part 3 of Definition 2, the key of that operation does not already appear in the Node's Ops sequence, and moreover the last operation added to the Node's Ops sequence with that key is a Delete. In other words, one of $Ops(C_L, x, k)$ or τ must be empty, and if τ is non-empty, then it ends with a Delete(k). Similarly, if τ is non-empty, then $Ops(C_L, x_L, k)$ must be empty.

We consider two subcases.

Case 3(a) Consider the case where τ is empty. Then, $Ops^*(C, x, k) = Ops(C_L, x, k) \cdot \phi$. Since $Ops(C_L, x, k)$ is a prefix of $Ops(C_L, x_L, k)$ by Invariant 7, and the CAS step s adds arrival points to x for any operations in $Ops(C_L, x_L, k)$ that are not in $Ops(C, x, k)$, we have $Ops^*(C, x, k) = Ops(C_L, x_L, k)$. Applying the induction hypothesis (1) to x_L at configuration C_L, the key k appears in a leaf of the tree T_L if and only

if $Ops(C_L, x_L, k)$ ends with an Insert(k). Thus, k appears in a leaf of the tree T' if and only if $Ops^*(C, x, k)$ ends with an Insert(k), since the BST property of T' ensures that k can only appear in the left subtree T_L of T'. Moreover, since $Ops^*(C, x, k) = Ops(C_L, x_L, k)$, and the responses in $Ops(C_L, x_L, k)$ are consistent with the sequential execution by induction hypothesis (2), claim (2) holds for x in C also.

Case 3(b) Now, consider the case where τ is non-empty. As mentioned above, $Ops(C_L, x_L, k)$ and $Ops(C_L, x, k)$ must then be empty. Since ϕ consists of operations from $Ops(C_L, x_L, k)$, ϕ must also be empty. Thus, $Ops^*(C', x, k) = Ops(C_L, x, k) \cdot \tau \cdot \phi = \tau$. As mentioned above, τ must end with a Delete(k). Since $Ops(C_L, x_L, k)$ is empty, induction hypothesis (1) applied to x_L in C_L implies that k does not appear in any leaf of T_L. Thus, k does not appear in any leaf of T', since the BST property of T' ensures that k cannot appear in the right subtree T_R of T'. So, claim (1) is satisfied with respect to k. Since τ is obtained (via Part 1 of Definition 2) by copying an Ops sequence some Node x' to x, claim (2) with respect to key k follows from induction hypothesis (2) applied to that Node x' when in the configuration before the copying is done.

This completes the proof that the leaves of the Version tree that $x.version$ points to in C' are exactly the keys of the set obtained by doing the operations in $Ops^*(C', x)$. It remains to prove that the operations whose arrival points are at C' itself also preserve the invariant. Only Part 1 of Definition 2 assigns arrival points to configurations, and it assigns arrival points only at leaf Nodes. Thus, if x is an internal Node, $Ops(C', x) = Ops^*(C', x)$ and the proof of the invariant is complete. Suppose x is a leaf Node. By definition, all operations whose arrival points at a leaf x whose key is k are either Insert(k) operations or Delete(k') operations with $k' \neq k$, and the responses recorded for these operations are all false. By Theorem D.2, $x.version$ points to a BST containing just one leaf with key k. Since we have already shown that performing the operations of $Ops^*(C', x)$ sequentially results in the set $\{k\}$, all the operations whose arrival points are at C' would not affect the set if they were performed sequentially after $Ops^*(C', x)$, and they would all return false in this sequential execution. Thus, the set that results from performing $Ops(C', x)$ is also $\{k\}$. This completes the proof. □

Corollary D.3. *Each update that terminates returns a response consistent with the linearization ordering.*

Proof. When an Insert that returns false at line 118 or a Delete that returns false at line 140 is first assigned an arrival point by Part 1 of Definition 2, the response value associated with it is false

Similarly, when an Insert that returns true at line 131 or a Delete that returns true at line 150 is first assigned an arrival point by Part 2 or Part 3 of Definition 2, the response value associated with it is true.

In both cases, the response value is carried up along with the operation to its arrival point at the root Node. It follows from Invariant 8.(2) that the response is consistent with performing the operations sequentially in their linearization order. □

The following invariant is trivial, since every internal Version object satisfies the invariant when it is created (at line 162) and its fields are immutable.

Invariant 9. *For every Version v that has children, $v.sum = v.left.sum + v.right.sum$.*

The following corollary is a consequence of the fact that each leaf Version containing a valid key (created at line 120) has *sum* field 1 and each leaf Version containing a sentinel key (created at line 109 or 120) has *sum* field 0.

Corollary D.4. *For every Version v, $v.sum$ is the number of leaves in the tree rooted at v that contain keys from U.*

Lemma 16. *The result returned by each query operation is consistent with the linearization.*

Proof. If we linearize any query operation (including Find) when it reads a value v from $Root.version$, it follows from Invariant 8 that the tree of Version objects rooted at v is a BST containing exactly the keys that a set would contain if all operations linearized prior to the query are performed in their linearization order. Moreover, by Theorem D.4, the *sum* fields in all nodes of the tree rooted at v are accurate for guiding order-statistic queries. Thus, the Version tree rooted at v is an immutable order-statistic tree for the set of items present in the set at the linearization point of the query. The claim follows. □