Lock-Free Augmented Trees

Panagiota Fatourou
FORTH ICS, Heraklion, Greece
University of Crete, Heraklion, Greece

Eric Ruppert
York University, Toronto, Canada

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 $\{0, \ldots, N-1\}$ 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 step complexity of the updates. As an added bonus, our technique supports arbitrary multi-point queries (such as range queries) with the same step complexity as they would have in the corresponding sequential data structure. 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 (at the cost of increasing step complexity of updates by a factor of $O(\log |S|)$).

2012 ACM Subject Classification Theory of computation → Concurrent algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases shared-memory, data structure, tree, binary search tree, augmentation, linearizable, lock-free, order statistic, snapshot

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.23

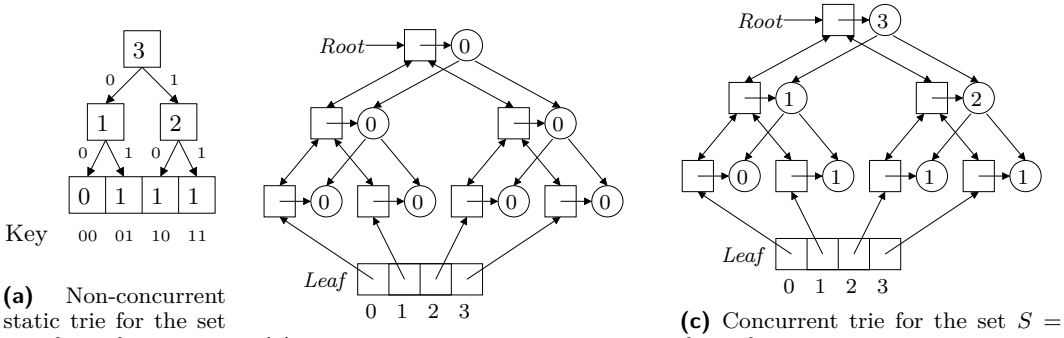
Related Version *Full Version*: `https://arxiv.org/abs/2405.10506` [24]

Funding This research was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the "Second Call for HFRI Research Projects to support Faculty Members and Researchers" (project name: PERSIST, project number: 3684) and the Natural Sciences and Engineering Research Council of Canada.

Acknowledgements We thank the anonymous reviewers for their comments.

1 Introduction

Augmentation is a fundamental technique to add functionality to 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. [18], which illustrates the technique with the most well-known example of augmenting a binary search tree (BST) so that each node stores the size of the subtree rooted at it. This adds support for many order-statistic queries, including finding the j-th smallest element in the BST or the rank of a given element, *in sub-linear time*. In a balanced BST, these queries take logarithmic time whereas a traversal of an unaugmented BST would take linear time to answer them.



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

(b) Initialization of concurrent trie.

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

Figure 1 Examples of the trie data structure when $U = \{0, 1, 2, 3\}$, where each node is augmented with a field that stores the number of elements in the subtree. Nodes are shown as squares, Versions are shown as circles containing their *sum* fields.

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 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 [18]. There are many other types of augmented trees, including one representing piecewise constant functions [12; 13, Section 4.5], measure trees [26], priority search trees [33] and segment trees [9, 10]. 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 [41] and tango trees [19]. 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 step complexity (i.e., number of steps taken) 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, in Section 3, we first illustrate the technique applied to a simple 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. To add support for order-statistic queries, each node stores the number of elements of S in its subtree. See Figure 1a for an example. Our technique mirrors this tree of nodes by a tree of Version objects, which store the mutable fields of the augmentation (see Figure 1c). 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, taking $O(\log N)$ steps in total. Our algorithm never changes fields of Version objects, including their child pointers. 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 updates. Thus, any query operation that follows pointers from the root can 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.

In Section 4, we describe how to apply the technique to a BST. This has additional complications because the structure of the tree changes as keys are inserted or deleted. We augment the lock-free BST of Ellen et al. [21], which has amortized step complexity $O(h + c)$ per operation, where h is the height of the tree and c is the point contention (i.e., the maximum number of updates running at any point in time). 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 increment the size field 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.

Our approach yields a query to find the number of keys in a lock-free tree in $O(1)$ steps. A previous, more general method for adding a size query to any dynamic set [38] is substantially more complicated, and their size queries take $\Omega(P)$ steps in a system of P processes.

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 to other recent work on providing linearizable range queries on concurrent trees [4, 7, 14, 17, 23] or more general snapshots [36, 37, 44]. Ordinarily, our snapshots can be discarded when the query completes, but they can also be used to maintain past versions of the data structure. Many of these other approaches use multiversioning and require complex schemes for unlinking old, obsolete versions from the data structure to facilitate garbage collection (e.g., [8, 45]). The simplicity of our approach avoids this.

2 Related Work

There is very little previous work on concurrent augmented trees. This year, Kokorin et al. [32] described 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. While queueing at the root, an update operation must determine whether it will ultimately succeed (by searching down to a leaf, and checking the queue of pending updates at each node along the way), so that it knows whether to modify the *size* field of nodes as it traverses them. 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.

Independently of this work, Sela and Petrank [39] recently gave a lock-based implementation of an augmented BST. Their approach is restricted to augmentations that compute aggregate functions based on an Abelian group operator (such as sum or product, but not max or min), whereas ours handles arbitrary augmentations. Their approach requires substantial coordination between concurrent operations. Updates are announced, and each query must then take into account information from all ongoing updates with timestamps earlier than its own, using a multiversioning system similar to [23, 44] that maintains version lists at each tree node. In both variants of their algorithm, queries and updates each take at least $\Omega(Ph)$ steps in the worst case when P processes access a tree of height h. Moreover, an update must hold a lock on the nodes where it is performing an insert or delete while it performs $\Omega(Ph)$ steps to update aggregated values.

Sun, Ferizovic and Blelloch [42] 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.

Independently of this work, Ko [31] used a binary trie structure to add support for predecessor queries to a lock-free data structure for a set drawn from the universe $U = \{0, 1, \ldots, N-1\}$. However, his trie design is quite different from the one we give in Section 3. It supports searches in $O(1)$ steps, while the amortized step complexity for updates and predecessor queries is $O(c^2 + \log N)$, where c is a measure of contention. Thus, searches are faster, but other operations are slower than in our trie. Moreover, Ko's approach does not appear to generalize to other order-statistic queries or other types of augmentations.

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 [5, 22, 29, 34]. 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 [28] 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.

Another technique that cooperatively builds trees bottom-up appears in Chandra, Jayanti and Tan's construction [16] 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 [11, 23, 36, 37, 44] 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 [44], 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 [20], 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 a 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.

Now, 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, respectively.
- 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 $O(1)$ steps and the other order-statistic queries can be answered in $O(\log N)$ steps. 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 [25; 30, 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. 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 the root Node's *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 3 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 the root Node's *version* field 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 [28], 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 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. To expedite access to the leaves, we use an array $Leaf[0..N-1]$, where $Leaf[k]$ points to the leaf Node for key k.

```
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: type Version                               ▷ used to store a Node's augmented data
6:     Version* left, right                   ▷ immutable pointers to children Versions
7:     int sum                                ▷ immutable sum of descendant leaves' bits
```

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

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 equal to 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. They first take a snapshot of the Version tree by reading $Root.version$ on line 41 or 47 and then execute 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 [24]; we sketch it here. 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, $U_{Root} = \langle 0, 1, \ldots, N-1 \rangle$. It can be shown by induction on the height of the Node x, that the Version tree rooted at $x.version$ is a perfect binary tree with $|U_x|$ leaves. Recall that the fields of Version objects are immutable, so the proof must only consider lines 17, 25 and 33, which create new Version objects. The induction step 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 we maintain as an invariant that,

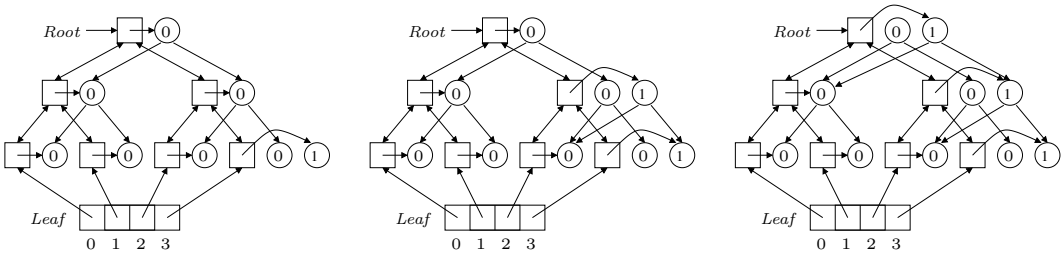
$$\text{for every internal Version } v, v.sum = v.left.sum + v.right.sum. \tag{1}$$

Since leaf Versions contain 0 or 1 (according to lines 17 and 25), $v.sum$ stores the sum of the bits stored in leaves of 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

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

13: Insert(int k) : Boolean                ▷ Add k to S; return true iff k was not already in S
14:     old ← Leaf[k].version
15:     result ← (old.sum = 0)
16:     if result then
17:         new ← new Version with sum ← 1, left ← Nil, and right ← Nil
18:         result ← CAS(Leaf[k].version, old, new)
19:     Propagate(Leaf[k].parent)
20:     return result

21: Delete(int k) : Boolean                ▷ Remove k from S; return true iff k was in S
22:     old ← Leaf[k].version
23:     result ← (old.sum = 1)
24:     if result then
25:         new ← new Version with sum ← 0, left ← Nil and right ← Nil
26:         result ← CAS(Leaf[k].version, old, new)
27:     Propagate(Leaf[k].parent)
28:     return result

29: Refresh(Node* x) : Boolean             ▷ Try to propagate information to x from its children
30:     old ← x.version
31:     v_L ← x.left.version
32:     v_R ← x.right.version
33:     new ← new Version with left ← v_L, right ← v_R, sum ← v_L.sum + v_R.sum
34:     return CAS(x.version, old, new)

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

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

46: Select(j) : int                        ▷ Return the jth smallest element in S
47:     v ← Root.version                   ▷ Start at the root
48:     i ← 1                              ▷ Keep track of breadth-first index of v in tree
49:     if v.sum < j then return Nil       ▷ No such element in S
50:     else
51:         while v.left ≠ Nil do
52:             if v.left.sum ≥ j then     ▷ Required element is in left subtree
53:                 v ← v.left
54:                 i ← 2i
55:             else                       ▷ Required element is in right subtree
56:                 v ← v.right
57:                 i ← 2i + 1
58:                 j ← j − v.left.sum     ▷ Adjust rank of element being searched for
59:         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.

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 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. C_0 is the initial configuration described in lines 8–12.

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.)*

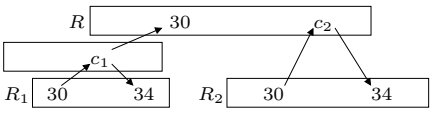


Figure 5 Calls to Refresh in proof that a double refresh successfully propagates updates to a Node from its children. The horizontal axis represents time, and boxes indicate the interval between a routine's invocation and its response. Numbers refer to line numbers in the pseudocode. An arrow $s_1 \to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

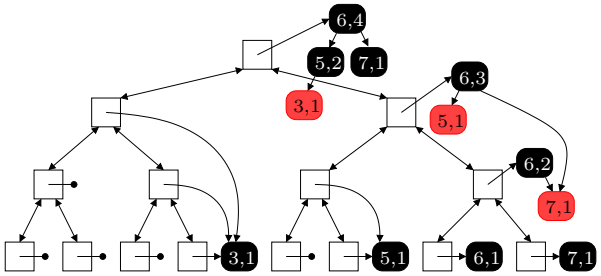


Figure 6 Augmenting the trie with red-black trees (RBTs) to speed up queries. $N = 8$ and $S = \{3, 5, 6, 7\}$. Squares are trie Nodes. Ovals are RBT nodes. Each RBT node has child pointers, and stores a key and a *size* field that represents the number of keys in the subtree. Black dots represent RBT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

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, respectively.*

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.*

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 for 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. If *op* terminates, we must also show 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 invariant (1).

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 $\mathsf{Insert}(k)$.

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 $\mathsf{Insert}(k)$ operations, which preserves the invariant. Similarly, a CAS that sets the bit to 0 is the arrival point of one or more $\mathsf{Delete}(k)$ operations, which preserves the invariant. If the step is an $\mathsf{Insert}(k)$'s read of $x.version$ when it has value 1 or a $\mathsf{Delete}(k)$'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.

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 $O(1)$ steps 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 $O(1)$ steps. A range query that returns R elements can be done in $O(R(\log \frac{N}{R} + 1))$ steps, 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 give a very pessimistic worst-case bound on the space used by objects that are still reachable. For each Node x, up to $O(\log N)$ different Versions belonging to x could be 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, 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 $Tree[2i+1]$ [30, 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

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 run in $\Theta(\log_2 N)$ steps 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 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 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 Insert(k) or 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.

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.

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 Step Complexity to $O(\log |S|)$

The step complexity of 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 [43]. To avoid destroying the smaller RBTs when performing a Join, one can use the path-copying technique of Driscoll et al. [20]. (Path copying has proved useful for a number of concurrent data structures, e.g., [3, 5, 6, 34].) For complete pseudocode, see [24].

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 $O(\log |S|)$ steps using the *size* fields of the RBT.

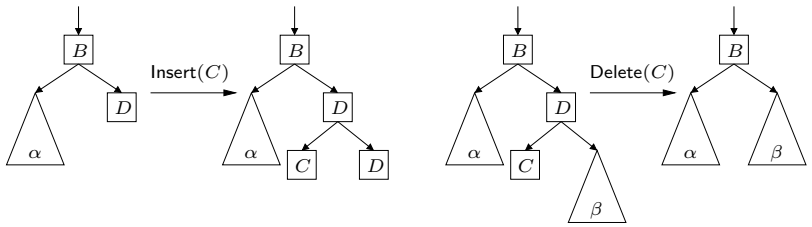


Figure 7 How updates modify a leaf-oriented BST. Here, α and β represent arbitrary subtrees.

There is a tradeoff: the step complexity of updates increases to $O(\log N \log \hat{n})$, where $\hat{n}$ denotes a bound on the maximum size the set S could have under any possible linearization of the update operations. This holds because a Join of two RBTs must be performed at each of $\log N$ Nodes of the Node tree during Propagate. 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 to 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 step and space complexity of our augmented BST depend on $|S|$ rather than $|U|$.

4.1 Basic Lock-free BST

We base our augmented BST on the lock-free BST of Ellen et al. [21], so we first give a brief overview of how this 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.

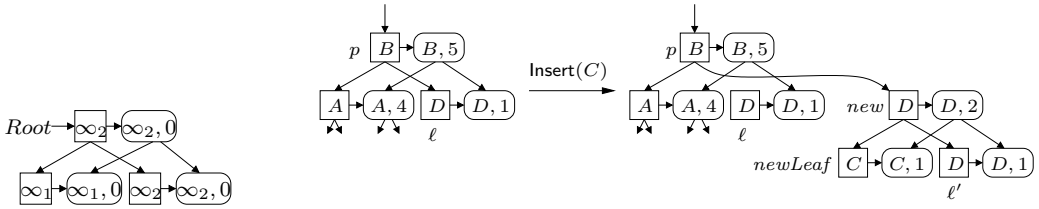
The tree satisfies the BST property at all times. We define the *search path* for a key k at some configuration C to be the path that a sequential search for k would take if it were executed without interruption in C. Searches in the lock-free BST ignore flags and simply follow child pointers until reaching a leaf. A search for k may pass through nodes that get removed by concurrent updates, but it was proved in [21] 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 [21] 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 9 on page 20 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 Appendix A.

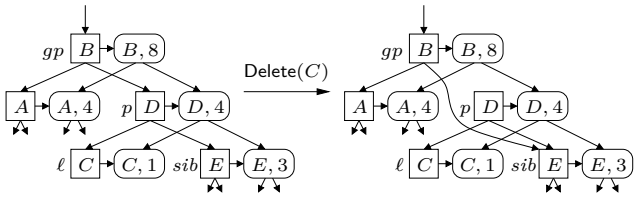
An **Insert** or **Delete** first runs the algorithm from [21] to modify the Node tree as shown in Figure 7. Figures 8b and 8c show the effects of the modification when Versions are also present. Then, the update calls Propagate to modify the *sum* fields of the Versions of all Nodes along the path from 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 these 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 store 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 these Nodes 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.



(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.

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. (It may be that this re-reading could be avoided, but it simplifies the proof of correctness.)

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. 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 also makes searches wait-free, unlike the original BST of [21], 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 [23] (later generalized by [44] 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 [23], 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.*

Pseudocode for the augmented BST appears in Appendix A and a sketch of the correctness proof is in Appendix B. For a detailed correctness proof, see [24].

4.3 Complexity

The amortized step complexity per operation on the unaugmented BST is $O(h + c)$, where h is the height of the Node tree and c is point contention [21]. 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 step complexity 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 step complexity of the update in the original algorithm of [21]. 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 step complexity per update operation remains $O(h + c)$. Since queries begin by taking a snapshot of the Version tree, queries are wait-free and take the same number of steps that they would in the sequential setting. For example, searches and order-statistic queries take $O(h)$ steps.

4.4 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 number of steps as in a sequential augmented balanced BST. For example, order-statistic queries can all be answered in $O(\log n)$ steps where n is the size of the set. This does, however, increase the amortized step complexity 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 can provide 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 [21], we believe our technique could also be applied to the similar lock-free BST design of Natarajan, Ramachandran and Mittal [35] or other concurrent trees. It would be interesting to apply it to a node-oriented tree such as [27], a balanced tree such as the lock-free chromatic BST of [15] or to a self-balancing concurrent tree such as the CB Tree [2]. In particular, the latter two 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., [35]).

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 [41] and segment trees [9, 10]. Shafiei [40] 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. doi:10.1145/225058.225271.
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. doi:10.1007/S00446-014-0229-0.
3. Vitaly Aksenov, Trevor Brown, Alexander Fedorov, and Ilya Kokorin. Poster: Unexpected scaling in path copying trees. In *Proc. 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 438–440, 2023. doi:10.1145/3572848.3577512.
4. 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. doi:10.1145/3178487.3178489.
5. 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. doi:10.4230/LIPICS.OPODIS.2023.17.
6. Benyamin Bashari and Philipp Woelfel. An efficient adaptive partial snapshot implementation. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 545–555, 2021. doi:10.1145/3465084.3467939.
7. 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):16:1–16:28, June 2020. doi:10.1145/3399718.
8. 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. doi:10.4230/LIPICS.DISC.2021.12.
9. J. L. Bentley. Solutions to Klee's rectangle problems. Technical report, Carnegie-Mellon University, Pittsburgh, PA, 1977.
10. Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.
11. 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. doi:10.1145/3627535.3638501.
12. 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. doi:10.1016/S0925-7721(02)00128-1.
13. Peter Brass. *Advanced Data Structures*. Cambridge University Press, 2008.
14. 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. doi:10.1007/978-3-642-35476-2_3.
15. 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. doi:10.1145/2555243.2555267.
16. 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. doi:10.1145/277697.277753.
17. 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. doi:10.1145/3007748.3007771.
18. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 17, pages 480–496. MIT Press, fourth edition, 2022.

19 Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic optimality–almost. *SIAM Journal on Computing*, 37(1):240–251, 2007. doi:10.1137/S0097539705447347.

20 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. doi:10.1016/0022-0000(89)90034-2.

21 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. Full version available online from https://users.ics.forth.gr/~faturu/BSTproof.pdf. doi:10.1145/2611462.2611486.

22 Panagiota Fatourou and Nikolaos D. Kallimanis. The RedBlue family of universal constructions. *Distributed Computing*, 33(6):485–513, 2020. doi:10.1007/S00446-020-00370-7.

23 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. doi:10.1145/3323165.3323197.

24 Panagiota Fatourou and Eric Ruppert. Lock-free augmented trees. Full version available from https://arxiv.org/abs/2405.10506, May 2024. doi:10.48550/arXiv.2405.10506.

25 Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960. doi:10.1145/367390.367400.

26 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. doi:10.1016/0734-189X(83)90111-1.

27 Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 161–171, 2012. doi:10.1145/2312005.2312036.

28 Prasad Jayanti. f-arrays: implementation and applications. In *Proc. 21st ACM Symposium on Principles of Distributed Computing*, pages 270–279. ACM, 2002. doi:10.1145/571825.571875.

29 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. doi:10.1007/11590156_33.

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

31 Jeremy Ko. A lock-free binary trie. In *Proc. 44th IEEE International Conference on Distributed Computing Systems*, 2024. To appear. Preliminary version available from https://arxiv.org/abs/2405.06208. doi:10.48550/arXiv.2405.06208.

32 Ilya Kokorin, Victor Yudov, Vitaly Aksenov, and Dan Alistarh. Wait-free trees with asymptotically-efficient range queries. In *Proc. IEEE International Parallel and Distributed Processing Symposium*, pages 169–179, 2024. doi:10.1109/IPDPS57955.2024.00023.

33 Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985. doi:10.1137/0214021.

34 Hossein Naderibeni and Eric Ruppert. A wait-free queue with polylogarithmic step complexity. *Distributed Computing*, 2024. Published online August, 2024. doi:10.1007/s00446-024-00471-7.

35 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. doi:10.1145/3391438.

36 Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 368–384, 2022. doi:10.1145/3503221.3508412.

37 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. doi:10.1145/2145816.2145836.

38 Gal Sela and Erez Petrank. Concurrent size. *Proc. of the ACM on Programming Languages*, 6(OOPSLA2):345–372, 2022. doi:10.1145/3563300.

39 Gal Sela and Erez Petrank. Concurrent aggregate queries. Manuscript available from https://arxiv.org/abs/2405.07434, May 2024. doi:10.48550/arXiv.2405.07434.

40 Niloufar Shafiei. Non-blocking Patricia tries with replace operations. *Distributed Computing*, 32(5):423–442, 2019. doi:10.1007/S00446-019-00347-1.

41 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. doi:10.1145/800076.802464.

42 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. doi:10.1145/3178487.3178509.

43 Robert Endre Tarjan. *Data Structures and Network Algorithms*, chapter 4.2, pages 45–57. SIAM, Philadelphia, USA, 1983. doi:10.1137/1.9781611970265.

44 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. doi:10.1145/3437801.3441602.

45 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. doi:10.1145/3572848.3577508.

A Pseudocode for Lock-Free Augmented BST

Here, we give more details about how to augment the lock-free BST of Ellen et al. [21]. Type definitions are given in Figure 9. High-level pseudocode for Insert and Delete is in Figure 10. These are mostly the same as in [21], 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 [21] for the detailed pseudocode. The new routines for handling Versions and example queries are in Figure 11.

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

```
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 augmentation

105: type Version                       ▷ used to store a Node's augmented data
106:   | U key                          ▷ immutable key of Node this Version belongs to
107:   | Version* left, right           ▷ immutable pointers to children Versions
108:   | int sum                        ▷ immutable sum of descendant leaves' bits
```

Figure 9 Object types used in lock-free augmented BST data structure.

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.

Refresh(x) is similar to the routine in Figure 3. Because the structure 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 161 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 165. Instead, an update operation stores 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 [21], where Find operations may starve.

B Sketch of Proof of Correctness for Augmented BST

A detailed proof of linearizability for the augmented BST is in [24]. We sketch it here. As in Section 3.2, we define arrival points of update operations at a Node to indicate when the updates have been propagated to that Node. We linearize updates at their arrival point at the root, and queries when they obtain a snapshot of the Version tree by reading $Root.version$. As in [21], sentinel Nodes as shown in Figure 8a ensure that the root Node never changes.

We again use two main claims: (1) every update operation has an arrival point at the root during the operation, and (2) 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. Claim (1) implies the linearization respects the real-time order of operations. Applying Claim (2) to the root shows that queries return results consistent with the linearization.

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

```
109: Initialize the data structure as shown in Figure 8a, where Root is a shared pointer

110: Insert(Key k) : Boolean
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 then               ▷ another update caused failure
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: Delete(Key k) : Boolean
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 then               ▷ another update caused failure
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 10 Pseudocode for augmented BST. The code for updates is given at a high level. For details, see [21]. Changes to Insert and Delete to support augmentation is shaded.

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

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

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

174: Select(j) : U                             ▷ Returns set's jth smallest element
175:     v ← Root.version
176:     if j > v.sum then                     ▷ Return Nil if size of set is less than j
177:         return Nil
178:     repeat                                ▷ Loop invariant: desired element is jth in v's subtree
179:         if j ≤ v.left.sum then
180:             v ← v.left
181:         else
182:             j ← j − v.left.sum
183:             v ← v.right
184:     until v is a leaf
185:     return v.key

186: Size : int                                ▷ Returns number of elements in the set
187:     return Root.version.sum
```

Figure 11 Pseudocode augmented BST, continued. We include Find, Select and Size as three examples of queries that use the augmentation.

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 [21], 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.

We describe the arrival points (which in turn defines the linearization) and sketch some of the key arguments about them. 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 [21] 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. 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, [21] proved 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. Deletes that return false are handled similarly.

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 those updates.

We use the definition of arrival points to prove that each update operation's arrival point at the root is between the update's invocation and response. 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 [21] 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.

The main invariant says 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. Proving this 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. Moreover, the boolean responses these operations will return are consistent with this sequential ordering. Applying this invariant to the root shows 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 $x.version$, if updates have changed the Node tree since $x.version$ was stored. However, 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. Hence, queries' results are consistent with the linearization.