# Practical Concurrent Binary Search Trees via Logical Ordering

Dana Drachsler
Technion
ddana@cs.technion.ac.il

Martin Vechev
ETH Zurich
martin.vechev@inf.ethz.ch

Eran Yahav
Technion
yahave@cs.technion.ac.il

## Abstract

We present practical, concurrent binary search tree (BST) algorithms that explicitly maintain *logical ordering* information in the data structure, permitting clean separation from its physical tree layout. We capture logical ordering using intervals, with the property that an item belongs to the tree if and only if the item is an endpoint of some interval. We are thus able to construct efficient, synchronization-free and intuitive lookup operations.

We present (i) a concurrent non-balanced BST with a lock-free lookup, and (ii) a concurrent AVL tree with a lock-free lookup that requires no synchronization with any mutating operations, including balancing operations. Our algorithms apply on-time deletion; that is, every request for removal of a node, results in its immediate removal from the tree. This new feature did not exist in previous concurrent internal tree algorithms.

We implemented our concurrent BST algorithms and evaluated them against several state-of-the-art concurrent tree algorithms. Our experimental results show that our algorithms with lock-free contains and on-time deletion are practical and often comparable to the state-of-the-art.

**Categories and Subject Descriptors** D.1.3 [*Concurrent Programming*]; D.3.3 [*Programming Languages*]: Language Constructs and Features - Concurrent programming structures; E.1 [*Data Structures*]: Trees

**Keywords** Concurrency, Search Trees

## 1. Introduction

Concurrent data structures are a fundamental building block for leveraging modern multi-core processors. Recent years have seen rising interest in scalable and efficient concurrent algorithms for data structures. In this paper, we focus on concurrent algorithms for binary search trees (BST), important in a variety of applications. A BST data structure supports the operations `insert`, `remove`, and `contains` with their standard meaning. Any correct BST algorithm must preserve two invariants: (i) the BST does not contain duplicate keys, and (ii) the tree follows the standard BST structural layout.

A key challenge in designing correct and efficient concurrent BST algorithms is to devise a scalable design for the *lookup* operation. This operation is invoked by all three operations to check whether a given element exists in the tree. Because the lookup operation can execute concurrently with other mutating operations, the physical location of an element in the tree might change while the lookup operation is executing. A particularly tricky case arises when the lookup operation *does not find* the element it is looking for. The algorithm then needs to decide whether to continue searching for the element elsewhere in the tree, or to conclude that it is not present. It can be difficult to decide which of these decisions is correct.

To illustrate the difficulty of correctly performing lookups in a BST while the tree is being mutated by concurrent operations, consider an interleaving of two concurrent operations. Figure 1(a) shows an initial tree with two threads operating on it. First, `T1` performs a `contains(7)` operation on the tree, reaches node 9 (one step away from reaching the target node 7), and is suspended. The resulting state is shown in Figure 1(b). Then, `T2` performs the entire `remove(3)` operation, which results in swapping nodes 3 and 7, and removing node 3 from the tree. The resulting tree is shown in Figure 1(c). Finally, the suspended thread `T1` is resumed and fails to find node 7, even though it is in the tree.

To address this challenge, some concurrent trees maintain all values in the leaves [7, 9, 11, 15], thus never changing the location of an element. Others provide no support for remove operations [4], and yet others use some form of notification such as version numbers [8] or node marking [10, 13] to detect concurrent updates during lookup. While these approaches differ on the exact details of how they synchronize, they all base their synchronization on the *tree layout*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*PPoPP '14*, February 15–19, 2014, Orlando, Florida, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2656-8/14/02...$15.00.
http://dx.doi.org/10.1145/2555243.2555269

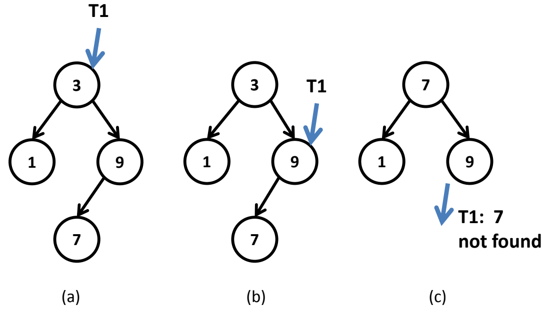


Figure 1: Concurrent lookup with mutating operations for two threads, `T1: contains(7)` ∥ `T2: remove(3)`. The initial tree is shown in (a); the tree after T1 reaches node 9 and before 3 is removed in (b); and the tree after 3 is removed in (c) (3 is swapped with 7 and removed). Here, T1 fails to find 7 even though it is in the tree at the time T1 returns false.

In this paper, we present a fundamentally different approach to constructing concurrent BST algorithms, one which leads to a simple and intuitive lookup operation that also enjoys desirable progress guarantees (i.e., lock-freedom). Below, we informally explain the idea by means of example. We provide an elaborate description of the algorithms in Section 4.

**Key Idea: Leveraging Logical Ordering** At the specification level, a binary search tree implements a set of elements such that a total order can be established between them. For instance, consider again the tree of Figure 1(a). This BST represents the set of integers $\{1, 3, 7, 9\}$, where elements can be ordered by the value of their keys: $1 < 3 < 7 < 9$. To avoid edge cases, we always add designated sentinel keys $-\infty$ and $\infty$ to any set, which for our example yields the set $\{-\infty, 1, 3, 7, 9, \infty\}$.

The *logical ordering* of elements can be *maintained explicitly* in the data structure, and this important property enables us to find the *successor* and *predecessor* of a node without traversing pointers along the tree layout. Because logical ordering is stable under layout manipulations (such as balancing), lookup operations can proceed concurrently with operations that mutate the layout.

Explicitly maintaining ordering information represents a different space-time-synchronization tradeoff than in other concurrent BSTs. In particular, we trade off the time for performing a lookup through traversal of layout pointers with the space required to explicitly maintain the ordering information (three pointers per node) and the time required to update the ordering information when it changes. This introduces some overhead for each individual operation, but allows lookup operations to operate without synchronization over the tree layout.

**Lookups using Ordering** To see how ordering is leveraged, consider again our running example and the tree of Figure 1(c). This tree represents the set of elements $\{-\infty, 1, 7, 9, \infty\}$. Previously, `contains` was looking for key 7, which it missed due to concurrent interference caused by the `remove(3)` operation. However, when the tree is equipped with ordering, after `contains` reaches node 9 and learns that its left child is `null`, it looks up the *predecessor* of 9, finding that $7 < 9$ is in the ordered set, meaning that 7 is in the tree, and thus `contains(7)` correctly returns `true`.

It is natural to view the strict total order relation $<$ maintained by our algorithm as the pairs of elements in its transitive reduction. For our example, these are the pairs: $(-\infty, 1), (1, 7), (7, 9), (9, \infty)$. These pairs can be viewed as intervals where a key belongs to the set if it is an endpoint of some interval and does not belong to the set otherwise. Our algorithm can be viewed as explicitly maintaining these intervals and using them to answer lookup queries and to synchronize operations. That is, in our trees, each node keeps its successor endpoint and its predecessor endpoint (which are unique); the synchronization may also be performed on these endpoints as opposed to only on the tree layout as in previous algorithms.

Note that the logical ordering is independent of the physical tree layout, and a number of layouts represent the same ordered set. The conceptual separation between ordering of elements and the physical tree layout enables us to design concurrent algorithms where the balancing of the tree (which rearranges the physical layout) is independent of, and requires no synchronization with, the lookup queries, which use ordering information.

Note that in this example, `remove` changed the physical layout of the tree, yet this operation need not be synchronized with the lookup query issued by `contains`.

**Main Contributions** The contributions of this paper are:

- Novel concurrent BST algorithms (both non-balanced and AVL), which leverage the idea of logical key ordering to obtain simple, intuitive, and robust lookup operations that also enjoy strong progress guarantees. By maintaining logical ordering we can cleanly separate operations that alter the tree layout (e.g., balancing) from those that require information about the presence of elements (e.g., lookup). Thus, balancing operations and lookup queries can proceed without synchronization.
- An implementation and evaluation of our algorithms – a non-balanced tree and an AVL tree. We evaluated our implementation with various loads and compared it to several state-of-the-art BSTs from the literature. The experimental results indicate that our trees (providing lock-free lookups) are practical and often comparable to state-of-the-art algorithms.

## 2. Background

A binary search tree (BST) is a data structure that consists of nodes, each associated with a unique *key* and having a parent, a *left* child and a *right* child. For every node, nodes in its left sub-tree have keys smaller than the node's key, and nodes in its right sub-tree have keys greater than the node's key. A BST supports three operations: `insert(k)`, `remove(k)` and `contains(k)`.

The `insert(k)` operation inserts $k$ to the tree if $k$ is not present. If $k$ was inserted, the insertion is considered *successful*. In a successful insertion, the correct parent is located and the new node is connected to it as a child.

The `remove(k)` operation removes $k$ from the tree if $k$ is present. If $k$ was removed, the removal is considered *successful*. Denote the node with key $k$ by $N_k$. A successful removal has three possible scenarios:

- $N_k$ is a leaf - remove it by updating $N_k$'s parent to point to `null`.
- $N_k$ has a single child - remove it by updating $N_k$'s parent to point to $N_k$'s child.
- $N_k$ has two children - the node is removed in three steps: (i) locate $N_k$'s successor node, denote it by $N_s$, (ii) remove $N_s$ from its location, and (iii) update $N_s$ to appear in $N_k$'s location.
  $N_k$'s *successor*, $N_s$, is the left-most child of its right sub-tree, and it is guaranteed to have at most one child.

**Removing a Node with Two Children (2C-removal)** Because removing a node with two children involves $N_k$'s successor, this type of removal often updates nodes which *are not adjacent, and many pointers may be traversed until $N_s$ is reached*. Common implementations thus attempt to avoid 2C-removal. Some implementations provide external trees [7, 9, 11, 15] where the values are kept only at the leaves, and inner nodes serve only as routing nodes. Inner nodes cannot be requested (by the user) to be removed and thus 2C-removal never occurs. Other implementations do not support the remove operation [4]. Recent works cope with this challenge by not *physically* removing nodes in the case of 2C-removal [8, 10]. Instead, they mark these nodes as *logically* removed, and remove them only when these nodes have a single child. Avoiding physical 2C-removal can result in a large number of "zombie" nodes in a tree where a node has been logically deleted but cannot be removed. This has implications both for the space consumed by the tree, and for the extension of search paths. The authors of [13] never physically remove nodes that have no children. They apply 2C-removal by copying the successor's key to the removed node, and removing the successor if it has a child.

**AVL Tree** Balanced trees provide logarithmic worst-case time complexity. This guarantee becomes crucial as the tree grows or when the values for the operations are not picked uniformly. In balanced trees, the tree maintains a balancing invariant. In AVL trees [1] it holds that for each node the difference between the heights of its left sub-tree and its right sub-tree is less than two. This difference is often called the *balance factor*. Applying a mutating operation to the tree may result in breaking the invariant for multiple nodes. Upon a violation of the invariant, *rotation*s are applied. To detect the violation, and to decide which type of rotation is required, nodes save local information. Traditionally, nodes keep their balance factor. However, they can keep the heights of their left and right sub-trees instead. We use the latter approach since it allows for more concurrency.

Another balanced internal tree is the red-black tree [2]. Pfaff showed that in a sequential setting, there is no clear winner between the two trees [16]. However, AVL trees typically have shorter paths than red-black trees.

**Relaxed Balanced Tree** Maintaining the AVL invariant in a concurrent setting induces severe bottlenecks. This is because during rotations, the sub-trees of the violated nodes must not be updated. Thus, it is beneficial to consider a *relaxed balanced tree*, where the rebalance operations are decoupled from the mutating operations and may be delayed. Bougé et al. [6] proved that a concurrent tree which applies the AVL rotations using the local information of the node and its children achieves a relaxed balanced tree, guaranteed to be strictly balanced when there are no ongoing mutating operations. This should be contrasted with approaches such as [10] that perform rotations in a separate thread and thus can only guarantee some form of "eventual balance."

We follow Bougé et al.'s method and achieve a relaxed balanced tree which is strictly balanced at a quiescent state.

## 3. Concurrent Trees with Logical Ordering

In this section, we present the principles underlying our concurrent tree algorithms. In Section 3.1, we explain how we explicitly maintain logical ordering information in the BST. In Section 3.2, we show how logical ordering enables lock-free lookup queries. In Section 3.3, we show how it enables on-time 2C-removal. Finally, in Section 3.4, we provide the highlights of our synchronization method.

We deliberately present the operations without describing how they are synchronized. We provide synchronization details in Section 3.4 and in Section 4. To simplify presentation, we present our tree as implementing a set. However, our actual implementation and evaluation use a more general implementation of a map.

### 3.1 Adding Explicit Predecessors and Successors

We augment the tree to support *predecessor* and *successor* queries in $O(1)$. To this end, we extend each node with predecessor and successor pointers, denoted `pred` and `succ`. These pointers allow us to operate separately on the *tree layout* and the *tree ordering*, which is captured by the predecessor and successor relations. Roughly speaking, we implement the set semantics using the ordering, and achieve the time complexity using the tree layout.

The challenges introduced by the `remove` operation are resolved to local mutations with respect to the tree ordering, without a need to consult the tree layout. Specifically, the search for the successor in the case of 2C-removal can now be resolved by following a single pointer.

**Maintaining the Predecessor and Successor** We now describe how `pred` and `succ` are maintained. In the following, we use $N_k$ to denote a node with key $k$.

**Insert** In a BST, a new node, $N$, is inserted as a child of either its predecessor, $p$, or its successor, $s$. Thus, $N$ can access and set $p$ and $s$ using its parent's `pred` and `succ` pointers. Then, $p$'s `succ` and $s$'s `pred` are updated to point to $N$. For example, consider the tree in Figure 2(a). On a call to `insert(7)`, the parent will be $N_9$, which is $N_7$'s successor. $N_7$'s predecessor is $N_9$'s old predecessor, $N_3$. During the insert, $N_3$'s `succ` and $N_9$'s `pred` are updated to point to $N_7$.

**Remove** Upon a removal of a node $N$, $N$'s predecessor and $N$'s successor are set to point to each other. This update occurs regardless of how many children $N$ has. $N$'s predecessor and successor are accessed via $N$'s `pred` and `succ`. This is illustrated by the tree in Figure 2(b). Upon applying `remove(3)`, the tree is updated to the tree of Figure 2(c). The removal of $N_3$ from the tree ordering is done by updating $N_1$'s `succ` (i.e., $N_3$'s predecessor) to point to $N_7$ (i.e., $N_3$'s successor), and by updating $N_7$'s `pred` to point to $N_1$.

### 3.2 Lock-Free Lookup Queries using Logical Ordering

In a sequential BST, whether a given value is present in the tree can be determined simply by following the child pointers until reaching a node with this value, or until reaching the end of a path in the tree. As shown in Section 1, in a concurrent setting, such traversal may lead to incorrect results due to concurrent mutations of the tree. To avoid this problem, we rely on the following observation: to determine whether $k$ is present in the tree, it is enough to have two keys, $k_1$, $k_2$, such that (i) $k_1$ and $k_2$ are in the tree, (ii) for every $\tilde{k} \in (k_1, k_2)$, $\tilde{k}$ is not in the tree, and (iii) $k \in [k_1, k_2]$.

Using the logical ordering and the above observation, we can determine whether a key $k$ is in the tree as follows:

- If $k$ was found during traversal, then $k$ is in the tree.
- If $k$ was not found, then we must find two keys, $k_1$ and $k_2$, that are in the tree and such that $k \in (k_1, k_2)$. The search for $k$ terminates when it reaches a node of value $\tilde{k}$ that lay at the end of the scanned path. If there are no concurrent updates, $\tilde{k}$ is either $k$'s predecessor or successor; thus, one of the following holds: (i) $k \in (\texttt{pred}(\tilde{k}), \tilde{k})$, or (ii) $k \in (\tilde{k}, \texttt{succ}(\tilde{k}))$. In the presence of concurrent updates, $k_1$ and $k_2$ must be found, which will be done via the `pred` and `succ` pointers.

To illustrate this, consider the tree of Figure 2(a). Suppose that thread $T$ is executing `contains(7)`, and has reached the end of the path, in which $N_9$ is the last node. Then, $T$ reads $N_9$'s `pred` and discovers that $N_3$ is $N_9$'s predecessor. Thus, $T$ infers that 7 does not appear in the tree.

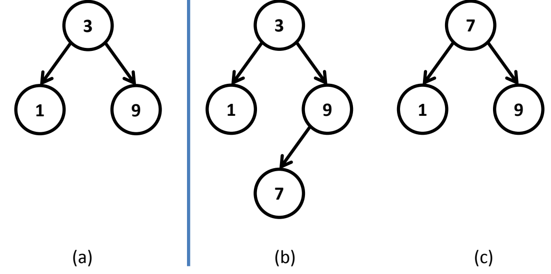


Figure 2: Lookup queries using intervals.

In contrast, suppose that $T$ is executing `contains(7)` on the tree of Figure 2(b), and that $T$ was suspended when it reached $N_9$. Then, another thread, $T'$, applies `remove(3)`, which results in the tree that appears in Figure 2(c). Next, $T$ resumes and discovers it has reached a leaf. Then, $T$ checks $N_9$'s `pred`, discovers $N_7$, and infers that 7 is in the tree. Note that $T$ is not allowed to infer that 7 is not in the tree because 7 was present in the tree throughout $T$'s traversal.

### 3.3 Removal using Logical Ordering

In a classic BST, a `remove` operation might have to scan an unbounded number of nodes to find the removed node's successor. This is because the search for the successor is done by searching for the left-most node in the removed node's right sub-tree. The height of the sub-tree is not bounded; thus, the search may scan an unbounded number of nodes. In concurrent algorithms, this means either blocking for a longer period or risking possible exposure to more concurrent updates. The latter might result in failure to find the key, even though it is present, and thus the search will incorrectly return a false result. In contrast, having the `succ` pointers allows the `remove` to find the successor using a single read.

### 3.4 Synchronization over the Logical Ordering

Our synchronization is based on locks, where each node can be locked in two separate layouts:

- The tree ordering layout
- The tree physical layout

Each update operation is applied in four steps:

1. Acquire ordering layout locks.
2. Acquire physical layout locks.
3. Update the ordering layout and release ordering locks.
4. Update the physical layout and release physical locks.

Locking the node in the tree's physical layout prevents concurrent updates to the node's physical layout information, that is, the node's children, parent and balance information. The synchronization over the physical layout is similar to previous approaches; see Section 4 for details.

Locking the node in the tree ordering layout prevents concurrent updates to the *intervals*. As described in Section 1, we capture the logical ordering via a set of intervals: $\{(p, s) \mid N_p, N_s \in \mathit{tree} \wedge \forall k \in (p, s), N_k \notin \mathit{tree}\}$. With each interval $(p, s)$ we associate a lock. The intervals can be split, upon insertion, or merged, upon removal. Upon split-

```
class Node<K> {
    final K key;
    volatile boolean mark;
    volatile Node<K> left, right;
    volatile Node<K> parent;
    volatile int leftHeight, rightHeight;
    Lock treeLock;
    volatile Node<K> succ, pred;
    Lock succLock;
}
```

Figure 3: Fields of the node data structure.

ting an interval $(p, s)$ to $(p, k)$ and $(k, s)$, we acquire $(p, s)$'s lock. Upon merging two intervals $(p, k), (k, s)$ to $(p, s)$, we acquire $(p, k)$'s and $(k, s)$'s locks.

Technically, the intervals are captured via the `pred` and `succ` pointers, and $(p, s)$'s lock is kept in $N_p$. This lock can be reached from $N_s$ by accessing $N_p$ (via $N_s$'s `pred`).

An update of $(p, s)$ to $(p, k), (k, s)$ is applied as follows:

1. $(p, s)$'s lock is acquired.
2. $N_k$ is created with `pred` set to $N_p$ and `succ` set to $N_s$.
3. $N_p$'s `succ` and $N_s$'s `pred` are updated to $N_k$.

Note that even though $(p, s)$'s lock is kept in $N_p$, it prevents concurrent updates to $N_s$'s `pred` as well.

For example, consider the tree in Figure 2(a). On a call to `insert(7)`, the interval $(3, 9)$ is locked (via $N_3$), after which, `pred` and `succ` are updated as described in Section 3.1. This results in the intervals $(3, 7), (7, 9)$.

An update of $(p, k), (k, s)$ to $(p, s)$ is applied as follows:

1. $(p, k)$'s and $(k, s)$'s locks are acquired.
2. $N_k$ is marked as removed (using a designated field). This also serves as an indication that $(k, s)$ is removed.
3. $N_p$'s `succ` is set to $N_s$ and $N_s$'s `pred` is set to $N_p$.

To illustrate this, consider the tree in Figure 2(b). Upon applying `remove(3)`, the intervals $(1, 3), (3, 7)$ are locked (via $N_1$ and $N_3$). Then, $N_3$ is marked as removed, which indicates that $(3, 7)$ is also removed. Next, `pred` and `succ` are updated as described in Section 3.1, which results in the interval $(1, 7)$. As an aside, we mention that only after these updates is the tree updated to the tree of Figure 2(c).

# 4. The Algorithm

In this section, we present the details of our implementation. The code is available at github.com/logicalordering/trees. We first present the basic data structure, and then describe the operations.

## 4.1 The Node Data Structure

Figure 3 shows the fields of a `Node` data structure in our tree. The `key` field is immutable; all other fields are mutable. Our node contains a `mark` field used to indicate whether the node was removed from the tree; this field is initially set to `false`. It maintains the heights of its sub-trees in `leftHeight` and `rightHeight` such that the AVL invariant can be checked and maintained. In addition to the fields of a standard BST node, it contains pointers to the predecessor, `pred`, and successor, `succ`, of the node. It also contains two locks:

**Algorithm 1:** search(k)

```
1 node = root
2 while true do
3     currKey = node.key
4     if currKey == k then return node
5     child = currKey < k ? node.right : node.left
6     if child == null then return node
7     node = child
8 end
```

- A `treeLock`: protects the tree's physical layout fields, `left`, `right`, `parent`, `leftHeight` and `rightHeight`.
- A `succLock`: protects the logical ordering layout fields, (i) the `succ` field, and (ii) the `pred` field of *the node pointed by `succ`*.

  That is, for every node $N$, $N$'s `succLock` protects the interval $(N$,`succ`$(N))$.

**The Initial Tree** The tree is initialized with two nodes, $N_{-\infty}$ and $N_\infty$, with keys $-\infty$ and $\infty$, which are each other's predecessor and successor. The root is $N_\infty$, and $N_{-\infty}$ is reachable only via the logical ordering layout (via the `pred` pointer).

## 4.2 The Search($k$) and Contains($k$) Operations

The `search` operation, shown in Algorithm 1, is the basic lookup operation that all other operations use. The `search` traverses once down the tree until the desired key is found or the end of the path is reached. It is oblivious to location updates caused by removals or rotations; thus, it may stray from its initial path. The `search` operation does not acquire any locks, and does not restart. Part of the beauty of our approach is the simplicity of this operation and the `contains` operation that follows.

The `contains` operation, which appears in Algorithm 2, begins by calling `search(`$k$`)`. If the returned node has key $k$, then `contains` returns `true` if the node's `mark` field is `false`, and `false` otherwise (as a marked node is logically removed). If the node has a key different than $k$, then two nodes are required to determine whether $k$ is in the tree, $N_{k_1}$ and $N_{k_2}$, that hold $k \in (k_1, k_2]$ and `succ`$(N_{k_1}) = N_{k_2}$. If $k_2 = k$ (i.e., $k$ was found), the decision is made as described before. If $k_2 > k$, it can be concluded that $k$ is not in the tree (actually it can be concluded that at *some* moment after the `contains` operation has started, $k$ was not in the tree; see Section 5 for further details). To obtain $N_{k_1}$ and $N_{k_2}$, the `contains` operation uses the node returned by `search`. It then traverses using the `pred` field until reaching the first node whose key is not greater than $k$. Once discovered, it scans nodes using the `succ` field, until reaching a node with a key equal to or greater than $k$. The last iteration of this loop reads $N_{k_1}$'s `succ` field and saves $N_{k_2}$ as required. Note that `contains` does not acquire any locks, and does not restart. Also note that after calling `search`, it only traverses the `pred` and `succ` fields.

**Algorithm 2:** contains(k)

```
1 node = search(k)
2 while node.key > k do  node = node.pred
3 while node.key < k do  node = node.succ
4 return (node.key == k && !(node.mark))
```

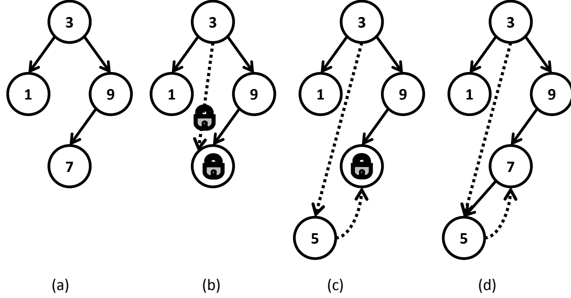


Figure 4: An example of the `insert(5)` operation. The initial tree is shown in (a); in (b) the tree after acquiring the `succLock` of 3 (5's predecessor), and after choosing and acquiring the `treeLock` of the parent, 7; the tree after updating the logical ordering layout in (c); and the tree after updating the tree's physical layout in (d).

### 4.3 The Insert($k$) Operation

The `insert` operation, shown in Algorithm 3, begins by calling `search(`$k$`)` and then operates on the returned $node$. Provided that no changes were applied to the tree from the beginning of the operation, $node$ is one of the following: (i) A node with key $k$, (ii) $k$'s predecessor in the tree, or (iii) $k$'s successor in the tree. At this point $k$'s predecessor is stored in $p$, and there is an attempt to lock $p$'s `succLock`. After locking, it needs to be checked that $k \in (p$'s key,`succ`$(p)$'s key$]$ and that the interval is not marked as removed. The latter is checked by confirming that $p$'s `mark` field is `false`. If the interval is not removed, then it is guaranteed also that both $p$ and `succ`$(p)$ are not logically removed: $p$ is not removed, because it shares the same `mark` field with the interval, and `succ`$(p)$ is not removed, because at the end of every operation, intervals that were not removed do not include removed nodes as edge points.

If the validation succeeds, the insert takes place; otherwise, it restarts. If `succ`$(p)$'s key is greater than $k$, then this is a successful insertion, and $k$ is inserted to the tree. Otherwise, `succ`$(p)$'s key is equal to $k$, and this is an unsuccessful insertion, in which case the tree remains unchanged.

A successful insertion begins with creating the new node with key $k$. Then, the parent is determined and locked via the `chooseParent` operation. Then, the update is applied, first by updating the logical ordering (lines 13–17) and then by updating the tree's physical layout (line 18). After that, balancing is applied, if needed. An example of the `insert` operation appears in Figure 4.

**Algorithm 3:** Insert(k)

```
1  while true do
2      node = search(k)
3      p = node.key > k ? node.pred : node
4      lock(p.succLock)
5      s = p.succ
6      if k ∈ (p.key,s.key] && !p.mark then
7          if s.key == k then        // Unsuccessful insert
8              unlock(p.succLock)
9              return false
10         end
11         newNode = new Node(k) // Successful insert
12         parent = chooseParent(p, s, node)
13         newNode.succ = s
14         newNode.pred = p
15         s.pred = newNode
16         p.succ = newNode
17         unlock(p.succLock)
18         insertToTree(parent, newNode)
19         return true
20     end
21     unlock(p.succLock)          // Validation failed,
        restart
22 end
```

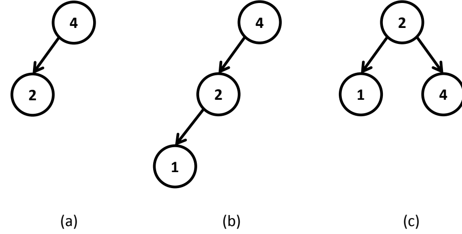


Figure 5: Choosing the correct parent for two concurrent threads, `T1: insert(1)` ‖ `T2: insert(3)`. The initial tree is shown in (a); the tree after T1 inserts 1 and T2 observes that 2 should be 3's parent in (b); the tree after rotation and T2's discovery that 4 should be the parent in (c).

**Choosing the Correct Parent** Whereas in a sequential BST, the node that was returned from `search` is the parent, in a concurrent AVL tree this is not necessarily the case. Consider the example of Figure 5, where two threads, $T_1$ and $T_2$, execute concurrently `insert(1)` and `insert(3)` to the tree of Figure 5(a). Both threads acquire the required `succLocks`: $T_1$ locks $-\infty$'s `succLock` and $T_2$ locks 2's `succLock`. Next, both threads observe that 2 should be their parent and $T_2$ is suspended. $T_1$ completes the insertion (Figure 5(b)), detects that 4 is not balanced and applies a rotation (Figure 5(c)). Then, $T_2$ resumes and observes that 2 cannot be the parent and 4 should be the new parent.

Thus, upon insertion, the correct parent must be chosen and concurrent rotations to it are prevented by acquiring its `treeLock`. This is done via the `chooseParent` operation that appears in Algorithm 4. In this operation, at every step, there is a node that is a candidate for the correct parent.

**Algorithm 4:** chooseParent(p, s, firstCand)

```
1  candidate = firstCand == p || firstCand == s? firstCand : p
2  while true do
3      lock(candidate.treeLock)
4      if candidate == p then
5          if candidate.right == null then return candidate
6          unlock(candidate.treeLock)
7          candidate = s
8      else
9          if candidate.left == null then return candidate
10         unlock(candidate.treeLock)
11         candidate = p
12     end
13 end
```

**Algorithm 5:** insertToTree(parent, newNode)

```
1  newNode.parent = parent
2  if parent.key < newNode.key then
3      parent.right = newNode
4      parent.rightHeight = 1
5  else
6      parent.left = newNode
7      parent.leftHeight = 1
8  end
9  rebalance(lockParent(parent), parent)
```

Then, its `treeLock` is acquired and if it is validated as the correct parent, the operation returns it. The candidates for the correct parent are the new node's predecessor and successor. To validate that a candidate is the correct parent, it needs to be checked that the appropriate child pointer is empty. If the candidate is the predecessor, the right child should be empty, and if it is the successor, the left child should be empty. If the required child pointer is not empty, then the lock is released, and the other candidate is checked. Typically, the first candidate is the node that was returned from the `search` operation. However, concurrent updates might also occur before the `succLock` is acquired. Then, this node is no longer the predecessor or successor. In this case, we pick (arbitrarily) the predecessor to be the first candidate.

**Updating the Physical Layout** The physical update to the tree, which appears in Algorithm 5, connects the chosen parent to the new node and sets the height of this sub-tree to one. Afterwards, the `rebalance` operation is invoked, to update the heights of the parent's ancestors, and to apply rotations if necessary. This operation requires the parent and its parent to be locked, and thus the `lockParent` is called.

The `lockParent` operation, shown in Algorithm 6, receives a locked node and locks the node pointed by its `parent` field. If after acquiring the lock, this node is the correct parent and it is not marked as removed, it is returned. Otherwise, the operation restarts. Note that the node's parent may change while it is not locked. This is because in order to change a node's parent, it is only necessary to acquire the `treeLocks` of its original and new parents, and there is no need to acquire the node's `treeLock`.

**Algorithm 6:** lockParent(node)

```
1  while true do
2      p = node.parent
3      lock(p.treeLock)
4      if node.parent == p && !p.mark then return p
5      unlock(p.treeLock)
6  end
```

**Algorithm 7:** remove(k)

```
1  while true do
2      node = search(k)
3      p = node.key > k? node.pred : node
4      lock(p.succLock)
5      s = p.succ
6      if k ∈ (p.key,s.key] && !p.mark then
7          if s.key > k then        // Unsuccessful remove
8              unlock(p.succLock)
9              return false
10         end
11         lock(s.succLock)      // Successful remove
12         hasTwoChildren = acquireTreeLocks(s)
13         s.mark = true
14         sSucc = s.succ
15         sSucc.pred = p
16         p.succ = sSucc
17         unlock(s.succLock)
18         unlock(p.succLock)
19         removeFromTree(s, hasTwoChildren)
20         return true
21     end
22     unlock(p.succLock)         // Validation failed,
        restart
23 end
```

### 4.4 The Remove($k$) Operation

The `remove` operation, shown in Algorithm 7, begins with a call to `search(k)` and then operates on the node that was returned. As in the `insert` operation, this operation tries to lock $k$'s predecessor, denoted $p$, and applies the same validation. If validation succeeds, the operation checks whether this is a successful removal, in which case $k$ will be removed, or an unsuccessful removal, in which case the operation will return without changing the tree.

A successful removal to a node, denoted $n$, is applied as follows. The first step is to acquire $n$'s `succLock`, and then the required `treeLock`s. Next, the *mark* field is set to `true` to indicate that this node is logically removed from the tree. Then, the logical ordering is updated (lines 14–18) and finally, the tree's physical layout is updated (line 19). Here, the two `succLock`s are required to prevent concurrent updates to $n$'s predecessor and successor.

**Acquiring `treeLocks`** The `acquireTreeLocks` operation, which appears in Algorithm 8, acquires all the required

**Algorithm 8:** acquireTreeLocks(n)

```
1  lock(n.treeLock)
2  lockParent(n)
3  if n.right == null || n.left == null then    // n is a leaf or
4      if n.right != null then          // has a single child
5          if !tryLock(n.right.treeLock) then restart
6      else if n.left != null then
7          if !tryLock(n.left.treeLock) then restart
8      end
9      return false
10 else                                 // n has two children
11     s = n.succ
12     if s.parent != n then
13         parent = s.parent
14         if !tryLock(parent.treeLock) then restart
15         if parent != s.parent || parent.mark then restart
16     end
17     if !tryLock(s.treeLock) then restart
18     if s.right != null then
19         if !tryLock(s.right.treeLock) then restart
20     end
21     return true
22 end
```

treeLocks for the removal. As discussed before, the removal of a node from the tree's physical layout is carried out differently when the node has two children than when it has less children. Thus, the operation's tasks are to determine how many children the node has, and acquire the set of required treeLocks accordingly. After the treeLocks are acquired, the number of children that the node has cannot change due to concurrent updates.

The acquireTreeLocks operation receives a node, $n$, acquires the required treeLocks, and returns true if $n$ has two children and false otherwise. To guarantee the consistency of its response, it is enough to acquire $n$'s treeLock (to block concurrent updates to its children by insertions, removals or rotations). However, the removal necessitates the acquisition of additional nodes' treeLocks. These nodes are located lower than $n$ in the tree. As shall be described in Section 5, the locking order of the treeLocks is from the lower nodes in the tree to the higher ones. Thus, to acquire locks on these nodes, and to avoid deadlocks, the locking is not blocking, and if an attempt to lock fails, current locks are released and the operation restarts. This means that the number of children that $n$ has may change. Thus, in each iteration, after acquiring $n$'s treeLock, it is necessary to recheck the number of children $n$ has.

The list of nodes whose treeLocks are acquired in this operation are:

- $n$ and $n$'s parent.
- If $n$ has less than two children, $n$'s child (if it exists).
- If $n$ has two children, $s$, $s$'s parent and $s$'s child (if it exists), where $s$ is $n$'s successor.

**Algorithm 9:** removeFromTree(n, hasTwoChildren)

```
1  if !hasTwoChildren then          // n is a leaf or has a
2      child = n.right == null ? n.left : n.right     // single
         child
3      parent = n.parent
4      updateChild(parent, n, child)
5  else                             // n has two children
6      s = n.succ
7      child = s.right
8      parent = s.parent
9      updateChild(parent, s, child)
10     copy n's left, right, leftHeight, rightHeight to s
11     n.left.parent = s
12     if n.right != null then     // n.right may be null if
13         n.right.parent = s      // s was the right child
14     end
15     updateChild(n.parent, n, s)
16     if parent == n then         // rebalance begins from s
17         parent = s
18     else        // rebalance begins from lower nodes
19         unlock(s.treeLock)
20     end
21     unlock(n.parent.treeLock)
22 end
23 unlock(n.treeLock)
24 rebalance(parent, child)
```

**Algorithm 10:** updateChild(parent, oldCh, newCh)

```
1  if parent.left == oldCh then
2      parent.left = newCh
3  else
4      parent.right = newCh
5  end
6  if newCh != null then newCh.parent = parent
```

**Updating the Physical Layout** The physical removal is done via the removeFromTree operation, which appears in Algorithm 9. If the node has at most one child, it is removed by connecting its parent to its child (which may be null). This update is applied via the updateChild operation (Algorithm 10).

A node with two children is removed by relocating $n$'s successor, denoted $s$, to $n$'s location in the tree. This is done in two steps: (i) $s$ is detached from its current location, by updating its parent to point to its child (lines 7–9), (ii) $s$'s location is updated to $n$'s location, by setting $s$' tree fields (i.e., parent, right, left, leftHeight, rightHeight) to $n$'s tree fields, and setting $n$'s parent and children to point to $s$ (lines 10–15).

During these updates, $s$ is not reachable via the tree layout pointers. However, concurrent searches cannot miss its key, which remains reachable via the logical ordering.

After $n$ was removed, the heights are updated starting from the location of the removal. If $n$ had less than two children, then the update begins from $n$'s parent. Otherwise, it begins from $s$'s original parent, or from $s$ if $n$ was its

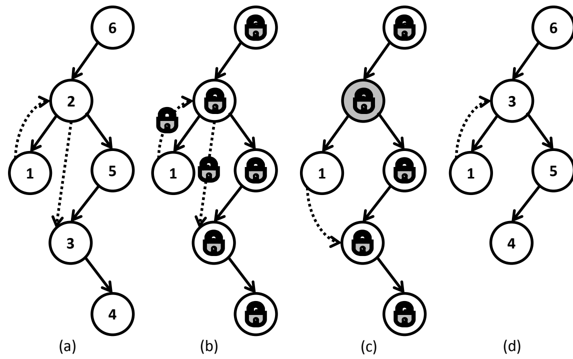


Figure 6: An example of the `remove(2)` operation. The initial tree is shown in (a); in (b) the tree after acquiring `succLocks` of 1 (2's predecessor) and 2, and the `treeLocks` of 2, 6 (2's parent), 3 (2's successor), 5 (3's parent) and 4 (3's child); in (c) the tree after marking 2 as removed and updating the logical ordering; in (d) the tree after relocating 3 to 2's location.

original parent. The heights are updated via the `rebalance` operation, which receives the nodes when they are locked.

An example of the `remove` operation appears in Figure 6.

### 4.5 Balancing the Tree

In AVL trees, after every update operation, a process of recovering the tree to a balanced state begins. The tree is balanced when all nodes are balanced. A node is balanced if the difference between the heights of its sub-trees is at most one. If it is imbalanced, the difference is two. Rotations are applied to rebalance imbalanced nodes. During the recovery process, multiple nodes may be examined, and any updates to these nodes or to their descendants must be blocked. Thus, a concurrent AVL may result in severe bottlenecks.

To avoid these bottlenecks, we apply a relaxed balancing. We use Bougé et al.'s algorithm [6] for relaxed balanced trees that guarantees the tree is balanced in a quiescent state. In their algorithm, nodes maintain two fields, denoted `leftHeight` and `rightHeight`, and an imbalanced node has a difference of *at least two* between the heights of its sub-trees. The decision of whether to apply a rotation, and if so, which type should be applied, is based on these fields. These heights may not reflect the current height of the left and right sub-trees. Thus, a rotation is not guaranteed to lead the node to a balanced state, nor is it guaranteed not to worsen the shape of the tree. However, as shown in [6], applying the AVL rotations on the basis of these heights leads to a strict AVL tree in a quiescent state.

Tree balance is validated from the node that was mutated by the `insert` or `remove` and up to the root. The root is a sentinel and thus it is not rotated. For each node that is scanned, its height is updated before checking whether it is balanced. If the traversal approached the node from the left child, then its `leftHeight` is updated, and otherwise its `rightHeight` is updated. If the height was not changed,

**Algorithm 11:** rotate(child, n, parent, leftRotation)

```
1 updateChild(parent, n, child)
2 n.parent = child
3 if leftRotation then
4 |   updateChild(n, child, child.left)
5 |   child.left = n
6 |   n.rightHeight = child.leftHeight
7 |   child.leftHeight = max(n.leftHeight, n.rightHeight) + 1
8 else Symmetric
```

and the node is balanced, then this operation did not cause a change in the height of its ancestors. In particular, the ancestors are balanced, and thus, the operation can terminate.

If the examined node is imbalanced, rotation is applied. Roughly speaking, a rotation switches the parent and child roles of the node and one of its children. That is, the child becomes the node's new parent, and the child's parent is the node's old parent. Switching the node with its left child is called a *right* rotation, and switching with the right child is called a *left* rotation. The decision of which rotation to apply is based on the node's *balance factor*, which is the difference between its `leftHeight` and `rightHeight`. In some cases, two consecutive rotations are required, first on the node's child and then on the node. The decision of whether to apply one or two rotations depends on the balance factor of the node's child. Algorithm 11 presents the code that applies a rotation, given a node, its child and its parent.

The `rebalance` operation (Algorithm 12), receives two locked nodes, *node* and its *child* (which may be `null`). After insertion, these nodes are the new node's grandparent and parent. After removal, if the removed node had less than two children, these nodes are its parent and child; otherwise, these are the parent and child of the node's successor. The `rebalance` begins with updating *node*'s height on the basis of *child*'s height (Algorithm 13). Then, it checks *node*'s balance factor. If the height of *node* has not changed and the balance factor is valid, the operation terminates. If the height has changed and the balance factor is valid, the traversal continues with the node's parent. Otherwise, if the balance factor is not valid, the rotation process begins.

The rotation process consists of several steps. First, a check is conducted to see whether *child* is the appropriate child to recover the balance factor. If not, *child*'s `treeLock` is released and *child* is set to the other child and attempted to be locked (lines 7–19). As in the `acquireTreeLocks` operation, this acquisition is against the locking order, and thus it is not blocking. If the lock acquisition fails, the `restart` operation (Algorithm 14) is called. In this operation, all `treeLocks` are released and reacquired. If after reacquiring the lock, *node* is marked as removed, the operation terminates. Otherwise, *child* is set to one of its children, its `treeLock` is acquired, and the `rebalance` restarts.

**Algorithm 12:** rebalance(node, child)

```
1  while node != root do
2      isLeft = (node.left == child)
3      updated = updateHeight(child, node, isLeft)
4      bf = node.leftHeight - node.rightHeight
5      if !updated && |bf| < 2 then return
6      while |bf| ≥ 2 do
7          if (isLeft && bf ≤ −2) || (!isLeft && bf ≥ 2) then
8              if child != null then unlock(child.treeLock)
9              child = isLeft? node.right : node.left
10             isLeft = !isLeft
11             if !tryLock(child.treeLock) then
12                 if !restart(node, parent) then return
13                 parent = null
14                 bf = node.leftHeight - node.rightHeight
15                 child = bf ≥ 2? node.left : node.right
16                 isLeft = (node.left == child)
17                 continue
18             end
19         end
20         chBF = child.leftHeight - child.rightHeight
21         if (isLeft && chBF <0) || (!isLeft && chBF >0)
            then
22             grandChild = isLeft? child.right : child.left
23             if !tryLock(grandChild.treeLock) then
24                 unlock(child.treeLock)
25                 if !restart(node, parent) then return
26                 <same code as in lines 13–17>
27             end
28             rotate(grandChild, child, node, isLeft)
29             unlock(child.treeLock)
30             child = grandChild
31         end
32         if parent == null then parent = lockParent(node)
33         rotate(child, node, parent, !isLeft)
34         bf = node.leftHeight - node.rightHeight
35         if |bf| ≥ 2 then
36             unlock(parent.treeLock)
37             parent = child
38             child = null
39             isLeft = bf ≥ 2? false: true
40             continue
41         end
42         temp = node; node = child; child = temp
43         isLeft = (node.left == child)
44         bf = node.leftHeight - node.rightHeight
45     end
46     if child != null then unlock(child.treeLock)
47     child = node
48     node = parent != null? parent: lockParent(node)
49     parent = null
50 end
```

A check is then performed to see whether two rotations are needed. If so, the `treeLock` of *child*'s child is acquired and the rotation is applied (lines 20–31). The acquisition of this lock is also non-blocking, and if it fails, `restart` is called. Next, the rotation to *node* is applied (lines 32–33). To this end, its parent's `treeLock` is acquired. After that, if the node is still not balanced, the process begins again (lines 34–41). Otherwise, we check whether its new parent (i.e., its old child) is balanced (lines 42–44). After both are balanced, the traversal continues upwards in the tree, with the node's old parent (lines 46–49).

**Algorithm 13:** updateHeight(ch, node, isLeft)

```
1  newH = ch == null? 0: max(ch.leftHeight, ch.rightHeight)
     + 1
2  oldH = isLeft? node.leftHeight : node.rightHeight
3  if isLeft then
4      node.leftHeight = newH
5  else
6      node.rightHeight = newH
7  end
8  return oldH == newH
```

**Algorithm 14:** restart(node, parent)

```
1  if parent != null then unlock(parent.treeLock)
2  while true do
3      unlock(node.treeLock)
4      lock(node.treeLock)
5      if node.mark then            // No need to balance
6          unlock(node.treeLock)
7          return false
8      end
9      bf = node.leftHeight - node.rightHeight
10     child = bf ≥ 2? node.left : node.right
11     if child == null then return true
12     if trylock(child.treeLock) then return true
13 end
```

Some of the lock releases are omitted from the code for brevity. We also note that there is an edge-case where *node* was not balanced, the `restart` operation was invoked, and afterwards *node* was detected as removed. If *node* was removed by relocating its successor, $s$, to *node*'s location, then $s$ is not balanced. However, it is the responsibility of the thread that removed *node* to invoke `rebalance` on $s$. This happens in `removeFromTree`, after returning from `rebalance`. We omit this code for simplicity.

### 4.6 Binary Search Tree without Balancing

From the above description of the AVL tree, one can construct a BST without balancing. The BST is very similar to the AVL tree we presented, and it can be obtained by removing the balancing operation and applying minor optimizations (i.e., acquiring fewer `treeLock`s).

### 4.7 Supporting Additional Operations

Our design can be used to support additional operations:

**Retrieving the Minimal/Maximal Values** Having the `pred` and `succ` fields in the nodes allows us to support the operations `min` and `max`. To access the minimal value,

it is enough to read the succ field of $N_{-\infty}$ and check its mark field. If the latter is false, then the node's key can be returned; otherwise, the operation is restarted. The max operation is implemented similarly using $N_{\infty}$'s pred field.

**Iterating over Tree Elements** Iteration requires implementation of: (i) first(), which returns the minimal node in the tree, and (ii) next(node), which returns the successor of the given node. first() is similar to minimal with the exception that it returns the *node* and not its key. The next(node) operation can be implemented similarly by reading the succ field from *node* (instead of $N_{-\infty}$).

# 5. Correctness

In this section we discuss the correctness of our algorithm.

## 5.1 Lock Ordering and Deadlock Freedom

We now present the locking order that threads obey, and thus show that deadlocks cannot occur. There are two types of locks, succLocks and treeLocks, and we order them such that the succLock should be acquired first. Between two succLocks we order acquisition such that the lock of the node with the smaller key should be acquired first.

Between two treeLocks we order acquisition such that the lock of the node that appears lower in the tree should be acquired first. Whenever threads determine that one node appears lower (or higher) than another node in the tree, it cannot change due to concurrent operations. This follows since threads lock nodes that have an ancestor-descendant relation and determine that a node is lower than another node only after acquiring one of their treeLocks. The roles of ancestor-descendant can only be switched via the rotate operation, which requires both of their treeLocks.

When locking treeLocks against the locking order is required, threads optimistically attempt to acquire the lock (without blocking on it), and if they fail, all locks are released and the operation is restarted. When locking several treeLocks against the locking order, threads lock from the higher node to the lower one. That is, threads obey another locking order. This approach cannot result in a livelock. Livelock may occur when threads contend without blocking on the same set of locks and acquire them in a different order. Here, livelock cannot occur when two threads attempt to acquire locks against the locking order because they obey another locking order. Nor can livelock occur when two threads attempt to acquire locks, one obeying the locking order and the other not obeying it, since the first one will block on the locks until acquiring them.

The BST follows the same lock ordering as the AVL tree, and uses a subset of its locks; thus, it is also deadlock free.

## 5.2 Linearizability

To show linearizability, we provide the linearization points for each operation. The linearization points of unsuccessful inserts and removes (neither affects the tree) are where they return false, i.e. in lines 9 of insert and remove.

The linearization point of a successful insert($k$) is in line 16, where $p$'s succ field is updated to point to the new node. Any future insert($k$) or remove($k$) will observe $k$ once it has acquired the succLock of $k$'s predecessor, or once it has detected a different node $n$, acquired its succLock and observed that $k \notin (n\text{'s key}, \text{succ}(n)\text{'s key})$.

The linearization point of a successful remove($k$) is in line 13, where $n$'s mark field is set to true. Any future insert($k$) or remove($k$) will acquire the succLock of $k$'s predecessor, and will observe that it has a different successor. Any future contains($k$) that observes $k$ will find its mark field set to true.

The linearization point of a successful contains($k$) is when the mark field of the node with value $k$ was observed to be false. The linearization point of an unsuccessful contains is more delicate. The simple case is when the node is marked as removed or when $k$ is not in the tree. In this case the linearization point is when the mark field was observed as true, or upon observing that *node* has a bigger key in line 3. However, it may happen that the nodes read in line 2 were also removed from the tree (but the update was not completed yet). In this case, it is possible that another concurrent thread inserts $k$. Since this update was not observed in line 3, the insertion must have begun after the contains began. Thus, we linearize the unsuccessful contains just before the linearization point of the new insert.

## 5.3 Progress Guarantees

We now provide a sketch of the proof of the contains' progress guarantees. The contains consists of two phases:

- The traversal along the tree via the *right* and *left* fields.
- The traversal along the nodes via the *pred* and *succ* fields.

The first phase of the traversal is lock-free. A thread may stray from its path due to rotations or removals of nodes with two children. However, in these cases another thread has made progress (it successfully applied an insert or remove).

The second phase is also lock-free. If there are no concurrent updates, after a finite number of steps, the thread will find an unmarked node which is the predecessor of the key under search, or a node that precedes the predecessor. Then, if there are no concurrent updates, after a finite number of steps, the thread will find the predecessor and successor of the key under search. This traversal may only be delayed if the order layout is concurrently updated by another thread. In that case, the update operation was linearized (before or when the order layout was updated) and the other thread has made progress.

# 6. Evaluation

To evaluate the performance of our algorithms, we ran experiments on an AMD Opteron(tm) Processor 6376 with 128GB RAM and 64 cores: four processors with sixteen cores each and with hyper-threading support. We used Ubuntu 12.04 LTS and OpenJDK Runtime version 1.7.0_45

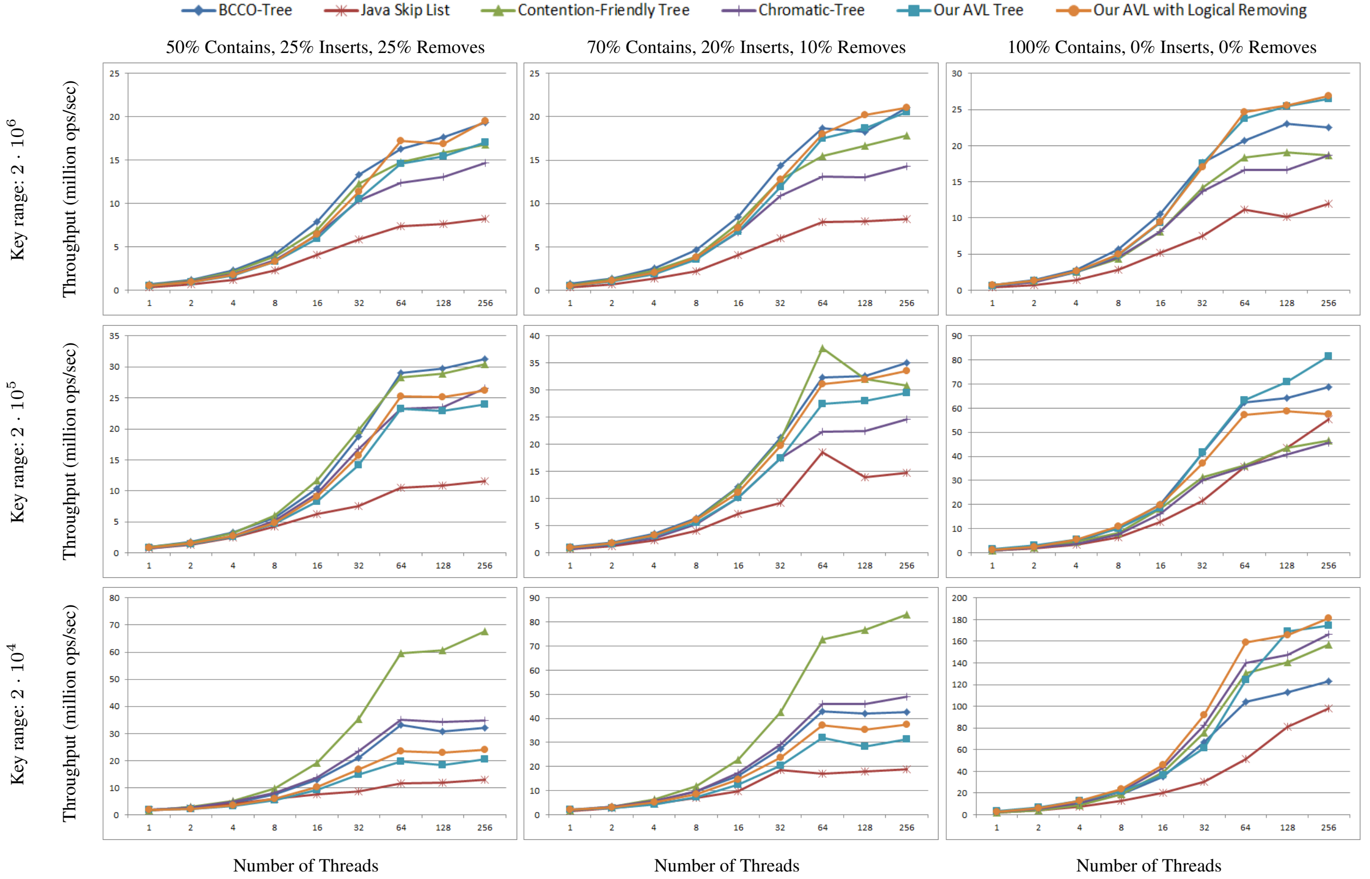


Table 1: Throughput for balanced tree implementations under different workloads on a 4-socket Opteron (64 h/w threads).

using the 64-bit Server VM (build 24.45-b08, mixed mode). We compared our algorithms to the following:

- BCCO Tree – The lock-based, relaxed AVL tree by Bronson et al. [8]. This is a variation of internal trees, referred to as *partially-external* trees.
- Contention-friendly tree – The lock-based, partially-external, relaxed AVL tree by Crain et al. [10], which delays rotations and removals and performs them in a separate maintenance thread.
- Java's Skip List – The non-blocking skip-list by Doug Lea, based on the work of Fraser and Harris [12], and included in the Java standard library.
- Chromatic Tree – The non-blocking, external, relaxed balance red-black tree, Chromatic6, by Brown et al. [9], which performs rotations only when the number of violations on a path is more than 6.
- EFRB-Tree – The non-blocking, external BST by Ellen et al. [11].

We also consider a variation of our trees, denoted *logical removing*, that implements a partially-external tree. In this variation, a node with two children is marked as logically removed via a designated flag, and it is not physically removed from the ordering layout or the physical layout. It will be physically removed only if its number of children reduces to one due to another removal or due to rotations. An insert can revive such a node by flipping this flag to false.

**Differentiating Features** Two key features distinguish our algorithms from past work on internal trees. First, our lookup operation is lock-free. This fault-tolerance property is important in large-scale systems where failure to guarantee some form of fault-tolerance makes reliability difficult to attain.

Second, in contrast to other approaches, our algorithms perform timely deletion. That is, they free the node upon removal (even when the deleted node is an internal node with two children), rather than keeping the node and physically deleting it later. This is useful as it keeps the memory consumption to what is expected by the programmer: a function of the keys which are *currently* in the tree. This is particularly important in long-running applications where failure to remove deleted nodes can slowly lead to longer traversals.

The main performance question we wanted to address was the cost of these unique features: *would the throughput of our algorithms be comparable to that of existing algorithms which lack this combination?*

We focus our evaluation on workloads that make heavy use of `contains` operations:

- *100C-0I-0R*: 100% contains, 0% inserts, 0% removes.
- *70C-20I-10R*: 70% contains, 20% inserts, 10% removes.
- *50C-25I-25R*: 50% contains, 25% inserts, 25% removes.

We ran five-second trials, where each thread reported the number of operations it completed. We report the total number of operations applied by all threads, that is, the total throughput. The number of threads is $2^i$ where $i$ varies between 0-8.

During the trial, each thread randomly chooses a type of operation with respect to the given distribution and then randomly chooses the value for that operation from a given range. The examined range sizes are: $2{\cdot}10^4$, $2{\cdot}10^5$ and $2{\cdot}10^6$. Before each trial, we prefilled the data structure as follows. For 100C-0I-0R and 50C-25I-25R, the data structures were prefilled to a size of ½ of the key range. For 70C-20I-10R, the data structures were prefilled to a size of ⅔ of the key range (the expected size at steady state). During prefilling, we ran the same workload as in the timed trials (i.e., same number of threads and same operations distribution), until reaching the desired size. We ran every experiment 8 times and report the arithmetic average. Each batch of 8 trials was run in its own JVM instance. To avoid HotSpot effects, we ran a warm-up phase before executing the trials. Table 1 compares balanced data structures, and Table 2 compares unbalanced data structures.

**Throughput Evaluation** In Table 1, it can be seen that under a heavy load of mutating operations (first column), and when the tree is quite small, our algorithms perform somewhat worse than the other trees. Under heavy load, threads spend more time waiting for locks. Also, since the lookup is optimistic, threads may progress along some path, and then due to rotations may find themselves scanning a different path. More changes result in more threads that stray from the correct path due to rotations. However, as the tree's size increases, even under heavy load of write operations our algorithm is comparable to the other implementations. When all of the operations are contains operations, our algorithm—while still providing lock-free contains—outperforms the state-of-the-art BSTs. In terms of space, the BCCO-tree may maintain up to 50% "zombie" nodes that have been logically, but not physically, removed. These nodes can be used to avoid allocation if a subsequent insert is attempting to insert the same key. This represents a different tradeoff than our approach, in which some allocations can be avoided by keeping "zombie" nodes in the tree after removal. The ability to avoid allocation using zombie nodes decreases as the key range increases and the probability for inserting a key that has been removed decreases. The benefit of saving allocation is also apparent when comparing our *logical removing* variation to our AVL under workloads that include update operations. In these workloads, the variation performs better than the AVL.

Table 2 presents the results for 70C-20I-10R and 100C-0I-0R of the unbalanced trees (50C-25I-25R produces similar results to 70C-20I-10R). It can be seen that our BST is often comparable to the *EFRB-Tree*. When all operations are contains operations, the comparison is between different implementations of lock-free contains for an internal tree (our tree), a partially-external tree (our variation), and an external tree (*EFRB-Tree*). When the operation mix includes update operations, threads may be blocked in both of our trees (waiting for locks), while in the *EFRB-Tree*, threads are never blocked. When the tree size is small, the external and partially-external trees have an advantage over our tree. Internal trees may suffer more contention in smaller trees since threads contend on nodes more frequently (since there could be up to two times fewer nodes in internal trees than in external trees). When the tree size is large, however, our tree is comparable to the *EFRB-Tree*.

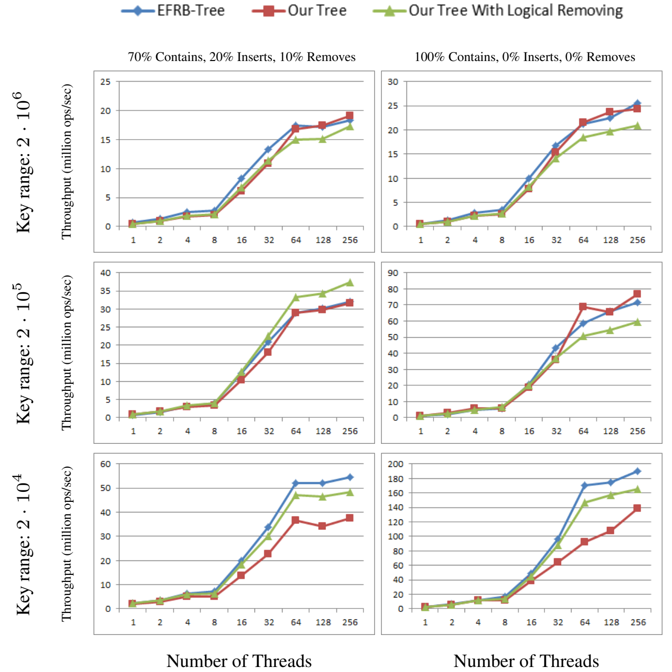


Table 2: Throughput for different BST implementations.

## 7. Related Work

The literature contains many variants of concurrent trees that can be roughly classified according to the following characteristics:

**Internal vs. External** An internal tree (e.g., [8, 10, 13]) maintains values in inner nodes whereas an external tree (e.g., [7, 9, 11, 15]) maintains values only at the leaves. One challenge in internal trees is removal of a node with two children. Bronson et al. [8] and Crain et al. [10] cope with this challenge by marking such a node as removed, and do not *physically* remove it until the node has a single child. Howley et al. [13] remove such a node by copying the value of its successor to the node, and then by removing

the successor if it has a child (otherwise it is only logically removed).

**Pessimistic vs. Optimistic** Most concurrent trees in the literature use optimistic concurrency [14], but it is also possible to use pessimistic locking protocols [3]. Optimistic trees are traversed without acquiring any lock, and locks are acquired when the required node is reached. One challenge in such trees is the validation that no other concurrent operation has occurred between detecting the required node after observing some state of the tree and the lock acquisition. Known algorithms [7, 8, 10, 11, 13] cope with this challenge by stamping the node with the ongoing operation or with a new *version* number.

**Balanced vs. Unbalanced** Balancing might require additional synchronization, and the literature contains both unbalanced [11, 13, 15] and balanced trees with different balancing mechanisms, e.g., red-black trees [5, 9], and AVL [8, 10]. Concurrent trees are typically *relaxed balanced*, meaning that the mutating operations are decoupled from the rebalancing operations. That is, a mutating operation is first completed and only sometime afterwards the rebalance process begins. This process may be delayed, for example, while waiting to acquire locks. In [8] balance is applied after every mutating operation, whereas in [10] balance is applied by a single thread that runs in the background.

**BST vs. Threaded BST** In a threaded BST, every node that normally had `null` in its right child pointer will have a pointer to the node's successor, and every node that normally had `null` in its left child pointer will have a pointer to the node's predecessor. In our approach, every node maintains this information and it is crucial for nodes that neither of their children is `null`. Pfaff [16] mentions the possibility of adding an in-order list to the sequential tree. This is done by extending each node to point to its predecessor and successor. Pfaff claims that it does not improve performance; no details are provided on the locking discipline used when this in-order list is added. In contrast, we show that using a locking that is based on logical-ordering is a sweet spot for simplifying the algorithm and obtaining performance better or comparable to state-of-the-art algorithms that use more complicated mechanisms.

## 8. Conclusion

We presented two practical concurrent binary search tree algorithms: a non-balanced tree and an AVL tree. The core idea behind our algorithms is the notion of logical ordering, which is explicitly maintained in the tree. Separating the logical ordering from the tree layout enables clean separation between operations which update the tree layout, such as rotations, and those that look for elements. We leveraged this idea to design an intuitive, simple lookup operation, which is also lock-free. We have implemented our algorithms and showed that they are competitive with state-of-the-art balanced BST implementations.

## Acknowledgements

We would like to thank Guy Gueta, Faith Ellen, Tim Harris, and Adam Morrison for their insightful comments.

## Errata

The statements in lines 2 and 3 of Algorithm 2 have to be swapped to obtain linearizability of the BST and AVL (as described in Section 5.2). We thank Noam Rinetzky, William Sigouin, Trevor Brown, and Dan Alistarh who (in parallel) have pointed out this subtlety.

## References

[1] ADELSON-VELSKII, G., AND LANDIS, E. M. An algorithm for the organization of information. In *Proc. of the USSR Academy of Sciences,146:263-266* (1962).

[2] BAYER, R. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica 1*, 4 (1972), 290–306.

[3] BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on b-trees. *Acta Informatica 9*, 1 (1977), 1–21.

[4] BENDER, M. A., FINEMAN, J. T., GILBERT, S., AND KUSZMAUL, B. C. Concurrent cache-oblivious b-trees. In *SPAA* (2005), pp. 228–237.

[5] BESA, J., AND ETEROVIC, Y. A concurrent red-black tree. *Journal of Parallel and Distributed Computing 73*, 4 (2013), 434 – 449.

[6] BOUGÉ, L., GABARRÓ, J., MESSEGUER, X., AND SCHABANEL, N. Height-relaxed avl rebalancing: A unified, fine-grained approach to concurrent dictionaries, 1998.

[7] BRAGINSKY, A., AND PETRANK, E. A lock-free b+tree. In *SPAA* (2012), pp. 58–67.

[8] BRONSON, N. G., CASPER, J., CHAFI, H., AND OLUKOTUN, K. A practical concurrent binary search tree. In *PPoPP* (2010), pp. 257–268.

[9] BROWN, T., ELLEN, F., AND RUPPERT, E. A general technique for non-blocking trees. In *Proc. 19th ACM Symposium on Principles and Practice of Parallel Programming* (2014).

[10] CRAIN, T., GRAMOLI, V., AND RAYNAL, M. A contention-friendly binary search tree. In *Euro-Par* (2013), pp. 229–240.

[11] ELLEN, F., FATOUROU, P., RUPPERT, E., AND VAN BREUGEL, F. Non-blocking binary search trees. In *PODC* (2010), pp. 131–140.

[12] FRASER, K. Practical lock-freedom. Tech. Rep. UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[13] HOWLEY, S. V., AND JONES, J. A non-blocking internal binary search tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures* (2012), pp. 161–171.

[14] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.

[15] NATARAJAN, A., AND MITTAL, N. Fast concurrent lock-free binary search trees. In *Proc. 19th ACM Symposium on Principles and Practice of Parallel Programming* (2014).

[16] PFAFF, B. Performance analysis of BSTs in system software. In *SIGMETRICS* (2004), ACM, pp. 410–411.