A transaction-friendly binary search tree

Tyler Crain, Vincent Gramoli, Michel Raynal

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1984 – septembre 2011

A Transaction-Friendly Binary Search Tree

Tyler Crain[*] , Vincent Gramoli[**] Michel Raynal[***]
tyler.crain@irisa.fr, vincent.gramoli@epfl.ch, raynal@irisa.fr

Abstract: Transactions, which provide optimistic synchronization by avoiding the use of blocking, greatly simplify multicore programming. In fact, the programmer has simply to encapsulate sequential operations or existing critical sections into transactions to obtain a safe concurrent program. Programmers have thus started evaluating transactional memory using data structures originally designed for pessimistic (i.e., non-optimistic) synchronization, whose prominent example is the red-black tree library developed by Oracle Labs that is part of STAMP and microbench distributions.

Unfortunately, existing data structures are badly suited for optimistic synchronization as they rely on strong structural invariants, like logarithmic tree depth, to bound the step complexity of pessimistically synchronized accesses. By contrast, this complexity does not apply to optimistically synchronized accesses thus making the invariants overly conservative. More dramatically, guaranteeing such invariants tends to increase the probability of aborting and restarting the same access before it completes.

In this paper, we introduce a concurrent binary search tree that breaks transiently its balance structural invariants for efficiency, a property we call *transaction-friendly*. We show that this new tree outperforms the existing transaction-based version of the AVL and the red-black trees. Its key novelty stems from the *decoupling* of update operations: they are split into one transaction that modifies the abstraction state and multiple ones that restructure its tree implementation. The resulting transaction-friendly library trades aborts for few additional access steps and, in particular, it speeds up a transaction-based travel reservation application by up to 3.5×.

Key-words: Transactional Memory, concurrent data structures, balanced binary trees

---

Un arbre binaire de recherche dédié aux accès transactionnels

Résumé : *Les transactions, qui utilisent une technique de synchronisation optimiste, simplifient la programmation des architectures multi-cœurs. En effet, un programmeur a seulement besoin d'insérer des oprations dans des transactions afin d'obtenir un programme concurrent correct. Les programmeurs ont donc tout naturellement utilisé cette encapsulation sur plusieurs structures de donnes originellement dédiées à la programmation pessimiste (non optimistes) pour èvaluer les transactions. L'exemple le plus probant tant l'arbre transactionnel rouge-noir développé par Oracle Labs et présent dans plusieurs distributions.*

Malheureusement, ces structures de données ne sont pas adaptées à des exécutions optimistes car elles reposent sur des invariants contraignants, comme la profondeur logarithmique d'un arbre, pour borner le coût des accés pessimistes. Une telle complexité ne s'applique pas aux accés optimistes et garantir de tels invariants n'est pas nècessaire et induit de la contention en augmentant la probabilité pour une transaction d'avorter et de recommencer.

Dans ce papier, nous présentons un arbre binaire de recherche qui viole de façon transitoire ces invariants pour gagner en efficacité. Nous montrons que cet arbre est plus efficace que les arbres AVL et rouge-noir. Sa nouveauté majeure rside dans la séparation de ses opérations de modifications : elles sont coupées en une transaction qui modifie l'abstraction et plusieurs autres qui restructurent son implémentation. La bibliothèque qui en résulte limite la quantité d'avortements en augmentant légèrement le nombre d'accés, en particulier, elle amèliore une application de réservation de voyage basée sur les transactions par un facteur multiplicatif de 3,5.

Mots clés : *mémoire transactionnelle, arbre binaire, structures de données concurrente*

---

[*] Projet ASAP : équipe commune avec l'INRIA, le CNRS, l'université de Rennes 1
[**] EPFL
[***] Projet ASAP : équipe commune avec l'INRIA, le CNRS, l'université de Rennes 1

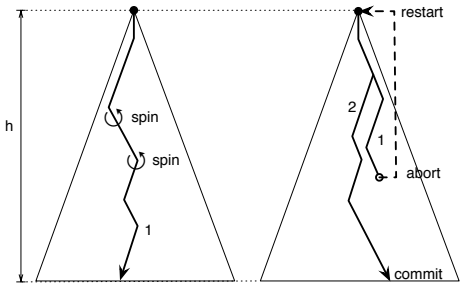


Figure 1: A balanced search tree whose complexity, in terms of the amount of accessed elements, is **(left)** proportional to h in pessimistic execution and **(right)** proportional to the number of restarts in an optimistic execution.

1 Introduction

The multicore era is changing the way we write concurrent programs. In such context, concurrent data structures are becoming a bottleneck building block of a wide variety of concurrent applications. Generally, they rely on invariants (for example balance in binary trees) which prevent them from scaling with multiple cores yet existing relaxations tend to break their semantics [32].

New programming constructs like transactions have been suggested [19, 33]. These transactions build upon *optimistic synchronization*, where a sequence of shared accesses is executed speculatively and might abort. They simplify concurrent programming for two reasons. First, the programmer only needs to delimit regions of sequential code into transactions or to replace critical sections by transactions to obtain a safe concurrent program. Second, the resulting transactional program is reusable by any programmer, hence a programmer composing operations from transactional library into another transaction is guaranteed to obtain new deadlock-free operations that execute atomically. By contrast, *pessimistic synchronization*, where each access to some location x blocks further accesses to x, is harder to program with [30,31] and hampers reusability.

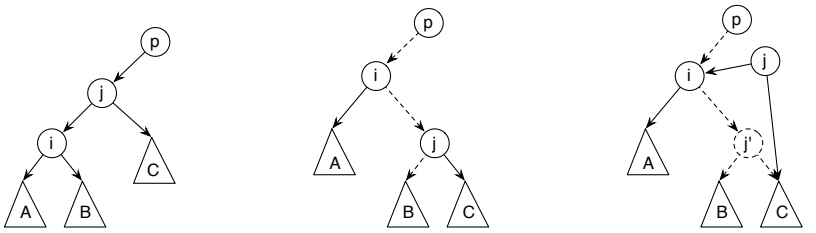
Yet it is unclear how one can adapt a data structure to access it efficiently through transactions. As a drawback of the simplicity of using transactions, the existing transactional programs spanning from low level libraries to topmost applications directly derive from sequential or pessimistically synchronized programs. The impacts of optimistic synchronization on the execution is simply ignored.

To illustrate the difference between optimistic and pessimistic synchronizations, consider the example of Figure 1 depicting their step complexity when traversing a tree of height h from its root to a leaf node. On the left, steps are executed pessimistically, potentially spinning before being able to acquire a lock, on the path converging towards the leaf node. On the right, steps are executed optimistically and some of them may abort and restart, depending on concurrent thread steps. The pessimistic execution of each thread is guaranteed to execute $O(h)$ steps, yet the optimistic one may need to execute $\Omega(hr)$ steps, where r is the number of restarts. Note that r depends on the probability of conflicts with concurrent transactions that depends, in turn, on the transaction length h. Although it is clear that a transaction must be aborted before violating the abstraction implemented by this tree, e.g., inserting k successfully in a set where k already exists, it is unclear whether a transaction must be aborted before slightly unbalancing the tree implementation to strictly preserve the balance invariant.

We introduce *transaction-friendly* data structures as data structures that transiently break such structural invariants without hampering the abstraction consistency in order to speed up transaction-based accesses. Here are our contributions.

- We propose a transaction-friendly binary search tree data structure implementing an associative array and a set abstractions that decouples the operations that modify the abstraction (we call these *abstract transactions*) from operations that modify the tree structure itself but not the abstraction (we call these *structural transactions*). An abstract transaction either inserts or deletes an element from the abstraction and in certain cases the insertion might also modify the tree structure. Some structural transactions rebalance the tree by executing a distributed rotation mechanism: each of these transactions executes a local rotation involving only a constant number of neighboring nodes. Some other structural transactions unlink and free a node that was logically deleted by a former abstract transaction.

- We prove the correctness (i.e., linearizability) of our tree and we compare its performance against existing transaction-based versions of an AVL tree and a red-black tree, widely used to evaluate transactions [14, 15, 20, 21, 33]. More specifically, we have run a micro-benchmark and travel reservation application on top of our transaction-friendly tree, on a 48-core machine. The transaction-friendly tree improves by up to 1.6× the performance of the AVL tree on the micro-benchmark, and by up to 3.5× the performance of the built-in red-black tree on the travel reservation application, already well-engineered for transactions.



(a) Initial tree (b) Result of usual right rotation (c) Result of new right rotation

Figure 2: The classical rotation modifies node n in the tree and forces a concurrent traversal at this node to backtrack. The new rotation left j unmodified, adds j' and postpones the physical deletion of j.

- We illustrate (i) the portability of our transaction-friendly tree by evaluating it on two different Transactional Memories (TMs), TinySTM [15] and $\mathcal{E}$-STM [16] and with different configuration settings, hence outlining that our performance benefit is independent from the transactional algorithm it uses, and (ii) its reusability by composing straightforwardly the remove and insert into a new move operation. In addition, we compared the benefit of relaxing data structures into transaction-friendly ones against the benefit of only relaxing transactions, by evaluating elastic transactions. It shows that, for a particular data structure, refactoring the data structure algorithm is preferable to refactoring the underlying transaction algorithm.

The paper is organized as follows. In Section 2 we describe the problem related to the use of transactions in existing balanced trees. In Section 3 we present our transaction-friendly binary search tree. In Section 4 we evaluate our tree experimentally and illustrate its portability and reusability. In Section 5 we discuss how to make other data structures transaction-friendly. In Section 6 we describe the related work and Section 7 concludes.

2 The Problem with Balanced Trees

In this section, we focus our attention on the structural invariant of existing tree libraries, namely the *balance*, and enlighten the impact of their restructuring, namely the *rebalancing*, on contention.

Trees provide logarithmic access time complexity given that they are balanced, meaning that among all downward paths from the root to a leaf, the length of the shortest path is not far apart the length of the longest path. Upon tree update, if the threshold is exceeded, the structural invariant is broken and a rebalancing is triggered to restructure accordingly. This threshold depends on the considered algorithm: AVL trees [1] do not tolerate the longest length to exceed the shortest by 2 whereas red-black trees [5] tolerate the longest to be twice the shortest, thus restructuring less frequently. Yet in both cases the restructuring is triggered immediately when the threshold is reached to hide the imbalance from further operations.

Generally, one takes an existing tree algorithm and encapsulates all its accesses within transactions to obtain a concurrent tree whose accesses are guaranteed atomic (i.e., linearizable), however, the obtained concurrent transactions would likely *conflict* (i.e., one will access the same location another is modifying), resulting in the need to abort one of these transactions which will lead to a significant waste of efforts. This is in part due to the fact that encapsulating an *update* operation (i.e., an insert or a remove operation) into a transaction boils down to encapsulating four phases in the same transaction:

1. the modification of the abstraction,
2. the corresponding structural adaptation,
3. a check to detect whether the threshold is reached and
4. the potential rebalancing.

A transaction-based red-black tree An example is the transaction-based binary search tree developed by Oracle Labs (formerly Sun Microsystems) and other researchers to extensively evaluate transactional memories [14, 15, 20, 21, 33]. This library relies on the classical red-black tree algorithm that bounds the step complexity of pessimistic insert/delete/contains. It has been slightly optimized for transactions by removing sentinel nodes to reduce false-conflicts, and we are aware of two benchmark-suite distributions that integrate it, STAMP and microbench.

Each of its update transactions encapsulate all the four phases given above even though phase (1) can be decoupled from phases (3) and (4) if transient violations of the balance invariant were tolerated. Such a decoupling is appealing given that

phase (4) is subject to conflicts. In fact, the algorithm balances the tree by executing rotations starting from the position where a node is inserted or deleted and possibly going all the way up to the root. As depicted in Figure 2(a) and (b), a rotation consists of replacing the node where the rotation occurs by the child and adding this replaced node as one of its subtrees. A node cannot be accessed concurrently by an abstract transaction and a rotation, otherwise the abstract transaction might miss the node it targets while being rotated downward. Similarly, rotations cannot access common nodes as one rotation may unbalance the others.

Moreover, the red-black tree does not allow any abstract transaction to access a node that is concurrently being deleted from the abstraction because phases (1) and (2) are tightly coupled within the same transaction. If this was allowed the abstract transaction might end up on the node that is no longer part of the tree. Fortunately, if the modification operation is a deletion then phase (1) can be decoupled from the structural modification of phase (2) by marking the targeted node as logically deleted in phase (1) effectively removing it from the set abstraction prior to unlinking it physically in phase (2). This improvement is important as that it lets a concurrent abstract transaction to travel through the node concurrently being logically deleted in phase (1) without conflicting. Making things worse, without decoupling these four phases, having to abort within phase (4) would typically require the three previous phases to restart as well. Finally, without decoupling only contains operations are guaranteed not to conflict with each other. With decoupling, insert/delete/contains will not conflict with each other unless they terminate on the same node as described in Section 3.

To conclude, for the transactions to preserve the atomicity and invariants of such tree algorithm, they will typically have to keep track of a large *read set* and *write set*, i.e., the sets of accessed memory locations that are protected by a transaction. Possessing large read/write sets increases the probability of conflicts and thus reduces concurrency. This is especially problematic in trees because the distribution of nodes in the read/write set is skewed so that the probability of the node being in the set is much higher for nodes near the root and the root is guaranteed to be in the read set.

Illustration To briefly illustrate the effect of tightly coupling update operations on the step complexity of classical transactional balanced trees we have counted the maximal number of reads necessary to complete typical insert/remove/contains operations. Note that this number includes the reads executed by the transaction each time it aborts in addition to the read set size of the transaction obtained at commit time.

Update	0%	10%	20%	30%	40%	50%
AVL tree	29	415	711	1008	1981	2081
Sun red-black tree	31	573	965	1108	1484	1545
Tx-friendly tree	29	75	123	120	144	180

Table 1: Maximum number of transactional reads per operation on three 2^{12}-sized balanced search trees as the update ratio increases

We evaluated the aforementioned red-black tree, an AVL tree, and our transaction-friendly tree on a 48-core machine using the same transactional memory algorithm[1]. The expectation of the tree sizes is fixed to 2^{12} during the experiments by performing an insert and a remove with the same probability. Table 1 depicts the maximum number of transactional reads per operation observed among 48 concurrent threads as we increase the update ratio, i.e., the proportion of insert/remove operations over contains operations.

For all three trees, the transactional read complexity of an operation increases with the update ratio due to the additional aborted efforts induced by the contention. Despite the red-black and the AVL tree's aim of keeping the complexity of pessimistic accesses $O(\log_2 n)$ (proportional to 12 in this case), where n is the tree size, the read complexity of optimistic accesses grows significantly (14× more at 10% update than at 0%) as the contention increases, independently from the tree size. As described in the sequel, the transaction-friendly tree succeeds in limiting the step complexity raise (2.6× more at 10% update), of data structure accesses when compared against the transactional versions of state-of-the-art tree algorithms.

3 The Transaction-Friendly Binary Search Tree

We introduce the transaction-friendly binary search tree by describing its implementation of a associative array abstraction, mapping a key to a value. In short, the tree speeds up the access transactions by decoupling two conflict-prone operations: the node deletion and the tree rotation. Although these two techniques have been used for decades in the context of data management [13, 25], our algorithm novelty lies in applying their combination to reduce transaction aborts. We first depict, in Algorithm 1, the pseudocode that looks like sequential code encapsulated within transactions before presenting, in Algorithm 2, more complex optimizations.

[1] TinySTM-CTL, i.e., with lazy acquirement [15].

Algorithm 1 A Portable Transaction-Friendly Binary Search Tree

```
1:  State of node n:
2:    node a record with fields:
3:      k ∈ ℕ, the node key
4:      v ∈ ℕ, the node value
5:      lh, rh ∈ ℕ, local height of left/right child,
6:        initially 0
7:      ℓ, r ∈ ℕ, left/right child pointers, initially
      ⊥
8:      localh ∈ ℕ, expected local height, initially 1
9:      del ∈ {true, false}, indicate whether
10:       logically deleted, initially false

11: State of process p:
12:   root, shared pointer to root

13: find(k)_p:
14:   next ← root
15:   while 1 do
16:     curr ← next
17:     val ← curr.k
18:     if val = k then
19:       break
20:     if val > k then next ← read(curr.r)
21:     else next ← read(curr.ℓ)
22:     if next = ⊥ then
23:       break
24:   return curr

25: contains(k)_p:
26:   transaction {
27:     result ← true
28:     curr ← find(k)
29:     if curr.k ≠ k then
30:       result ← false
31:     else
32:       if read(curr.del) then result ← false
33:   } // current transaction tries to commit
34:   return result

35: insert(k, v)_p:
36:   transaction {
37:     result ← true
38:     curr ← find(k)
39:     if curr.k = k then
40:       if read(curr.del) then
41:         write(curr.del, false)
42:       else result ← false
43:     else
44:       // allocate a node called new
45:       new.k ← k; new.v ← v
46:       if curr.k > k then write(curr.r, new)
47:       else write(curr.ℓ, new)
48:   } // current transaction tries to commit
49:   return result

50: right_rotate(parent, left-child)_p:
51:   transaction {
52:     if left-child then
53:       n ← read(parent.ℓ)
54:     else
55:       n ← read(parent.r)
56:     if n = ⊥ then
57:       return false
58:     ℓ ← read(n.ℓ)
59:     if ℓ = ⊥ then
60:       return false
61:     ℓr ← read(ℓ.r)
62:     write(n.ℓ, ℓr)
63:     write(ℓ.r, n)
64:     if ℓ then
65:       write(parent.ℓ, ℓ)
66:     else
67:       write(parent.r, ℓ)
68:     update-balance-values()
69:   } // current transaction tries to commit
70:   return true

71: delete(k)_p:
72:   transaction {
73:     result ← true
74:     curr ← find(v)
75:     if curr.k ≠ k then
76:       result ← false
77:     else
78:       if read(curr.del) then
79:         result ← false
80:       else write(curr.del, true)
81:   } // current transaction tries to commit
82:   return result

83: remove(parent, left-child)_p:
84:   transaction {
85:     if left-child then
86:       n ← read(parent.ℓ)
87:     else
88:       n ← read(parent.r)
89:     if n = ⊥ then
90:       return false
91:     if ¬read(n.del) then
92:       return false
93:     if (child ← read(n.ℓ)) ≠ ⊥ then
94:       if (child ← read(n.r)) ≠ ⊥ then
95:         return false
96:     if left-child then
97:       write(parent.ℓ, child)
98:     else
99:       write(parent.r, child)
100:    update-balance-values()
101:  } // current transaction tries to commit
102:  return true
```

3.1 Decoupling tree rotation

The motivation for rotation decoupling stems from two separate observations: (i) a rotation is tied to the modification that triggers it, hence the process modifying the tree is also responsible of ensuring that its modification does not break the balance invariant and (ii) a rotation affects different parts of the tree, hence an isolated conflict can abort the rotation performed at multiple nodes. In response to these two issues we introduce a dedicated rotator thread to complete modifying transactions faster and we distribute the rotation in multiple local operations to reduce conflicts between transactions. Note that our rotating thread is similar to the collector thread proposed by Dijkstra et al. [13] to garbage collect stale nodes.

Instead of performing rotations within the insert/delete operations, rotations are distributed and performed separately, and the information updated at each node by these local rotations is propagated in the background. This allows the read set of the insert/delete operations to only contain the path from the root to the node(s) being modified and the write set to only contain the nodes that need be modified in order to ensure the abstraction moification (i.e. the nodes at the bottom of the search path), thus reducing conflicts significantly. Let us consider a specific example. If rotations are performed within the insert/delete operations then each rotation increases the read and write set sizes. Take an insert operation that triggers a right rotation such as the one depicted in Figures 2(a)-2(b) where the node that is inserted by this operation in subtree C. Before the rotation the read set for the nodes p, j, i is $\{p.\ell, j.r\}$ and the write set is $\{\}$. Now with the rotation the read set becomes $\{p.\ell, i.r, j.\ell, j.r\}$ and the write set becomes $\{p.\ell, i.r, j.\ell\}$ as denoted in the figure by dashed arrows. Due to $\{p.\ell\}$ being modified, any concurrent transaction that traverses any part of this section of the tree (including all nodes j, i, and subtrees A, B, C, D) will have a read/write conflict with this transaction. In the worst case an insert/delete operation triggers rotations all the way up to the root resulting in conflicts with all concurrent transactions.

Rotation As previously described, rotations are not required to ensure the atomicity of the insert/delete/contains operations so it is not necessary to perform rotations in the same transaction as the insert or delete. Instead rotations are performed separately in their own transactions.

The transaction-friendly binary search tree decouples the rotations by separating them from the insertions and deletions and by distributing them among nodes. More specifically, neither do the insert/delete operations comprise any rotation, nor do the rotations execute on a large block of nodes. Hence, rotations that occur near the root can still cause a large amount of

conflicts, but rotations performed further down the tree are less subject to conflict. If rotations are performed in blocks then even the rotations that occur further down the tree will be part of a likely conflicting transaction, so instead rotations are performed each as a single transaction. Keeping the insert/delete/contains and rotate/remove operations as small as possible allows more operations to execute at the same time without conflicts, increasing concurrency.

Performing local rotations rather than global ones has other benefits. If rotations are performed as blocks then, due to concurrent insert/delete operations, not all of the rotations may still be valid once the transaction commits. Each concurrent insert/delete operation might require a certain set of rotations to balance the tree, but because the operations are happening concurrently the appropriate rotations to balance the tree are constantly changing and since each operation only has a partial view of the tree it might not know what the appropriate rotations are. With local rotations, each time a rotation is performed it uses the most up-to-date local information avoiding repeating rotations at the same location.

The actual code for the rotation is straightforward. Each rotation is performed just as it would be performed in a sequential binary tree (see Figure 2(a)-2(b)), but within a transaction.

Deciding when to perform a rotation is done based on local balance information omitted from the pseudocode. This technique was introduced in [6] and works as follows. *lefth* (*righth*) is a local variable to each node keeps track of the estimated height of the left (right) subtree. *localh* (also a local variable to each node) is always 1 larger than the maximum value of *lefth* and *righth*. If the difference between *lefth* and *righth* is greater than 1 then a rotation is triggered. After the rotation these values are updated as indicated by a dedicated function (line 68). Since these values are local to the node the estimated heights of the subtrees might not always be accurate. The propagate operation (described in the next paragraph) is used to update the estimated heights. Using the propagate operation and local rotations, the tree is guaranteed to be eventually perfectly balanced as in [6, 7].

Propagate This is a maintenance operation performed frequently on every node in the tree in order to keep balance information updated. Due to concurrency, it might propagate the correct height information at any given time. Fortunately the tree can still be eventually perfectly balanced, all that is required is that the algorithm ensures that a node knows when it has an empty subtree (i.e. when $node.\ell$ is $\perp$, *node.lefth* must be 0). This is insured because when a new node is added to the tree it has *lefth* and *righth* set to 0 and these values are updated when a node is removed or a rotation takes place. Each propagate operation is performed as a single transaction at each node by first traveling to the left and right child nodes, checking their *localh* values and using these values to update *lefth*, *righth*, and *localh* of the parent node. The propagate operation only modifies these 3 values which are never accessed in abstract transactions. This allows the propagate operation to execute without ever causing a conflict with an abstract transaction.

Limitations Unfortunately, spreading rotations and modifications into distinct transactions still does not allow insert/delete/contains operations that are being performed on separate keys to execute concurrently. Consider a delete operation that deletes a node at the root. In order to remove this node a successor is taken from the bottom of the tree to that it becomes the new root. This now creates a point of contention at the root and where the successor was removed. Every concurrent transaction that accesses the tree will have a read/write conflict with this transaction. Below we discuss how to address this issue.

3.2 Decoupling node deletion

The transaction-friendly binary search tree exploits logical deletion to further reduce the amount of transaction conflicts. This two-phase deletion technique has been previously used for memory management like in [25], for example, to reduce locking in database indexes. Each node has a *deleted* flag, initialized to false when the node is inserted into the tree. First, the delete phase consists of removing the given key k from the abstraction—it logically deletes a node by setting a *deleted* flag to true (line 80). Second, the remove phase physically deletes the node from the tree to prevent it from growing too large. Each of these are performed as a separate transaction.

The deletion decoupling reduces conflicts by two means. First, it spreads out the two deletion phases in two separate transactions, hence reducing the size of the delete transaction. Second, deleting logically node i simply consists in setting the *deleted* flag to true (line 80), thus avoiding conflicts with concurrent set abstract transactions that have parsed i.

3.3 Find, contains, insert, delete and remove

Find The find operation is a helper function called implicitly by other functions within a transaction, thus it is never called explicitly by the application programmer. This operation looks for a given key k by parsing the tree similarly to a sequential code. At each node it goes right if the key of the node is larger than k (line 20), otherwise it goes left (line 21). Starting from the root it continues until it either finds a node with k (line 18) or until it reaches a leaf (line 22) returning the node (line 24). Notice that if it reaches a leaf, it has performed a transactional read on the child pointer of this leaf (lines 20–21),

ensuring that the value is $\perp$ so that the node remains a leaf throughout the transaction ensuring that some other concurrent transaction will not insert a node with key k.

Contains The contains operation first executes the find starting from the root, this returns a node (line 28). If the key of the node returned is equal to the key being searched for it performs a transactional read of the *deleted* flag (line 32), if the flag is false the operation returns true, otherwise it returns false. If the key of the returned node is not equal to the key being searched for then a node with the key being searched for is not in the tree and false is returned (lines 30 and 34).

Insertion The insert(k, v) operation uses the find procedure that returns a node (line 38). If a node is found with the same *key* as the one being searched for then the *deleted* flag is checked using a transactional read (line 40). If the flag is false then the tree already contains k and false is returned (lines 42 and 49). If the flag is true then the flag is updated to false (line 41) and true is returned. Otherwise if the *key* of the node returned is not equal to k then a new node is allocated and added as the appropriate child of the node returned by the find operation (lines 44-47). Notice that only in this final case does the operation modify the structure of the tree.

Deletion The execution of the deletion operation is done as follows: Similar to the other operations, the find procedure is used in order to locate the node to be deleted (line 74). A transactional read is then performed on the *deleted* flag (line 78). If *deleted* is true then the operation returns false (line 79 and line 82), if *deleted* is false it is set to true (line 80) and the operation returns true. If the find procedure does not return a node with the same *key* as the one being searched for then false is returned (line 76 and 82). Notice that this operation never modifies the tree structure.

Consequently, the insert/delete/contains operations can only conflict with each other in two cases.

1. Two insert/delete/contains operations are being performed concurrently on some key k and a node with key k exists in the tree. Here (if at least one of the operations is an insert or delete) there will be a read/write conflict on the node's *deleted* flag. Note that there will be no conflict with any other concurrent operation that is being done on a different key.

2. An insert that is being performed for some key k where no node with key k exists in the tree. Here the insert operation will add a new node to the tree, and will have a read/write conflict with any operation that had read the pointer when it was $\perp$ (before it was changed to point to the new node).

Algorithm 2 Optimizations to the Transaction-Friendly Binary Search Tree

```
1:  State of node n:
2:    node the same record with an extra field:
3:      rem ∈ {true, false} indicate whether
4:        physically deleted, initially false

5:  remove(parent, left-child)_p:
6:    transaction {
7:      if read(parent.rem) then
8:        return false
9:      if left-child then
10:       n ← read(parent.ℓ)
11:     else
12:       n ← read(parent.r)
13:     if n = ⊥ then
14:       return false
15:     if ¬read(n.deleted) then
16:       return false
17:     if (child ← read(n.ℓ)) ≠ ⊥ then
18:       if (child ← read(n.r)) ≠ ⊥ then
19:         return false
20:     if left-child then
21:       write(parent.ℓ, child)
22:     else
23:       write(parent.r, child)
24:     write(n.ℓ, parent)
25:     write(n.r, parent)
26:     write(n.rem, true)
27:     update-balance-values()
28:   } // current transaction tries to commit
29:   return true

30: find(k)_p:
31:   next ← root
32:   rem ← true
33:   while 1 do
34:     while 1 do
35:       parent ← curr
36:       curr ← next
37:       val ← curr.k
38:       if val = k then
39:         if ¬(rem ← read(curr.rem)) then
40:           break
41:       if val > k then next ← uread(curr.r)
42:       else next ← uread(curr.ℓ)
43:       if next = ⊥ then
44:         if ¬(rem ← read(curr.rem)) then
45:           if val > k then
46:             next ← read(curr.r)
47:           else next ← read(curr.ℓ)
48:           if next = ⊥ then break
49:         else
50:           if val ≤ k then
51:             next ← uread(curr.r)
52:           else next ← uread(curr.ℓ)
53:     if curr.k > par.k then
54:       tmp ← read(par.r)
55:     else tmp ← read(par.ℓ)
56:     if curr = tmp then break
57:     else next ← curr
58:       curr ← parent
59:   return curr

60: right_rotate(parent, left-child)_p:
61:   transaction {
62:     if read(parent.rem) then
63:       return false
64:     if left-child then
65:       n ← read(parent.ℓ)
66:     else
67:       n ← read(parent.r)
68:     if n = ⊥ then
69:       return false
70:     ℓ ← read(n.ℓ)
71:     if ℓ = ⊥ then
72:       return false
73:     ℓr ← read(l.r)
74:     r ← read(n.r)
75:     // allocate a node called new
76:     new.key ← n.key
77:     new.ℓ ← ℓr
78:     new.r ← r
79:     write(ℓ.r, new)
80:     write(n.rem, true)
81:     if left-child then
82:       write(parent.ℓ, ℓ)
83:     else
84:       write(parent.r, ℓ)
85:     update-balance-values()
86:   } // current transaction tries to commit
87:   return true
```

Removal Removing a node that has no children is as simple as unlinking the node from its parent (lines 97–99). Removing a node that has 1 child is done by just unlinking it from its parent, then linking its parent to its child (also lines 97–99). Each of these removal procedures is a very small transaction, only performing a single transactional write. This transaction conflicts only with concurrent transactions that read the link from the parent before it is changed.

Upon removal of a node i with two children, the node in the tree with the immediately larger key than i's must be found at the bottom of the tree. This performs reads on all the way to the leaf and a write at the parent of i, creating a conflict with any operation that has parsed this node. Fortunately, in practice such removals are not necessary. In fact only nodes with no more than one children are removed from the tree (if the node has two children, the remove operation returns without doing anything, cf. line 95). It turns out that removing nodes with no more than one children is enough to keep the tree from growing so large that it affects performance.

The removal operation is performed by the maintenance thread, as it is traversing the tree performing rotation and propogate operations it also checks for logically deleted nodes to be removed.

Limitations The parse phase of most functions is prone to false-conflicts, as it comprises read operations that do not actually need to return values from the same snapshot. Specifically, by the time a parse transaction reaches a leaf, the value it read at the root likely no longer matters, thus a conflict with a concurrent root update could simply be ignored. Nevertheless, the standard TM interface forces all transactions to adopt the same strongest semantics prone to false-conflicts [17]. Below we discuss how to extend the basic TM interface to cope with such false-conflicts.

3.4 Optional improvements

In previous sections, we have described a transaction-friendly tree that fulfills the standard TM interface [11] for the sake of portability across a large body of research work on TM. Now, we propose to further reduce aborts related to the rotation and the find operation at the cost of an additional lightweight read operation, uread, that breaks this interface. These optimization are thus usable only in TM systems providing additional explicit calls [2, 15, 21] and do not aim at replacing but complementing the previous algorithm to preserve its portability. These optimizations complementing Algorithm 1 are depicted in Algorithm 2, they do not affect the existing contains/insert/delete operations.

By speeding up the find operation, these optimization improve the performance of all the other operations, namely contains/insert/delete.

Lightweight reads The key idea is to avoid validating superfluous read accesses when an operation parses the tree structure. This idea has been exploited by elastic transactions that use a bounded buffer instead of a read set to validate only immediately preceding reads, thus implementing a form of hand-over-hand locking transaction for search structure [16]. We could have used different extensions to implement these optimizations. DSTM [21] proposes early release to force a transaction stop keeping track of a read set entry. Alternatively, the current distribution of TinySTM [15] as well as View Transactions [2] comprise lightweight reads that do not record anything in the read set. While we could have used any of these approaches to increase concurrency we have chosen the unit-loads of TinySTM, hence the name uread. This uread returns the most recent value written to memory by a committed transaction by potentially spin-waiting on the location until it stops being concurrently modified.

A first interesting result, is that the read/write set sizes can be kept at a size of $O(k)$ instead of the $O(k \log n)$ obtained with the previous tree algorithm, where k is the number of nested contains/insert/delete operations nested in a transaction. The reasonning behing this is as follow: Upon success, a contains only needs to ensure that the node it found is still in the tree when the transaction commits, and can ignore the state of other nodes it had traversed. Upon failure, it only needs to ensure that the node i it is looking for is not in the tree when the transaction commits, this requires to check whether the pointer from the parent that would point to i is $\perp$ (i.e. this pointer should be in the read set of the transaction and its value is $\perp$). In a similar vein, insert and delete only need to validate the position in the tree where they aimed at inserting or deleting. Therefore, contains/insert/delete only increases the size of the read/write set by a constant instead of a logarithmic amount.

It is worth mentioning that ureads have a further advantage over normal reads other than making conflicts less likely: Transactional reads are more expensive to perform than unit reads. This is because in addition to needing to store a list keeping track of the reads done so far, a TM that uses invisible reads will need to perform validation of the read set with a worst case cost of $O(s^2)$, where s is the size of the read set, and a TM that uses visible reads will perform a modification to shared memory for each read.

Rotations Rotations remain conflict-prone in Algorithm 1 as they incur a conflict when crossing the region of the tree parsed by a contains/insert/delete operation. If ureads are used in the contains/insert/delete operations then rotations will only conflict with these operations if they finish at one of the two nodes that are rotated by rotation operation (for example

in Figure 2(a) this would be the node i or j). A rotation at the root will only conflict with a contains/insert/delete that finished at (or at the rotated child of) the root, any operations that travel further down the tree will not conflict.

Figure 2(c) displays the result of the new rotation that is slightly different than the previous one. Instead of modifying j directly, j is unlinked from its parent (effectively removing it from the tree, lines 82–84) and a new node j' is created (line 75), taking j's place in the tree (lines 82–84). During the rotation j has a removed flag that is set to true (line 80), letting concurrent operations know that j is no longer in the tree but its deallocation is postponed. Now consider a concurrent operation that is traversing the tree and is preempted on j during the rotation. If a normal rotation is performed the concurrent operation will either have to backtrack or the transaction would have to abort (as the node it is searching for might be in the subtree A). Using the new rotation, the preempted operation will still have a path to A.

Find, contains, delete and remove The interesting point for the find operation is that the search continues until it finds a node with the *removed* flag set to false (line 39 and 44). Once the leaf or a node with the same key as the one being searched for is reached a transactional read is performed on the *removed* flag to ensure that the node is not removed from the tree (by some other operation) at least until the transaction commits. If *removed* is true then the operation continues traversing the tree, if *removed* is false then the correct node has been found. Next, if the node is a leaf, a transactional read must be performed on the appropriate child pointer to ensure this node remains a leaf throughout the transaction (lines 46–47). If this read does not return $\perp$ then the operation continues traversing the tree. Otherwise the operation then leaves the nested while loop (lines 40 and 48), but the find operation does not return yet.

One additional transactional read must be performed in order to ensure safety. This is the read of the parent's pointer to the node about to be returned (lines 54–55). If this read does not return the same node as found previously, the find operation continues parsing the tree starting from the parent (lines 57–58). Otherwise the process leaves the while loop (line 56) and the node is returned (line 59).

The advantage of this updated find operation is that ureads are used to parse the tree, it only uses transactional reads to ensure atomicity when it reaches what is suspected to be the last node it has to parse. The original algorithm exclusively uses transactional reads to parse the tree and because of this, modifications to the structure of the tree that occur along the parsed path cause conflicts, which do not occur in the updated algorithm. The contains/insert/delete operations themselves are identical in both algorithms.

Removals The remove operation requires some modification in order to ensure safety when using ureads during the parse phase. Normally if a contains/insert/delete operation is preempted on a node that is removed then that operation will have to backtrack or abort the transaction. This can be avoided as follows. When a node is removed, its left and right child pointers are set to point to its previous parent (lines 24-25). This allows a preempted operation a path back to the tree. The removed node also has its *removed* flag set to true (line 26) letting preempted operations know it is no longer in the tree (the node is left to be freed later by garbage collection).

Theorem 1 *The* insert, contains *and* delete *operations of Algorithm 1 with optimizations from Algorithm 2 are linearizable.*

The proof of theorem 1 is available in a companion technical report whose URL has been hidden for the sake of anonymity.

	Insert	Delete	Contains	Remove	Rotate
Insert	$\langle G, P, P\rangle$	$\langle G, P, P\rangle$	$\langle G, P, P\rangle$	$\langle \emptyset, G, P\rangle$	$\langle \emptyset, G, P\rangle$
Delete		$\langle G, P, P\rangle$	$\langle G, P, P\rangle$	$\langle \emptyset, G, P\rangle$	$\langle \emptyset, G, P\rangle$
Contains			$\langle N, N, N\rangle$	$\langle \emptyset, G, P\rangle$	$\langle \emptyset, G, P\rangle$
Remove				$\langle \emptyset, G, G\rangle$	$\langle \emptyset, G, G\rangle$
Rotate					$\langle \emptyset, G, G\rangle$

Table 2: Tree operations and their conflicts ⟨ Red-black/AVL tree, TF-tree, Optimized TF-Tree ⟩. G is a global conflict, meaning the operations can have a possible conflict anywhere. P is a partial conclift, meaning the operations might have a conflict but only if (for the case of two non-maintenance operations) they both have the same final node in the traversal or if (for the case of 1 maintenance and 1 non-maintenance operation) the final node in the traversal is one that is modified by the maintenance operation, N means the operations will never conflict, $\emptyset$ means not applicable for this tree

4 Experimental Evaluation

We experimented our library by integrating it in (i) a micro-benchmark integer set to get a precise understanding of the performance causes and in (ii) the tree-based vacation reservation system that is part of the STAMP suite [8] and whose

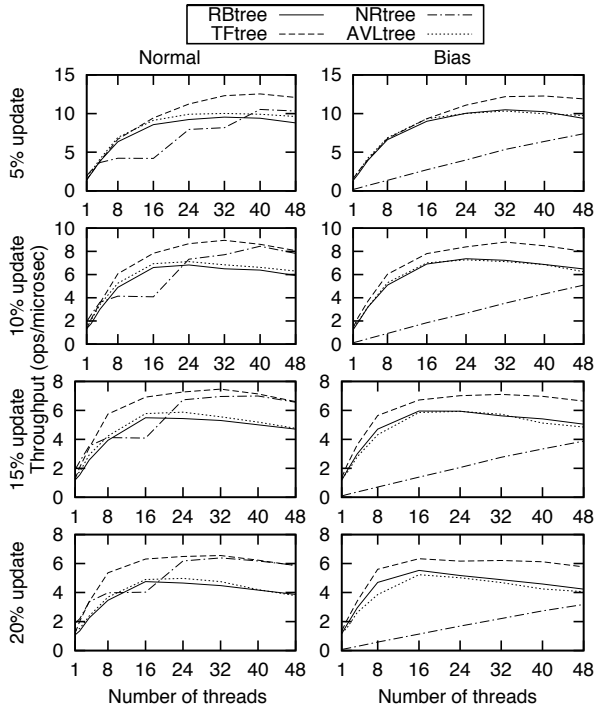


Figure 3: Comparing the AVL-tree, the red-black tree, the no-restructuring tree against the transaction-friendly tree on an integer set micro-benchmark with from 5% **(top)** to 20% updates **(bottom)**

runs sometimes exceed half an hour. The machine used for our experiments is a four AMD Opteron 12-core Processor 6172 at 2.1 Ghz with 32 GB of RAM running Linux 2.6.32, thus comprising 48 cores in total.

4.1 Testbed choices

We evaluate our tree against well-engineered tree algorithms especially dedicated to transactional workloads. The red-black tree is a mature implementation developed and improved by expert programmers from Oracle Labs (formerly Sun Microsystems) and others to show good performance of transactional memory in numerous papers [12, 14, 15, 21]. The observed performance is generally scalable when contention is low, most of integer set benchmarks on which they are tested consider the ratio of attempted updates instead of effective updates. To avoid the misleading (attempted) update ratios that capture the number of update operation calls, we consider the *effective* update ratios that does not increase if an update operation, like remove, is called but fails in finding its parameter value thus failing as well in modifying the data structure.

The AVL tree we evaluate as well as the aforementioned red-black tree are also part of STAMP [8] that we also experiment here. As mentioned before one of the main refactoring of this red-black tree implementation is to avoid the use of sentinel nodes that would produce false-conflicts within transactions. This improvement could be considered a first-step towards obtaining a transaction-friendly binary search tree, however, the modification-restructuring, which remains tightly coupled, prevents scalability to high levels of parallelism.

4.2 Biased workloads and the effect of restructuring

In this section, we evaluate the performance of our transaction-friendly tree on an integer set micro-benchmark providing remove, insert, and contains operations, similarly to the benchmark used to evaluate state-of-the-art TM algorithms [15]. We implemented two set libraries that we added to microbench: our non-optimized transaction-friendly tree and a baseline tree that is similar but never rebalances the structure whatever modifications occur. Figure 3 depicts the performance obtained from four different binary search trees: the red-black tree (RBtree), our transaction-friendly tree (TFtree), the no-restructuring tree (NRtree) and the AVL tree (AVLtree).

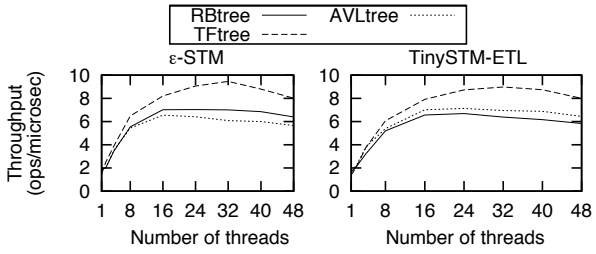


Figure 4: The transaction-friendly library running with **(left)** another TM library ($\mathcal{E}$-STM) and with **(right)** the previous TM library in a different configuration (TinySTM-ETL, i.e., with eager acquirement)

The performance is expressed as the number of operations executed per microsecond. The integer set is initialized with 2^{12} elements and the update ratio varies between 5% and 20%.

On both the normal and biased workloads, the transaction-friendly tree scales with the number of threads. The no-restructuring tree performance drops to a linear shape under the biased workload as expected: as it does not rebalance, the complexity increases with the length of the longest path from the root to a leaf that, in turn, increases with the number of performed updates. In contrast, the transaction-friendly tree can only be unbalanced during a transient period of time which is too short to affect the performance even under biased workloads.

The transaction-friendly tree improves both the red-black tree and the AVL tree performance by up to 1.5× and 1.6×, respectively. The transaction-friendly tree is less prone to contention than AVL and red-black trees, which both share similar performance penalties due to contention.

4.3 Portability to other TM algorithms

The transaction-friendly tree is an inherently efficient data structure that is portable to any TM systems. It fulfills the TM interface standardized in [11] and thus does not require the presence of explicit escape mechanisms like light-read [2], early release [21], snap [10] to avoid extra TM bookkeeping (our uread optimization being optional). Nor does it require high-level conflict detection, like open nesting [3, 26, 27] or transactional boosting [20]. Such improvements rely on explicit calls or user-defined abstract locks, and are not supported by existing TM compilers [11] which limits their portability. To make sure that the obtained results are not biased by the underlying TM algorithm, we evaluated the trees on top of another TM library, $\mathcal{E}$-STM, and on top of a very different TM design than the one used so far: with eager-acquirement.

The obtained results, depicted in Figure 4 look similar to the ones obtained with TinySTM-CTL (Figure 3) in that the transaction-friendly tree executes significantly faster than other trees for all TM settings. This suggests that the improvement of transaction-friendly tree is potentially independent from the TM system used. A more detailed comparison of the improvement obtained using elastic transactions on red-black trees against the improvement of replacing the red-black tree by the transaction-friendly tree is depicted in Figure 5(left). It clearly shows that the elastic improvements is negligible compared to the transaction-friendly tree one (be it optimized or not).

This confirms our intuition that decoupling each update into an abstract transaction and multiple structural transactions increases performance, even though it may seem to contradict other results that show short transactions penalizing performance [35]. This later observation argued that fine-grained transactions are more costly than coarser grained transactions when the cost of starting and committing transactions becomes non-negligible. It is noteworthy, however, that such an observation was made on very short transactions that comprise only one load and two stores on average, hence much shorter than the abstract transactions that we have used here.

4.4 Reusability for specific application needs

We illustrate the reusability of the transaction-friendly tree by composing remove and insert from the existing interface to obtain a new atomic and deadlock-free move operation. Reusability is appealing to simplify concurrent programming by making it modular: a programmer can reuse an existing library without having to understand its synchronization internals. While reusability of sequential programs is straightforward, concurrent programs can generally be reused only if the programmer understand how each element is protected. For example, reusing a library can lead to deadlocks if shared data are locked in a different order than what is recommended by the library. Additionally, a lock-striping library may not conflict with a concurrent program that locks locations independently even though they protect common locations, thus leading to inconsistencies.

Algorithm 3 Move operation

```
1: move(old_key, new_key)_p:
2:   transaction {
3:     ret ← false
4:     if ¬contains(new_key) then
5:       if v ← delete(old_key) then
6:         insert(new_key, v)
7:         ret ← true
8:   } // current transaction tries to commit
9:   return ret
```

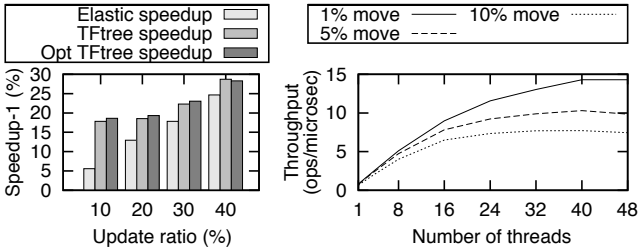


Figure 5: Relaxed transactions used by the red-black tree vs. classical transactions used by the transaction-friendly tree **(left)** and performance of the extended associated array abstraction implemented by the transaction-friendly tree **(right)**

Figure 5(right) indicates the performance on workloads comprising from 1% to 10% of move operations among 10% update operations and 90% contains operations. The performance decreases as more move operations execute, because a move protects more elements in the data structure than a simple insert or remove operation and during a longer period of time. Cederman and Tsigas propose an efficient lock-free move alternative [9] but it is unclear how someone could port it to TM systems.

4.5 The vacation travel reservation application

We experiment our optimized library tree with a realistic travel reservation application from the STAMP suite [8], called vacation. This application is suitable for evaluating concurrent binary search tree in realistic settings as it represents a database with four tables implemented as tree-based directories (cars, rooms, flights, and customers) accessed concurrently by client transactions.

Figure 6 depicts the execution time of the STAMP vacation application building on the built-in red-black tree library, our optimized transaction-friendly tree, and the baseline no-restructuring tree. We added the speedup obtained with each of these tree libraries over the performance of bare sequential code of vacation without synchronization. (A tree library outperforms the sequential tree when its speedup exceeds 1.) The chosen workloads are the two default configurations ("low contention" and "high contention") taken from the STAMP release, with the default number of transactions, 8× more transactions than by default and 16× more, to increase the duration and the contention of the benchmark without using more threads than cores.

Vacation executes always faster on top of our transaction-friendly tree than on top of its default red-black tree. For example, the transaction-friendly tree improves performance by up to 1.3× with the default number of transactions and to 3.5× with 16× more transactions. The reason of this is twofold:

1. In the built-in red-black tree, if an operation parses a location that is being deleted, then the parse and the deletion conflict. Note that it is more restrictive than the unreachability of deleted nodes guaranteed by the transaction-friendly tree, as the latter tree decouples the logical and physical deletion of a node (as we explained in Figure 2) to allow a co-located parse to resume. In both cases, no operation can parse and reach a deleted node, however, the difference applies to parse operations that got preempted at the node being deleted.

2. The balance requirement of the red-black tree is not flexible, even though its level of imbalance can be much higher than the AVL tree one: the longest path from the root to a leaf in the red-black tree can be twice as long as the shortest one. This inflexibility requires that the restructuring gets triggered immediately after the imbalance threshold is reached. Hence, a modification of the tree is always coupled with the imbalance check and the potential rotations, in the same transaction. In contrast, the restructuring of the transaction-friendly tree does not have such long transactions and is flexible, thus allowing concurrent transactions to proceed on a tree whose imbalance level is transiently higher than expected.

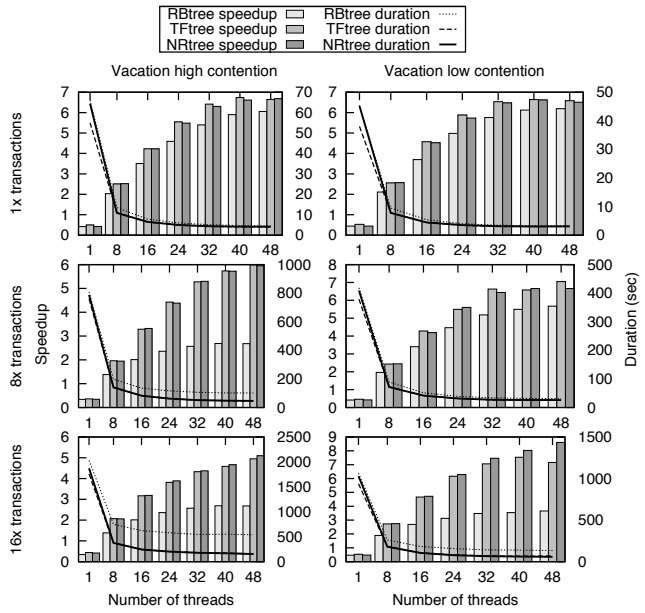


Figure 6: The speedup (over single-threaded sequential) and the corresponding duration of the vacation application built upon the red-black tree (RBtree), the transaction-friendly tree (TFtree) and the no-restructuring tree (NRtree) on **(left)** high contention and **(right)** low contention workload, and with **(top)** the default number of transaction, **(middle)** 8× more transactions and **(bottom)** 16× more transactions

Finally, we observe that vacation presents very good performance on top of the no-restructuring tree library. In rare cases, the transaction-friendly tree outperforms the no-restructuring tree probably because the no-restructuring does not physically delete nodes from the tree, thus leading to a larger tree than the abstraction. The performance is comparable to the transaction-friendly tree one and exceeds it sometimes, especially with 16× the default number of transactions. The reason is that rotations are the bottleneck in highly contented workload as they induce frequent aborts; unlike the transaction-friendly and red-black trees that rotate, the no-restructuring tree simply ignores the imbalances.

5 Other Transaction-Friendly Data Structures

We discuss how to derive other transaction-friendly data structures.

Skip lists Skip lists ensure 2-base logarithmic access time complexity by guaranteeing that its top-level is $O(\log n)$, where n is the number of elements it contains. Hence the top-level must be adapted to guarantee this invariant when n changes significantly. Moreover, new elements are inserted by creating a new node with a level ℓ (lower than the structure top-level) tossed with a probability whose distribution decreases as values get larger. Each node must be linked to the data structure from the bottom-most level 1 to its own level ℓ by modifying a data structure pointer at each of these ℓ levels, however, the element starts being reachable as soon as its node is linked at the bottom-most level 1. Linking it at the remaining $\ell - 1$ levels is only necessary to maintain the logarithmic complexity invariant. Hence in an optimistic execution, one can simply defer the linking of the 2 to $\ell - 1$ levels and the top-level adjustment to diminishes conflicts and improve efficiency.

Hash tables Hash tables record a set of values whose hash returns an address in an array. If two values share the same hash they are stored in the same bucket pointed by the corresponding address—this bucket is generally implemented using a sorted linked list [24]. The hash table guarantees constant access time complexity as long as its *load*, the number of its elements divided by the number of its buckets remains constant. If the load exceeds some threshold a *resize* operation must restructure the hash table by adding new buckets and rebalancing the values among buckets to reduce the load accordingly. Such a resize could be delayed especially in transactions where the step complexity may be much higher than expected due to contention.

6 Related Work

Aside from the optimistic concurrency control context, various relaxed balanced trees have been proposed. The idea of decoupling the update and the rebalancing have been originally proposed in [18] and has been applied to AVL trees in [22,29] and to red-black trees in [28]. Manber and Ladner propose a lock-based tree whose rebalancing is the task of separate maintenance threads running with a lower priority than other threads [23]. Bougé et al. in [6] propose a local rotation locking only the nodes being rotated. The combination of local rotations executed by different threads self-stabilize to a tree where no nodes is marked for removals. The main objective of these techniques is still to keep the tree depth low enough for the lock-based operations to be efficient. Such solutions do not apply to speculative operations due to aborts.

Ballard [4] proposes a relaxed red-black tree insertion well-suited for transactions. When an insertion unbalances the red-black tree it marks the inserted node rather than rebalancing the tree immediately. Another transaction encountering the marked node must rebalance the tree before restarting. The relaxed insertion was shown generally more efficient than the original insertion when run with DSTM [21] on 4 cores. Even though the solution limits the waste of effort per aborting rotation, it increases the number of restarts per rotation. By contrast, our local rotation does not require the rotating transaction to restart, hence benefiting both insertions and removals.

Bronson et al. [7] introduce an efficient object-oriented binary search tree implementation. The algorithm uses underlying time-based TM principles to achieve good performance, however, its operations cannot be encapsulated within overly conservative transactions. For example, a key optimization of this tree distinguishes whether a modification at some node i grows or shrinks the subtree rooted in i. A conflict involving a growth could be ignored as no descendant are removed and a search preempted at node i will safely resume in the resulting subtree. Such an optimization is not possible using TMs that track conflicts between read/write accesses to the shared memory. This implementation choice results in higher performance by avoiding the TM overhead, but limits reusability due to the lack of bookkeeping. For example, a programmer willing to implement a size operation would need to explicitly clone the data structure to disable the growth optimization. Therefore, the programmer of a concurrent application that builds upon this binary search tree library must be aware of the synchronization internals of this library (including the growth optimization) to reuse it.

Felber et al. [16] present elastic transactions, a relaxed transaction model that guarantees reusability. In the companion technical report, the red-black tree library from Sun Microsystems has been shown executing efficiently on top of an implementation of the elastic transaction model, $\mathcal{E}$-STM. The implementation idea consists of encapsulating the (i) operations that locate a position in the red-black tree (like insert, contains, remove) into an *elastic* transaction to increase concurrency and (ii) other operations, like size, into a regular transaction. This approach is orthogonal to ours as it aims at improving the performance of the underlying TM, independently from the data structure, by introducing relaxed transactions. Hence, although elastic transactions can cut themselves upon conflict detection, the resulting $\mathcal{E}$-STM, still suffers from congestion and wasted work when applied to non-transaction-friendly data structures. The results presented in Section 4.4 confirm that the elastic speedup is limited on a classical red-black tree.

7 Conclusion

Transaction-based data structures are becoming a bottleneck in multicore programming, playing the role of synchronization toolboxes a programmer can rely on to write a concurrent application easily. To exploit adequately multicore architectures with transactions, traditional data structures require a code refactoring to diminish the effect of contention and make them efficient. By contrast with traditional pessimistic concurrency control, it allows the programmer to directly observe the impact of contention as part of the step complexity because conflicts potentially lead to subsequent speculative re-executions.

We have illustrated, through a binary search tree use-case, how one can exploit this information to improve the performance through code refactoring of traditional data structures. Besides this example, we envision substantial benefit for other existing data structures, like skip lists, linked lists and hash tables. Evaluating this benefit may help in choosing the best *concurrency-friendly data structure* [34] between data structures that have high access complexity but localized restructuring like linked lists, and data structures that rely on costly restructuring to guarantee constant-time accesses, like hash tables.

References

[1] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 146, pages 263–266, 1962.

[2] Yehuda Afek, Adam Morrison, and Moran Tzafrir. Brief announcement: view transactions: transactional model with relaxed consistency checks. In *PODC*, pages 65–66. ACM, 2010.

[3] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. In *PPoPP*, pages 151–162. ACM, 2009.

[4] Lucia Ballard. Conflict avoidance: Data structures in transactional memory. Master's thesis, Brown University, May 2006.

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

[6] Luc Bougé, Joaquim Gabarro, Xavier Messeguer, and Nicolas Schabanel. Height-relaxed avl rebalancing: A unified, fine-grained approach to concurrent dictionaries. Technical Report RR1998-18, LIP, ENS Lyon, March 1998.

[7] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *PPoPP*, pages 257–268. ACM, 2010.

[8] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, September 2008.

[9] Daniel Cederman and Philippas Tsigas. Supporting lock-free composition of concurrent data objects. In *PPoPP*, pages 339–340. ACM, 2010.

[10] Christopher Cole and Maurice Herlihy. Snapshots and software transactional memory. *Sci. Comput. Program.*, 58:310–324, December 2005.

[11] Intel Corporation. Intel transactional memory compiler and runtime application binary interface, May 2009.

[12] D. Dice, O. Shalev, , and N. Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.

[13] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21:966–975, November 1978.

[14] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54(4):70–77, 2011.

[15] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, pages 237–246. ACM, 2008.

[16] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *DISC*, LNCS, pages 93–107. Springer-Verlag, 2009.

[17] Vincent Gramoli and Rachid Guerraoui. Brief announcement: Transaction polymorphism. In *SPAA*, pages 311–312. ACM, Jun 2011.

[18] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21. IEEE, 1978.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[20] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*. ACM, 2008.

[21] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*. ACM, 2003.

[22] J. L. W. Kessels. On-the-fly optimization of data structures. *Comm. ACM*, 26:895–901, 1983.

[23] Udi Manbar and Richard E. Ladner. Concurrency control in a dynamic search structure. *ACM Trans. Database Syst.*, 9:439–455, September 1984.

[24] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.

[25] C. Mohan. Commit-LSN: a novel and simple method for reducing locking and latching in transaction processing systems. In *VLDB*, pages 406–418. Morgan Kaufmann Publishers Inc., 1990.

[26] J. Eliot B. Moss. Open nested transactions: Semantics and support. In *WMPI*, 2006.

[27] Yang Ni, Vijay Menon, Ali-Reza Abd-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP*. ACM, 2007.

[28] O. Nurmi and E. Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *PODS*, pages 192–198. ACM, 1991.

[29] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *PODS*, pages 170–176. ACM, 1987.

[30] Victor Pankratius and Ali-Reza Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *SPAA*, pages 43–52. ACM, 2011.

[31] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *PPoPP*, pages 47–56. ACM, 2010.

[32] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54:76–84, 2011.

[33] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213. ACM, 1995.

[34] Herb Sutter. Choose concurrency-friendly data structures. *Dr. Dobb's Journal*, June 2008.

[35] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA*, pages 265–274. ACM, 2008.

A Correctness Proof

There are two parts in the proof. First we have to ensure the structure of the tree is always a valid binary search tree. This is important because in a binary search tree at any time there is exactly one correct location for a key k, the term used for such a tree is *valid binary tree*. A binary tree that does not have the previous property is simply called a *binary tree*. Second we have to show the insert, delete, and contains operations are linearizable.

It is important to remember for this proof that when a transaction commits the transactional reads and writes appear as if they have happened atomically and that unit-read operations only return values from previously committed transactions (or the value written by the current transaction, if the transaction has written to the location being read).

Another important part of this proof is the way the tree is first created. It is created with a root node with key ∞ so that all nodes will always be on its left subtree. This node will always be the root (i.e. it will not be modified by rotate or removal operations), this makes simpler operations and proofs.

A.1 Definitions

[V: Couldn't we put this paragraph in the alg. description instead of this section?] First some definitions.

A.1.1 Binary Trees

Each node has two boolean state variables. When the variable *deleted* is false it entails that the value of *node.key* is in the set represented by the tree. When it is *true* it entails that the value of *node.key* is not in the set represented by the tree. When the variable *removed* is false it entails that the node exists in the tree (meaning a path exists from the root of the tree to the node). When it is *true* it entails that the node does not exist in the tree (meaning no path exists from the root of the tree to the node).

In the proof we will use the phrase *a node can reach a range of keys* which is explained here. Take the root, from this node there is a path to any node in the tree meaning any key that is in the set is reachable from the root. Furthermore for any key that is not in the tree the root has a path to where it would be inserted (the leaf that would be its parent). This means that the root with key k node has a range $[-\infty, \infty]$. Now its left child has range $[-\infty, k]$ and its right child has range $[k, -\infty]$. Or for example consider some node with range $(10, 20]$ and key 14. Its left child will have a range $(10, 14)$ and its right child will have a range $(14, 20]$. have a path to it.

The phrase *a node n has a path to a range of keys at least as large as some other node n'* is also used in the proof. It means that every key that is in the range of n' is also in the range of n (and possibly some more). For example any node will have a path to a range of keys at least at large as its left child (in fact it has the exact range of its left and right child combined).

A.1.2 Set operations

Here traditional operations on the set are defined in the context of transactions. It is important to remember that the TM guarantees a linearization of transactions.

The following definitions are used in defining the operations. Saying a key k is in (not in) the set before the committal of a transaction T means that if some transaction $T1$ performs a contains operation on key k and is serialized as the transaction immediately before T, $T1$ would return true (false).

Saying a key k is in (not in) the set after the committal of a transaction T means that if some transaction $T2$ performs a contains operation on key k and is serialized as the transaction immediately after T, $T2$ would return true (false).

delete For transaction that commits a successful delete operation of key k, before the commit k was in the set and afterwards k is not in the set. For transaction that commits a failed delete operation of key k, before and after the commit k was not in the set.

insert For transaction that commits a successful insert operation of key k, before the commit k was not in the set and after the commit k is in the set. For transaction that commits a failed insert operation of key k, before and after the commit k is in the set.

contains For transaction that commits a successful contains operation of key k, before and after the commit k was in the set. For transaction that commits a failed contains operation of key k, before and after the commit k was not in the set.

A.2 Lemmas and Theorems

Lemma 1 *A node has at most one parent whose value of removed is* false*.*

Proof. Assume by contradiction that a node has more then one parent whose value of *removed* is false. There are three operation where a node can be given a new parent. First during the insert operations on lines 46–47, but this is a new node so before this line it has no parent. Second during the remove operation on lines 21–23, but by line 8 the other parent has *removed* set to true. Third during the *right-rotate* operation the node l gets a new parent on lines 82–84, but by line 80 the other parent has *removed* set to true. Also during the right_rotate operation the node $n2$ gets l as a parent (line 79), but since it is a new node it has no other parent. (This holds for the left_rotate operation by symmetry) Given this, a node will have at most one parent with *removed* = false. □

Lemma 2 *A node with removed* = false *only has paths from it to other nodes with removed* = false*.*

Proof. This proof is by induction on the number j of operations done on a node with key k.

The base case is when a new node is created and added to the tree. This can happen in the insert operation. During the operation a new node is created with no children and for itself it has *removed* = false (line 44) and the proof holds.

Now for the induction step from $j = m - 1$ to $j = m$, this could be a contains, delete, insert, remove, or rotate operation. First note that the contains and delete operations do not change any children pointers. If this is an insert operation then at $j = m - 1$ *left* or *right* must be $\perp$ (line 48 of the find operation) and the proof obviously still holds. If this is a remove operation, then the child of the node is being removed. This means that the new child will either be $\perp$ or the child of the child (lines 17–23), but by induction these nodes have *removed* = false. By symmetry this holds for the right and left children. Otherwise this is a rotate operation. First consider right rotations. By induction we know that node n only has paths to nodes with *removed* = false. After the rotation node l points to nodes that had paths from n before the rotation as well as $n2$ (line 79) which has *removed* = false. $n2$ points to nodes that had paths from n before the rotation (lines 77–78). By symmetry this holds for left rotations. □

Lemma 3 *A node with removed* = false *has at least one parent with removed* = false *that points to it (except the root, as it has no parent).*

Proof. Assume by contradiction that there is no parent with *removed* = false. When a node is first added to the tree it has a parent with *removed* = false (line 44 of the find operation). The only operations that removes links from nodes are the remove and rotate operations (line 26 and 80), but in each case when a link is removed, a new link from a different node with *removed* = false is added (lines 21–23 and 82–84). □

Lemma 4 *A node with removed* = false *has a path from the root node to it.*

Proof. Given that the root is always the same node and it always has *removed* = false the proof of this lemma follows directly from lemmas 2 and 3. □

Lemma 5 *A node with removed* = true *has no path from the root node to it.*

Proof. By the way the tree is structured the root node always has *removed* = false. Now it follows directly from lemma 2 that there is no path from the root to a node with *removed* = true. □

Lemma 6 *The nodes with removed* = false *make up a single binary tree.*

Proof. From the structure of a node, it can have at most 2 children. Now by lemmas 1, 4, and 5 the proof follows. □

Lemma 7 *A rotation operation on a valid binary search tree results in a valid binary tree of the nodes with removed* = false.

Proof. From Figure 2 which describes the *rotation* operation and due to the use of transactional reads/writes we can see that the resulting tree is equivalent to a tree with a classical binary tree rotation performed on it. □

For the following Lemma, we assume that the find operation is performed correctly (this will be proved in a later lemma).

Lemma 8 *Assuming a correct* find *operation, a successful* insert *operation on a valid binary search tree results in a valid binary tree of the nodes with removed* = false.

Proof. There are two cases to consider.

First the key k that we are inserting is already contained in the tree with *removed* = false (lines 38–39 of find). In this case the structure of the tree will not be modified, thus the resulting tree will still be valid.

Second consider that there is no node in the tree with key k. In this case the find operation will return the correct node that will then become the parent of the new node with key k (line 46–47). Using transactional reads, the find operation ensures that the parent node has *removed* = false (line 44) and $\perp$ (line 48) for the child pointer where the new node will be inserted. The new node is then created and added to the tree as the child of the node returned from find. This is done using a transactional write (lines 46–47) which ensures that the value of the pointer was $\perp$ before the write, and finally resulting in a valid tree containing the new node after the transaction commits. □

Lemma 9 *A successful* remove *operation on a valid binary search tree results in a valid binary tree that does not contain the node removed.*

Proof. A removal can only be preformed on a node n with at least one $\perp$ child which is ensured by a transactional read (lines 17–19). Node n being removed is unlinked from its parent (lines 21-23) (the parent is ensured to be in the tree by a transaction read on *removed* = false) effectively removing it from the tree (lemma 1). If both children of n are set to $\perp$, then the parent's new child becomes $\perp$ (lines 17–23) leaving the tree still valid. If node n has a child c such that $c \neq \perp$, then c becomes the parent's new child (lines 17–23). By lemma 2 this c must have *removed* = false and by lemma 4 it is part of the valid binary tree during the transaction (until the transaction commits). Thus the resulting tree is still valid. □

Lemma 10 *Modifications to the tree structure are only performed on nodes with removed* = false.

Proof. A rotate, insert or remove operation can modify the tree.

During a right rotate operation the nodes that are modified are n, l and the parent of n (lines 79–84). A transactional read is performed on the remove variable of the parent node (line 63), this along with lemma 2 ensures that *removed* = false for the parent of n, n, and l. By symmetry the left rotate operation also only modifies nodes with *removed* = false.

During a successful insert operation a new node might be added to the tree, in this case its parent node is modified (lines 46–47), and a transactional read is performed on the parent to ensure that *removed* = false (line 44 of the find operation).

During a remove operation a node is removed from the tree. A transactional read is performed on the node and its parent (line 8), this along with lemma 2 ensures that *removed* = false for both nodes. □

Lemma 11 *From a node with removed* = true *there is always a path to some node with removed* = false.

Proof. There are two places where a node can be set to *removed* = true. First during the remove operation, in this case the node that is removed has both of the nodes child pointers are set to a node with *removed* = false (lines 24–25). Second during the rotate operation, in this case the node that is removed does not have its child pointers changed. By line 72 n must have at least 1 child and using lemma 2 this child must have *removed* = false.

Now notice that once the *removed* field of a node is set to true it will never be reverted to false (lemma 10). This along with the use of induction on the length of the path to a node with *removed* = false completes the proof. □

Lemma 12 *From any node every path leads to a leaf node (or $\perp$).*

Proof. This proof is the same as lemma 11 with a small modification.

First consider a node with *removed* = false. By lemma 6 it is clear that there is a path from this node to a leaf node.

Now consider a node with *removed* = true. There are two places where a node can be set to *removed* = true. First in removal, but in this case the node's left and right pointers are set to a node with *removed* = false (lines 24–25). Second in rotation, here the node's left and right pointers are not changed. Before the rotation the node has *removed* = false therefore by lemma 2 the (node's left and right) pointers will point to either nodes with *removed* = false or $\perp$.

Now notice that after a node has had *removed* set to false the node will not be modified again (lemma 10), this along with the use of induction on the length of the path to a node with *removed* = false or $\perp$ completes the proof. □

The phrase *a node that has removed* = true *from a rotation operation* refers to the node that is no longer in the tree after the rotation. For example in figure 2(c) this would be the node j. This phrase is used in the following lemmas.

Lemma 13 *A node n with key k that has removed* = true *from a right_rotation operation for its left child has a path to a range of keys at least as large as n did before the rotation (including the new node $n2$ with key k), and for its right child a path to a range of keys at least as large as the right child before the rotation, or $\perp$.*

Proof. Assume that the node n has key k and before the rotation has a path to a range $[a, b]$. This means that node l (the left child of n) has a path to the range $[a, k]$. Assume node l has key j. The right child of n is either *bot* or some node r with a path to the range $[k, b]$.

After the rotation l keeps it left child with its right child changing to $n2$. $n2$'s right child becomes n's right child, and $n2$'s left child becomes l's old right child. This leaves $n2$ with a path to the range $[j, b]$, and l with a path to the range $[a, b]$.

During the *rotation* operation no modifications are made to node n, except setting *removed* = true. Now n's left child is l giving it a path to the range of keys $[a, b]$. Node n still has the same right child who still has the same range, $[k, b]$. □

Lemma 14 *A node n with key k that has removed* = true *from a $left\_rotation$ operation for its right child has a path to a range of keys at least as large as n did before the rotation (including the new node $n2$ with key k), and for its left child a path to a range of keys at least as large as the left child before the rotation, or $\perp$.*

Proof. This proof follows from symmetry and lemma 13. □

Lemma 15 *A node that has removed* = true *from a* remove *operation has a path to a range of keys as least as large as just before the* remove *operation took place.*

Proof. Assume that the node n (where n is the node to be removed) has key k. Assume that n has a parent node p with range $[a, b]$ with key j. This leaves n with range $[a, k]$ if n is the left child (or $[k, b]$ if n is the right child, the proof of this case will follow by symmetry).

After the removal *removed* will be set to true (line 26) for n and both its child pointers will point to p (lines 24-25). Node p will still have a range of $[a, b]$ and will be reachable from n, which completes the proof. □

Lemma 16 *A node that has removed* = true *has either:*

- *for each child a path to a range of keys at least as large as they did when it had removed* = false *or*
- *a single $\perp$ child with the other child having a path to a range of keys at least as large as both children before when it had removed* = false.

Proof. The proof follows using lemmas 13, 14, and 15 as well as induction on the length of the path from the node to a node with *removed* = false. □

Lemma 17 *A* find *operation on a valid binary search tree always returns the correct location from the valid binary tree.*

Proof. Assume by contradiction that a find operation does not return the correct location from the valid binary tree. By lemma 6 we know that every node with *removed* = false is a node in the valid binary tree so there are two possibilities. The operation never returns or the operation returns the wrong location. First for the operation never returning. From lemma 12 we know that there are no cycles in the path from any node so the operation will not get stuck in an endless loop. In order for the operation to complete it must reach a node with *removed* = false that either matches the key being searched for

(lines 38–39) or has $\perp$ as the appropriate child (lines 44–47). Lemma 11 is enough to show that this will always happen and the operation will terminate.

The second possibility where the operation returns the wrong location is more difficult to prove. To this end, we prove by induction on the number of nodes traversed by the operation that the find operation will always have a path to the correct location. In order for the operation to reach the correct location the following must be true: After each traversal from one node to the next the operation must either be at the correct location or have a path from the current location to a range of keys that includes the key being searched for. Once the correct location is reached transactional reads are used to ensure that the location remains correct throughout the transaction (lines 39, and 44–47).

- The base case is easy given that the operation starts from the root which is always part of the valid tree and can reach all nodes with *removed* = false (lemma 4).
- Now the induction step. Assume that after $n-1$ nodes have been traversed the operation has a path to a range of keys that includes the key k that is being searched for. If the $n-1^{th}$ node is the correct location the the proof is done, otherwise the operation must travel to the n^{th} node. Now it must be shown that the after the operation travels to the n^{th} node it is still on the correct path. There are 2 possibilities.
  1. The operation can move from a node with *removed* = false (at the time of the load of the child pointer, lines 41–42). By lemma 2 the child must also have *removed* = false (at the time of the load of the child pointer), in this case the traversal is performed on a valid binary tree (lemma 6). Since the choice of the the child is based on a standard tree traversal the operation must still be on the correct path.
  2. The operation can move from a node with *removed* = true. By the assumption given by induction the $n-1^{th}$ node has a path to the range of keys that includes k. From the $n-1^{th}$ node either the right or left child must be chosen. The node is chosen based on a standard tree traversal. If the node is not $\perp$ then by lemma 16 it must have a path to the same range as when it has *removed* = false, which includes k (the $n-1^{th}$ node's range includes k so the n^{th} node's range must also). Otherwise if one child is $\perp$ then the other child is chosen (lines 51–52), which must have a path to a range at least as large as the $n-1^{th}$ node by lemma 16 (which includes k).

□

Theorem 2 *An* insert *operation is valid.*

Proof. The insert operation starts by performing a find operation (line 38). From lemma 17 we know that the find operation will return the correct place. There are two possibilities, either the find operation returns a node in the tree with key k (line 38), or it returns a node in the tree that has $\perp$ for the child where k must be inserted (line 48).

First consider the case where a node with key k is returned. The transactional read on *removed* (line 39 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. Next a transactional read is done on the node's *deleted* variable (line 40) ensuring that this value will remain the same throughout the transaction. If the read on *deleted* returns false then the insert operation can return false because the node exists in the set. Otherwise if the read on *deleted* returns true then a transaction performs a transactional write setting *deleted* to false (line 41). The transactional read on *deleted* ensures that k is not in the set before the transaction commits. The transactional write on *deleted* ensures that k is in the set after the transaction commits.

Second consider the case where a node with $key \neq k$ is returned. On lines 46–47 of the find operation a transactional read has been done on the child pointer of this node, returning $\perp$, which must be valid throughout the transaction. The transactional read on *removed* (line 44 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. By lemma 6 this node belongs to a correct binary tree, this along with lemma 17 ensures that the child of this node is the only place where a node with key k could exist (and since the value of the child pointer of $\perp$ a node with key k is not in the tree). A new node with key k is created and then added to the tree using a transactional write to the child pointer (lines 46–47). The *removed* and *deleted* fields of a newly created node can only be false so when the transaction commits the new node will be in the tree and k will be in the set. □

Theorem 3 *A* contains *operation is valid.*

Proof. The contains operation starts by performing a find operation (line 28). From lemma 17 we know that the find operation will return the correct place. There are two possibilities, either the find operation returns a node in the tree with key k (line 38), or it returns a node in the tree that has $\perp$ for the child where k could exist (line 44).

First consider the case where a node with key k is returned. The transactional read on *removed* (line 39 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. Next a transactional read is done on the node's *deleted* variable (line 32) ensuring that this value will remain the same throughout the transaction. If

the read on *deleted* returns false then the *contains* operation can return true because the node exists in the set otherwise false can be returned because the node does not exist in the set.

Second consider the case where a node with $key \neq k$ is returned. On lines 46–47 of the find operation a transactional read has been done on the child pointer of this node, returning $\perp$, and due to the transactional read the pointer must remain as $\perp$ throughout the transaction. The transactional read on *removed* (line 44 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. By lemma 6 this node belongs to a correct binary tree, this along with lemma 17 ensures that the child of this node is the only place where a node with key k could exist (and since the value of the child pointer is $\perp$ a node with key k is not in the tree). The contains operation can then return false. □

Theorem 4 *A* delete *operation is valid.*

Proof. The delete operation is almost the same as the contains operation. The only difference is in the case where a node with key k is returned from the find operation. The transactional read on *removed* (line 39 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. Next a transactional read is done on the node's *deleted* variable (line 78) ensuring that this value will remain the same throughout the transaction. If the read on *deleted* returns true then the delete operation can return false because the node does not exist in the set. Otherwise k is in the set and a transactional write is performed setting the nodes *deleted* variable to true. Since *removed* is false this node is part of the valid tree so it is the only place where key k can be so by setting *deleted* to true k must not be in the set. □