A General Technique for Non-blocking Trees

Trevor Brown, University of Toronto, Canada
Faith Ellen, University of Toronto, Canada
Eric Ruppert, York University, Canada

PPoPP 2014

Problem

Balanced binary search trees (BSTs) are important, but there has been little success implementing them without locks.

- *Coarse-grained transactions.* Limited concurrency, high abort rates, code needs fallback paths.
- *Single-word CAS.* Extremely complex algorithms and proofs or lack of rigorous proofs.
- *Multi-word compare-and-swap (CAS).* Inefficiency.

Previous Frameworks for Tree Updates

- Tsay and Li (1994): wait-free trees using LL/SC.
  - Every update or search must copy an entire path from root to leaf.
- Natarajan et al. (2013): extends Tsay and Li's framework.
  - Searches no longer copy nodes.
  - Updates can avoid copying some nodes in special cases.
  - Updates must "lock" each node on a root to leaf path with CAS (similar to lock-coupling).

Goals of This Work

Goal 1

A template for efficient non-blocking implementation of down-trees that are:

- Linearizable
- Non-blocking
- Relatively simple to prove correct, and
- Allow disjoint updates to succeed concurrently.

Goal 2

A **practical**, **provably correct**, non-blocking balanced BST.

LLX and SCX

The LLX and SCX primitives:

- can be implemented from CAS, and
- work on data records, which contain some mutable fields and some immutable fields

LLX(r) returns a snapshot of the mutable fields of r

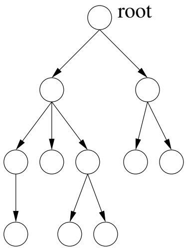
SCX(V, R, field, new) by process *p*

- writes value new into field,
  which is a mutable field of a data record in V
- finalizes all data records in R
- only if no record in V has changed since *p*'s LLX on it

After a data record is finalized, no further changes allowed.

Template

We give a template using LLX/SCX to make local changes to a down-tree.

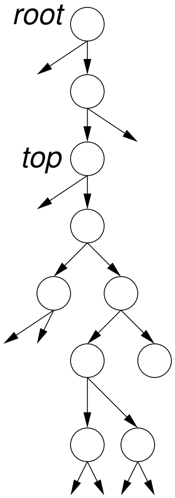


Any data structure based on a down-tree that follows our template for all its updates is automatically linearizable and non-blocking.

Representing a Tree Using Data Records

- Represent each node as a data record.
- Child pointers are mutable fields.
- Other data stored in a node is immutable.
  (To change any of this data, a new copy of the node is made.)

Tree Update Template

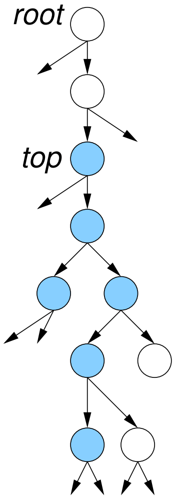


1. read child pointers, from *root* to some node *top*
2. LLX *top* and a contiguous set of its descendants
3. Select subgraph *R* to replace and create replacement subgraph *N*
4. Use SCX to change child pointer of *par*:
   - replaces *R* by *N*
   - and finalizes *R*
   - only if LLXed nodes unchanged

Requirements

- Children of *R* = Children of *N*
- Must LLX *par* and all nodes in *R*

Tree Update Template

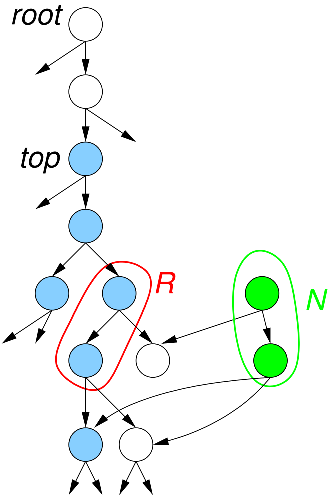


1. read child pointers, from *root* to some node *top*
2. LLX *top* and a contiguous set of its descendants
3. Select subgraph *R* to replace and create replacement subgraph *N*
4. Use SCX to change child pointer of *par*:
   - replaces *R* by *N*
   - and finalizes *R*
   - only if LLXed nodes unchanged

Requirements

- Children of *R* = Children of *N*
- Must LLX *par* and all nodes in *R*

Tree Update Template

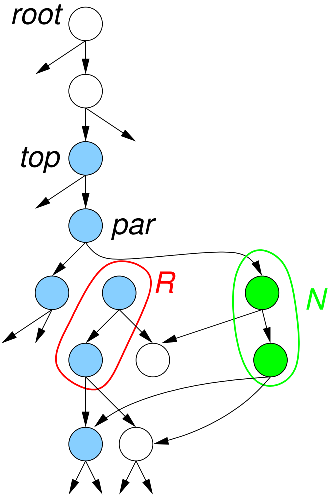


1. read child pointers, from *root* to some node *top*
2. LLX *top* and a contiguous set of its descendants
3. Select subgraph *R* to replace and create replacement subgraph *N*
4. Use SCX to change child pointer of *par*:
   - replaces *R* by *N*
   - and finalizes *R*
   - only if LLXed nodes unchanged

Requirements

- Children of *R* = Children of *N*
- Must LLX *par* and all nodes in *R*

Tree Update Template

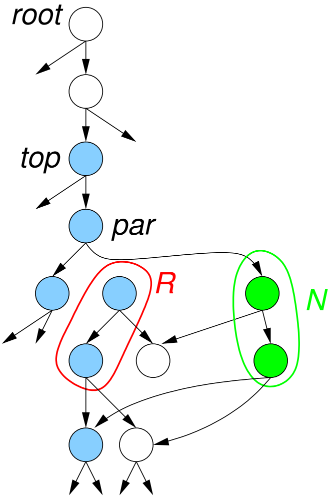


1. read child pointers, from *root* to some node *top*
2. LLX *top* and a contiguous set of its descendants
3. Select subgraph *R* to replace and create replacement subgraph *N*
4. Use SCX to change child pointer of *par*:
   - replaces *R* by *N*
   - and finalizes *R*
   - only if LLXed nodes unchanged

Requirements

- Children of *R* = Children of *N*
- Must LLX *par* and all nodes in *R*

Tree Update Template



1. read child pointers, from *root* to some node *top*
2. LLX *top* and a contiguous set of its descendants
3. Select subgraph *R* to replace and create replacement subgraph *N*
4. Use SCX to change child pointer of *par*:
   - replaces *R* by *N*
   - and finalizes *R*
   - only if LLXed nodes unchanged

Requirements

- Children of *R* = Children of *N*
- Must LLX *par* and all nodes in *R*

Linearizing the Tree Updates

Linearization point of each update: its successful SCX.

Invariant

Tree is the same as it would be if tree updates were done in order by linearization points.

Fast Queries

Some queries can be done quickly by reads only.

Example: SEARCH(k) **in a BST**

- Just read sequence of pointers from root to leaf.
- Ignore concurrent updates along the path

Leaf reached was on the search path to k at some time during the SEARCH.
This is sufficient to linearize the SEARCH

Non-blocking Balanced Trees

- Braginsky and Petrank, SPAA 2012: B+tree.
- Natarajan, Savoie and Mittal, SSS 2013: red-black tree using wait-free tree framework.

Red-Black Trees

- Each node is red or black.
- Root is black.
- Leaves are black.
- Red nodes have black parents.
- Every root to leaf path contains the same number of black nodes.

Search, Insert, Delete take $O(\log n)$ steps.

amortized $O(1)$ rebalancing steps per update

Red-Black Trees

- Each node is red or black.
- Root is black.
- Leaves are black.
- Red nodes have black parents.
- Every root to leaf path contains the same number of black nodes.

Search, Insert, Delete take $O(\log n)$ steps.

amortized $O(1)$ rebalancing steps per update

Red-Black Trees

- Each node is red or black.
- Root is black.
- Leaves are black.
- Red nodes have black parents.
- Every root to leaf path contains the same number of black nodes.

Search, Insert, Delete take $O(\log n)$ steps.

amortized $O(1)$ rebalancing steps per update

Concurrent Red-Black Trees

Each update and its necessary rebalancing steps must be performed atomically.

- Limits concurrency.

Solution: decouple reblancing from updating, so they can be interleaved.

Concurrent Red-Black Trees

Each update and its necessary rebalancing steps must be performed atomically.

- Limits concurrency.

Solution: decouple reblancing from updating, so they can be interleaved.

Chromatic Tree

Relaxed version of red-black tree designed for use with locks.

- allow red node to have a red parent (red-red violation).
- allow a black node to count more than others (overweight violation).
- a chromatic tree with no violations is a red-black tree.
- rebalancing steps can be deferred and interleaved with inserts and deletes.
- amortized O(1) rebalancing steps per insert or delete.
- 22 different rebalancing steps.

Chromatic Tree

Relaxed version of red-black tree designed for use with locks.

- allow red node to have a red parent (red-red violation).
- allow a black node to count more than others (overweight violation).
- a chromatic tree with no violations is a red-black tree.
- rebalancing steps can be deferred and interleaved with inserts and deletes.
- amortized O(1) rebalancing steps per insert or delete.
- 22 different rebalancing steps.

Chromatic Tree

Relaxed version of red-black tree designed for use with locks.

- allow red node to have a red parent (red-red violation).
- allow a black node to count more than others (overweight violation).
- a chromatic tree with no violations is a red-black tree.
- rebalancing steps can be deferred and interleaved with inserts and deletes.
- amortized O(1) rebalancing steps per insert or delete.
- 22 different rebalancing steps.

Chromatic Tree

Relaxed version of red-black tree designed for use with locks.

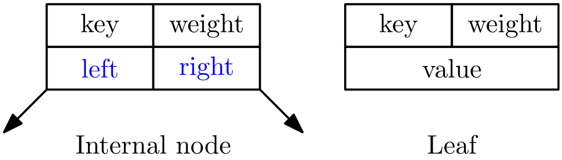
- allow red node to have a red parent (red-red violation).
- allow a black node to count more than others (overweight violation).
- a chromatic tree with no violations is a red-black tree.
- rebalancing steps can be deferred and interleaved with inserts and deletes.
- amortized O(1) rebalancing steps per insert or delete.
- 22 different rebalancing steps.

Representing a Chromatic Tree

Leaf oriented: every value is stored in a leaf.
Internal nodes store keys that are only used to direct searches towards a leaf.

Each node is represented by a data record.

- immutable fields: key, weight, value
- mutable fields: left, right



Search

Same as in a sequential binary search tree
Can completely ignore concurrent updates

Insert(k,v)

repeat

  Search for leaf u_x where k should be inserted

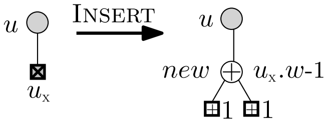
  If $u_x.key = k$ then return

  try to apply INSERT using tree update template

  if successful then

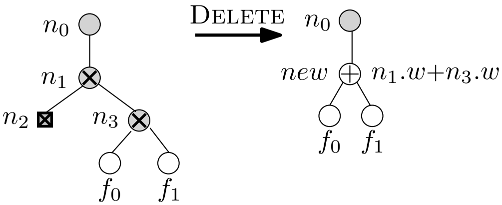
    if a violation was created then Cleanup(k)

    return



Delete(k)

repeat
  Search for leaf n_2 where k should be located
  If $n_2.key \neq k$ then return
  try to apply DELETE using tree update template
  if successful then
    if a violation was created then Cleanup(k)
    return



Cleanup(k)

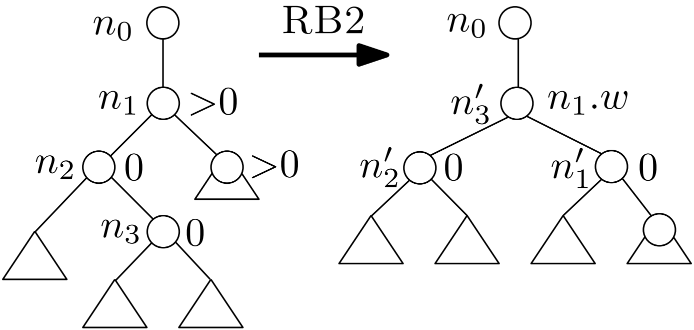
repeat

    Search for leaf with key k until a violation is found
    If no violation found then return
    Choose which rebalancing step to apply
    Try to apply the rebalancing step using tree update template

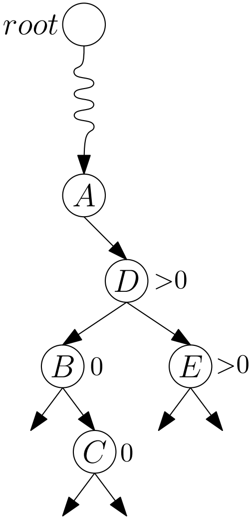
INVARIANT: If a violation is on the search path for k before a rebalancing step, then it is eliminated or it remains on this path.

When contention is c, chromatic tree has height $O(c + \log n)$.

Applying a Rebalancing Step



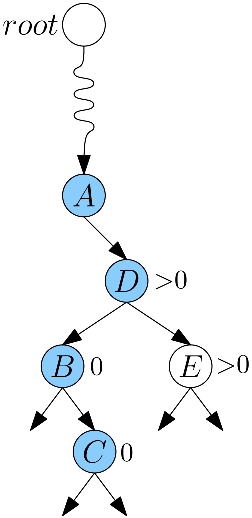
Applying a Rebalancing Step



Apply RB2 to fix violation at *C*:

1. Read path from root to C
2. LLX *A*, *D*, *B*, *C*
3. Create new nodes B', C', D' to replace $R = \langle D, B, C \rangle$
4. SCX($\langle A, D, B, C \rangle$, $\langle D, B, C \rangle$, $A.right$, B')
   - changes A.right to B' and
   - finalizes *D*, *B*, *C*
   - only if *A*, *D*, *B*, *C* unchanged

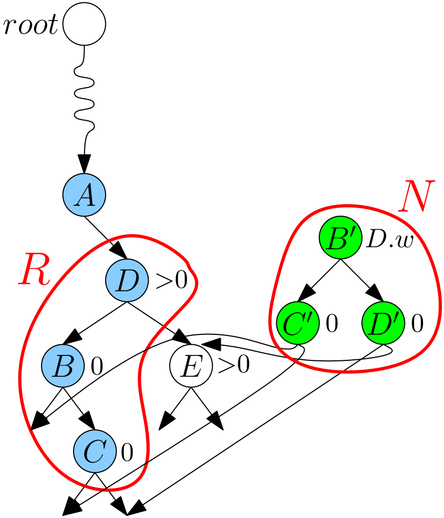
Applying a Rebalancing Step



Apply RB2 to fix violation at C:

1. Read path from root to C
2. LLX A, D, B, C
3. Create new nodes B', C', D' to replace $R = \langle D, B, C \rangle$
4. SCX($\langle A, D, B, C \rangle$, $\langle D, B, C \rangle$, $A.right$, B')
   - changes A.right to B' and
   - finalizes D, B, C
   - only if A, D, B, C unchanged

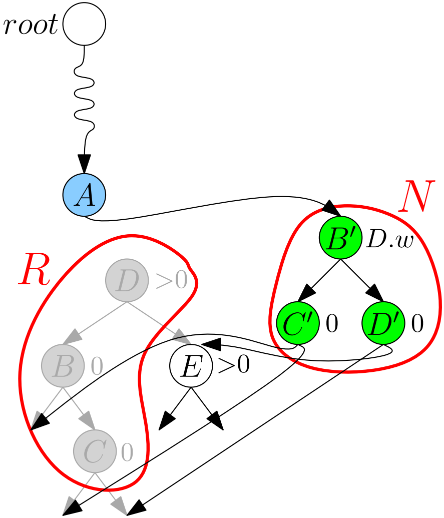
Applying a Rebalancing Step



Apply RB2 to fix violation at C:

1. Read path from root to C
2. LLX A, D, B, C
3. Create new nodes B', C', D' to replace $R = \langle D, B, C \rangle$
4. SCX($\langle A, D, B, C \rangle$, $\langle D, B, C \rangle$, $A.right$, B')
   - changes A.right to B' and
   - finalizes D, B, C
   - only if A, D, B, C unchanged

Applying a Rebalancing Step



Apply RB2 to fix violation at *C*:

1. Read path from root to C
2. LLX *A*, *D*, *B*, *C*
3. Create new nodes *B′*, *C′*, *D′* to replace *R* = ⟨*D*, *B*, *C*⟩
4. SCX(⟨*A*, *D*, *B*, *C*⟩, ⟨*D*, *B*, *C*⟩, *A.right*, *B′*)
   - changes A.right to *B′* and
   - finalizes *D*, *B*, *C*
   - only if *A*, *D*, *B*, *C* unchanged

Using the Tree Update Template

Makes application of rebalancing steps easy:

- atomically replace the left side with the right side
- specific details of rebalancing steps are unimportant

Makes proofs of correctness and proofs of progress much easier:

data structure	correctness	progress
unbalanced binary search tree	19 pages	4 pages
B+ Tree	27 pages	6 pages
chromatic tree	4 pages	1 page

Improving Practical Performance

Count violations as you search for the leaf to update
After updating, invoke Cleanup if at least b violations seen

Tree has height $O(c + b + \log n)$

Other Non-blocking Balanced Trees

Relaxed balance data structures:

- decouple rebalancing from other updates to the data structure
- allow updates to be interleaved arbitrarily
- many relaxed balance versions of sequential data structures exist in the literature
- are well suited for non-blocking implementations using the tree update template

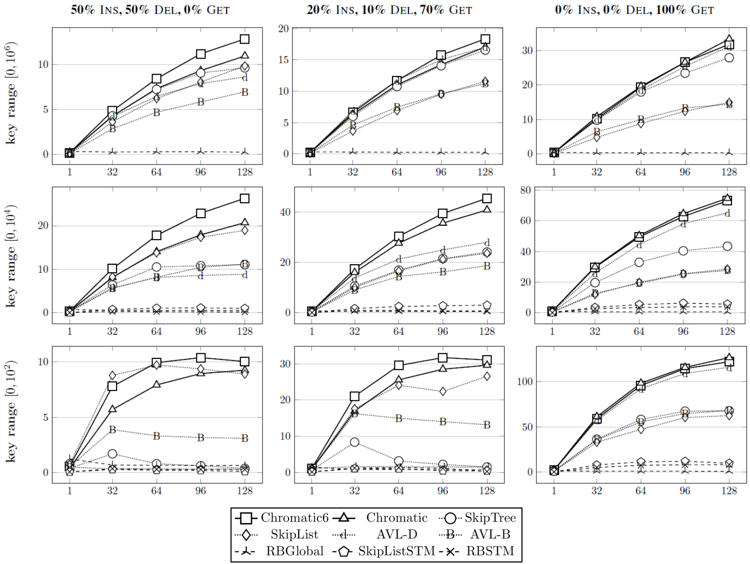
Relaxed AVL tree: non-blocking implementation took a first-year undergraduate student one week

Experiments

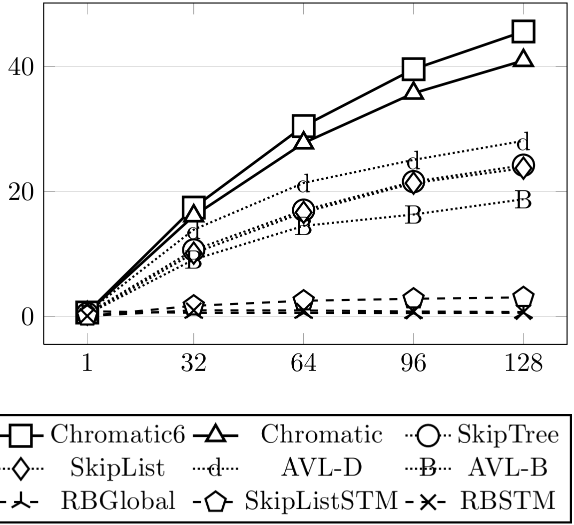
Measured the throughput (number of operations/second) for

- Chromatic tree
- Chromatic tree allowing 5 violations (Chromatic 6)
- Non-blocking multiway search tree (SkipTree)
- Non-blocking skip list (SkipList)
- lock-based AVL tree (AVL-B)
- lock-based AVL tree with non-blocking search (AVL-D)
- STM-based skiplist (SkipListSTM)
- STM-based red-black tree (RBSTM)
- lock-based red-black tree (RBGlobal)

Experiments

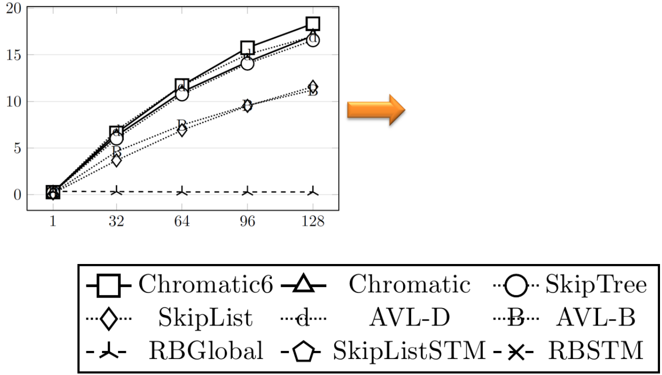


20% Ins, 10% Del, 70% Get, Key Range $[0, 10^4)$

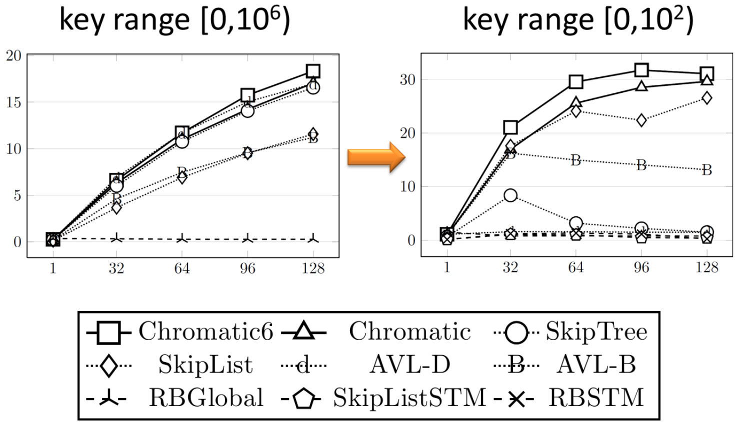


How Performance Changes with Contention

key range $[0,10^6)$



How Performance Changes with Contention



Summary

- Template for building non-blocking trees
  vastly simplifies proof of correctness, proof of progress.
- Very efficient, provably correct implementation of non-blocking chromatic tree.
- Searches are invisible, and extremely fast.
- Cleanup algorithm allows tree invariants to be relaxed for better concurrency without losing the height bound. (Idea can be applied to many relaxed data structures.)

Future work:

- Investigating HTM implementation of LLX/SCX