Verification-Friendly Concurrent Balanced Binary Search Tree

Dana Drachsler, Technion, Israel

Joint work with:
Martin Vechev, ETH, Switzerland
Eran Yahav, Technion, Israel
Motivation

• Balanced *Binary Search Tree* (BST) is an efficient data-structure for storing unique elements
  ▫ No repetitions are allowed

• Formal verification:
  ▫ Given a program, prove some property
  ▫ In the tree:
    • prove that repetitions of elements cannot occur
Motivation

- Formal verification was applied to the sequential algorithm (e.g. using Isabelle [6])
- However, in a concurrent setting, formal verification is more complicated
Motivation

• There seems to be a trade-off between algorithms that are easy to verify and algorithms that are practical
• A concurrent BST that is protected by a global lock is easy to verify
• Practical concurrent trees use sophisticated mechanisms
  ▫ Many different cases to reason about
  ▫ Harder to verify
Goal

- We gap this trade-off by presenting a concurrent BST that is both practical and simple to reason about
- Our key idea:
  - Integrate the property into the algorithm
- We achieve a fine-grained locking balanced BST
- Our tree is very similar to the sequential tree
- Our mechanism allows breaking the proof into several separated proofs
Outline

Binary Search Tree → Balanced Binary Search Tree

Concurrent Binary Search Tree → Concurrent Balanced Binary Search Tree
Binary Search Tree

- A data-structure that stores elements
- Consists of nodes
- Each node represents an element
  - Internal tree
- Each element has a unique key
  - Repetitions are not allowed
- Each node in the tree holds:
  - The left sub-tree has elements with *smaller* keys
  - The right sub-tree has elements with *bigger* keys
Binary Search Tree

• In other words, BST maintains two types of invariants:
  ▫ Set invariant
    • Each key appears at most once
  ▫ BST invariants
    • For each node:
      • The keys in the left sub-tree are smaller
      • The keys in the right sub-tree are bigger
Binary Search Tree

- Supports the following operations:
  - Contains

Contains 24?
Binary Search Tree

- Supports the following operations:
  - Contains

```
   6
  / \
 3   12
   /   \
 24?
```
Binary Search Tree

- Supports the following operations:
  - Insert
    - The new node is always a leaf
Binary Search Tree

• Supports the following operations:
  ▫ Insert
    • The new node is always a leaf
Binary Search Tree

- Supports the following operations:
  - Remove
    - The removed node, \( n \), may be:
      - A leaf
Binary Search Tree

• Supports the following operations:
  ▫ **Remove**
    • The removed node, \( n \), may be:
      • A leaf
Binary Search Tree

- Supports the following operations:
  - **Remove**
    - The removed node, $n$, may be:
      - A leaf
      - A parent of a single child
        - $n$’s parent is connected to $n$’s child
Binary Search Tree

- Supports the following operations:
  - **Remove**
    - The removed node, $n$, may be:
      - A leaf
      - A parent of a single child
        - $n$’s parent is connected to $n$’s child
Binary Search Tree

- Supports the following operations:
  - **Remove**
    - The removed node, \( n \), may be:
      - A leaf
      - A parent of a single child
        - \( n \)’s parent is connected to \( n \)’s child
      - A parent of two children
        - \( n \)’s successor is relocated to \( n \)’s location
Binary Search Tree

- Supports the following operations:
  - **Remove**
    - The removed node, \( n \), may be:
      - A leaf
      - A parent of a single child
        - \( n \)’s parent is connected to \( n \)’s child
      - A parent of two children
        - \( n \)’s successor is relocated to \( n \)’s location
Outline

Binary Search Tree -> Balanced Binary Search Tree

Concurrent Binary Search Tree -> Concurrent Balanced Binary Search Tree
Challenges in Concurrent BST

• Consider the following tree:
  ▫ Thread A searches for 9
Challenges in Concurrent BST

• Consider the following tree:
  ▫ Thread A searches for 9 and pauses
Challenges in Concurrent BST

• Consider the following tree:
  ▫ Thread A searches for 9 and pauses
  ▫ Thread B removes 6
Challenges in Concurrent BST

• Consider the following tree:
  ▫ Thread A searches for 9 and pauses
  ▫ Thread B removes 6
Challenges in Concurrent BST

• Consider the following tree:
  ▫ Thread A searches for 9 and pauses
  ▫ Thread B removes 6
  ▫ Thread A resumes the search
Challenges in Concurrent BST

• Consider the following tree:
  ▫ Thread A searches for 9 and pauses
  ▫ Thread B removes 6
  ▫ Thread A resumes the search and observes that 9 is not present
How do others cope with this challenge?

• By not supporting the remove operation
  ▫ Bender et al. [1]
How do others cope with this challenge?

• By using external trees
  ▫ Only leaves can be removed
  ▫ Use more space than internal trees
  ▫ Ellen et al. [4]
How do others cope with this challenge?

- Many concurrent algorithms for data-structures remove elements in two steps:
  - Marking the node as *logically removed*
How do others cope with this challenge?

- Many concurrent algorithms for data-structures remove elements in two steps:
  - Marking the node as *logically* removed
  - Update pointers to *physically* remove the node
How do others cope with this challenge?

- By marking the node as removed without physically removing it
  - Also known as partially-external trees
  - Bronson et al. [2]
  - Crain et al. [3]
How do others cope with this challenge?

- By marking the node as removed without *physically* removing it
  - Also known as partially-external trees
  - Bronson et al. [2]
  - Crain et al. [3]
How do others cope with this challenge?

- By marking the node as removed without *physically* removing it
  - Also known as partially-external trees
  - Bronson et al. [2]
  - Crain et al. [3]
How do others cope with this challenge?

- By marking the node as removed without physically removing it
  - Also known as partially-external trees
  - Bronson et al. [2]
  - Crain et al. [3]
How do others cope with this challenge?

• By marking the node as removed without *physically* removing it
  ▪ Howley et al. [5]

A: contains(9)
How do others cope with this challenge?

• By marking the node as removed without *physically* removing it
  ▫ Howley et al. [5]

A: contains(9)
How do others cope with this challenge?

• By marking the node as removed without physically removing it
  ▫ Howley et al. [5]
How do others cope with this challenge?

- By marking the node as removed without *physically* removing it
  - Howley et al. [5]
How do others cope with this challenge?

- These solutions leave removed nodes in the tree
- Is it possible to *physically* remove nodes?
- Trivial solution: use global lock
How do others cope with this challenge?

• These solutions leave removed nodes in the tree
• Is it possible to *physically* remove nodes?
• Trivial solution: use global lock
• **Observation**: To determine whether \( k \) is in the tree it is enough to have \( p, s \) such that:
  ▫ \( p, s \) belong to the tree
  ▫ Any \( w \in (p, s) \) is not in the tree
Our Approach

- Maintain the predecessor-successor relation
  - The set layout
- Consult this relation before making final decisions
Our Approach

- Maintain the predecessor-successor relation
  - The set layout
- Consult this relation before making final decisions
Our Approach

- Maintain the predecessor-successor relation
  - The set layout
- Consult this relation before making final decisions

A: contains(9)
Our Approach

• Maintain the predecessor-successor relation
  ▫ The set layout
• Consult this relation before making final decisions
Our Approach

- Maintain the predecessor-successor relation
  - The set layout
- Consult this relation before making final decisions
Our Approach

- Maintain the predecessor-successor relation
  - The set layout
- Consult this relation before making final decisions

A: contains(9)
B: remove(6)
Our Approach

- Maintain the predecessor-successor relation
  - The set layout
- Consult this relation before making final decisions
- This relation allows us to lock the required nodes even if they are not adjacent
  - Enjoy the benefits of the global lock
  - While enabling more parallelism

A: contains(9)
B: remove(6)
Contains(k)

- Traverse the tree using the tree pointers
- If $k$ was found
  - Return true
- Otherwise, upon reaching to a leaf $l$, confirm:
  - $k \in (l$'s predecessor, $l$) or $k \in (l, l$'s successor)
  - and return false

- This operation does not acquire locks
Update Operations

• The synchronization is based on locks
• Each update operation locks:
  ▫ The relevant nodes in the tree
  ▫ The relevant intervals
Insert(k)

- Traverse the tree to find the location
Insert(k)

- Traverse the tree to find the location
- Let $l$ be the node found
Insert\( (k) \)

- Traverse the tree to find the location
- Let \( l \) be the node found
- If \( k \leq l \): lock \( l \)'s predecessor edge
Insert($k$)

- Traverse the tree to find the location
- Let $l$ be the node found
- If $k \leq l$: lock $l$’s predecessor edge
  - Lock $l$
**Insert**\( (k) \)

- Traverse the tree to find the location
- Let \( l \) be the node found
- If \( k \leq l \): lock \( l \)’s predecessor edge
  - Lock \( l \)
  - Update predecessor-successor
Insert(k)

- Traverse the tree to find the location
- Let $l$ be the node found
- If $k \leq l$: lock $l$’s predecessor edge
  - Lock $l$
  - Update predecessor-successor
  - Add $k$
Insert(k)

- Traverse the tree to find the location
- Let $l$ be the node found
- If $k \leq l$: lock $l$’s predecessor edge
  - Lock $l$
  - Update predecessor-successor
  - Add $k$
- Else: lock $l$’s successor
  - Symmetric.
Remove($k$)

- Traverse the tree to find $k$
- Let $n$ be the node found
- Lock $n$’s predecessor edge
Remove(k)

- Traverse the tree to find $k$
- Let $n$ be the node found
- Lock $n$’s predecessor edge
  - Lock $n$’s successor edge
Remove(k)

- Traverse the tree to find $k$
- Let $n$ be the node found
- Lock $n$’s predecessor edge
  - Lock $n$’s successor edge
  - Lock $n$, $n$’s children and parent
Remove($k$)

- Traverse the tree to find $k$
- Let $n$ be the node found
- Lock $n$’s predecessor edge
  - Lock $n$’s successor edge
  - Lock $n$, $n$’s children and parent
- If $n$ has at most 1 child:
  - Mark $n$ as removed
Remove(k)

- Traverse the tree to find $k$
- Let $n$ be the node found
- Lock $n$’s predecessor edge
  - Lock $n$’s successor edge
  - Lock $n$, $n$’s children and parent
- If $n$ has at most 1 child:
  - Mark $n$ as removed
  - Update predecessor-successor
Remove$(k)$

• Traverse the tree to find $k$
• Let $n$ be the node found
• Lock $n$’s predecessor edge
  ▫ Lock $n$’s successor edge
  ▫ Lock $n$, $n$’s children and parent
• If $n$ has at most 1 child:
  • Mark $n$ as removed
  • Update predecessor-successor
  • Connect $n$’s parent and child
Remove(k)

- Traverse the tree to find k
- Let n be the node found
- Lock n’s predecessor edge
  - Lock n’s successor edge
  - Lock n, n’s children and parent
- If n has at most 1 child:
  - Mark n as removed
  - Update predecessor-successor
  - Connect n’s parent and child
Remove(k)

- If $n$ has 2 children:
Remove(k)

- If $n$ has 2 children:
  - Lock $n$'s successor, its parent and child
Remove(k)

- If \( n \) has 2 children:
  - Lock \( n \)'s successor, its parent and child
  - Release \( n \)'s children locks
Remove(k)

- If \( n \) has 2 children:
  - Lock \( n \)'s successor, its parent and child
  - Release \( n \)'s children locks
  - Mark \( n \) as removed
Remove(k)

- If \( n \) has 2 children:
  - Lock \( n \)'s successor, its parent and child
  - Release \( n \)'s children locks
  - Mark \( n \) as removed
  - Update predecessor-successor
**Remove(k)**

- **If n has 2 children:**
  - Lock n’s successor, its parent and child
  - Release n’s children locks
  - Mark n as removed
  - Update predecessor-successor
  - Connect the successor’s parent to the successor’s child and relocate n’s successor
Remove($k$)

- **If $n$ has 2 children:**
  - Lock $n$’s successor, its parent and child
  - Release $n$’s children locks
  - Mark $n$ as removed
  - Update predecessor-successor
  - Connect the successor’s parent to the successor’s child and relocate $n$’s successor
Update Operations Scheme

- Traverse the tree to find $k$
- Lock interval: $[p, s]$
- Confirm that the interval is *appropriate*:
  - $k \in [p, s]$
  - $p$ is not marked as removed
- Lock tree locks
- Update predecessor-successor relation
- Update tree layout
- Release all locks
Correctness

• The BST maintains two invariants
  ▫ **Set invariant**
    • Protected by set-locks
  ▫ **BST invariants**
    • Protected by tree-locks

• The intervals allow us to separate the proof into two proofs
Correctness

• Set invariant
  ▫ Each key appears at most once
• A new key, $k$, is added only after locking an interval $[p, s]$ such that $k \in (p, s)$
• $k$ is not added if $k = p$ or $k = s$
• $k$ cannot be added concurrently by another thread
Correctness

- BST invariants
  - For each node:
    - The keys in the left sub-tree are smaller
    - The keys in the right sub-tree are bigger
- The invariants may only be broken while updating the tree layout
- Any update operation locks all updated nodes
- Locks are released only after the BST invariants are held
Outline

Binary Search Tree → Balanced Binary Search Tree → Concurrent Binary Search Tree → Concurrent Balanced Binary Search Tree
Balanced Binary Search Tree

- In BST, insert, remove and contains run in $O(\log n)$ in average.
- In balanced BST, these operations run in $O(\log n)$ in the worst case.
- There are several known implementations for balanced BSTs
  - We will focus on AVL trees
AVL Trees

• Each node maintains the invariant:
  ▫ The heights of the left and right sub-trees differ by at most 1
AVL Trees

• Each node maintains the invariant:
  ▫ The heights of the left and right sub-trees differ by at most 1
• Insertion and removal may break the invariant
AVL Trees

- Each node maintains the invariant:
  - The heights of the left and right sub-trees differ by at most 1
- Insertion and removal may break the invariant
  - Rotations are applied to fix it
  - Rotations operate on adjacent nodes
AVL Trees

• Each node maintains the invariant:
  ▫ The heights of the left and right sub-trees differ by at most 1
• Insertion and removal may break the invariant
  ▫ Rotations are applied to fix it
  ▫ Rotations operate on adjacent nodes
Balancing Our Tree

• After insertion or removal the tree is traversed bottom-up beginning from the point where an update has occurred
• If violation is detected, rotations are applied
  ▫ Only tree layout locks need to be acquired
Balancing Our Tree

• Rotations may lead to temporary disappearance of nodes from the tree layout
• However, the set-layout is unaffected by these rotations
• Since we consult the set-layout before making final decisions, this cannot lead to wrong decisions
Overview

Binary Search Tree → Balanced Binary Search Tree

Concurrent Binary Search Tree → Concurrent Balanced Binary Search Tree
Evaluation

• We compared our tree to state-of-the-art implementations
• Experiments ran on a machine with 32 cores
Evaluation

- 90% contains, 9% insert, 1% remove

200,000 keys

2,000,000 keys
Summary

• We presented a practical concurrent balanced BST
• Our main insight is that maintaining explicitly the set layout results in a simpler algorithm for the concurrent balanced BST

Thank you!
References


