A Concurrency-Optimal Binary Search Tree

Vitaly Aksenov, INRIA Paris / ITMO University
Vincent Gramoli, University of Sydney
Petr Kuznetsov, Telecom ParisTech
Anna Malova, Washington University in St.Louis
Srivatsan Ravi, University of Southern California

Euro-Par 2017

Motivation

- How to measure efficiency of the data structure?
  - Practically: performance evaluation. Non-portable, architecture- and workload- dependent.
  - Theoretically: lower bounds. Usually worst-case behaviour, rarely observed in practice.
- Concurrency-optimality [Gramoli et al., SIROCCO 2016]. List-based Set [Gramoli et al., DISC 2015].
- This paper: a concurrency-optimal binary search tree exists and performs well.

Outline

Outline

Sequential partially-external BST

Concurrent Binary Search Tree

Concurrency Optimality

Concurrency-optimal BST

Evaluation

Conclusion

Specification

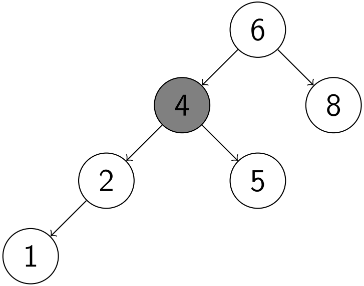
High-level: Set type.

- `Insert`(x)
  - `true` if x does not exists
  - `false` if x exist
- `Delete`(x)
  - `true` if x exists
  - `false` if x does not exist
- `Contains`(x)
  - `true` if x exists
  - `false` if x does not exist

Partially-external BST

Data structure: partially-external binary search tree.

- Key in the node is greater than keys in the left subtree and is less than keys in the right subtree.
- Two types of nodes: DATA (white) or ROUTING (grey).
- Invariant: ROUTING nodes always have two children.
- Set consists of keys in DATA nodes.



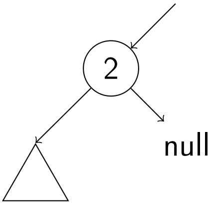
{1, 2, 5, 6, 8}

Traverse

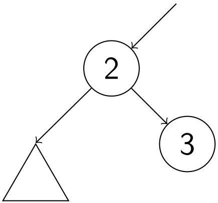
At the beginning of the operation we traverse a tree starting from the root to find a node with x to delete or a position to insert x:

- If the key in the current node is greater than x then go to the left subtree.
- If the key in the current node is less than x then go to the right subtree.
- If the key in the current node equals to x or the node is a leaf then stop.

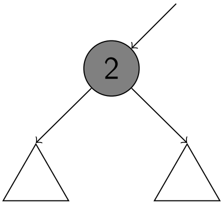
Sequential. Insert leaf. Insert(3)



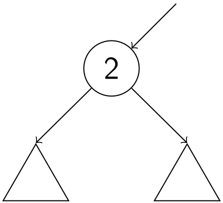
Sequential. Insert leaf. Insert(3)



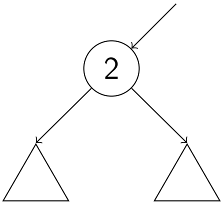
Sequential. Insert ROUTING. Insert(2)



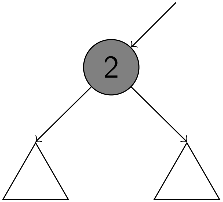
Sequential. Insert ROUTING. Insert(2)



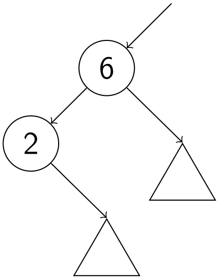
Sequential. Delete node with two children. Delete(2)



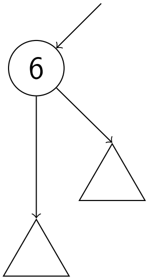
Sequential. Delete node with two children. Delete(2)



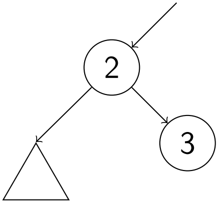
Sequential. Delete node with one child. Delete(2)



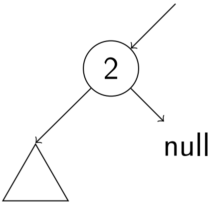
Sequential. Delete node with one child. Delete(2)



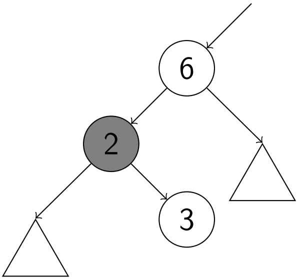
Sequential. Delete leaf with DATA parent. Delete(3)



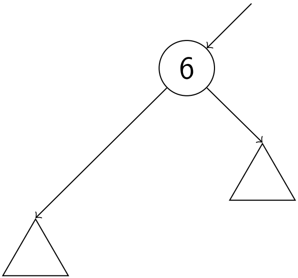
Sequential. Delete leaf with DATA parent. Delete(3)



Sequential. Delete leaf with ROUTING parent. Delete(3)



Sequential. Delete leaf with ROUTING parent. Delete(3)



Outline

Concurrent BST

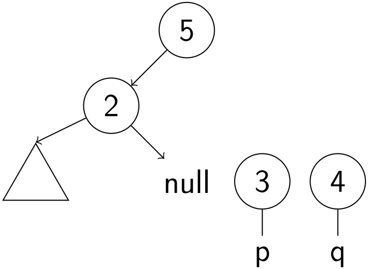
- Supports `Insert`, `Delete` and `Contains`.
- BST structure and invariants.
- Linearizability [Herlihy et al., TOPLAS 1990] with respect to Set type.
- State-of-the-art concurrent BSTs: [Bronson et al., PPoPP 2010], [Ellen et al., PODC 2010], [Crain et al., Euro-Par 2013], [Drachsler et al., PPoPP 2014], [Natarajan et al., PPoPP 2014].

Outline

Non-linearizable schedule

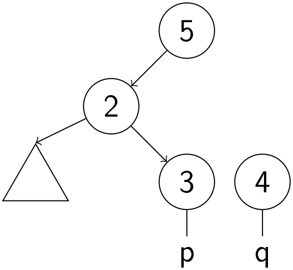
- Consider for a moment that we run sequential implementation in a concurrent environment.
- **Schedule** is an execution of the sequential algorithm in concurrent environment

p inserts 3 and q inserts 4. They traverse the tree and are ready to insert a leaf.



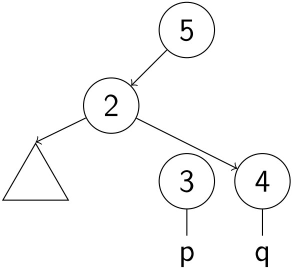
Non-linearizable schedule

p sets the leaf link to 3.

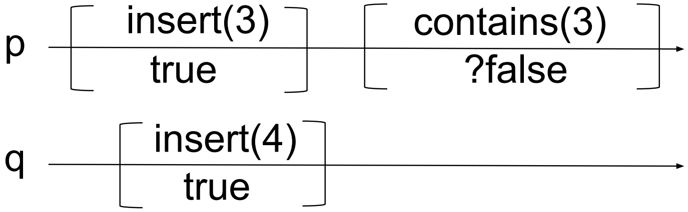


Non-linearizable schedule

q overwrites the link. `Insert(3)` is lost: `Contains(3)` returns `false`.



Non-linearizable schedule



Schedules

- Schedule is **accepted** if some execution of a concurrent implementation contains it as a subsequence.
  - Not all schedules should be accepted. (As the presented one)
- **Observably correct** schedules: the prefixes of the schedule are linearizable and the shared data structure is a BST.

Definition

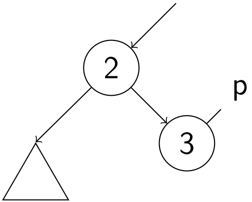
An implementation is **concurrency-optimal** if it accepts all observably correct schedules and only observably correct schedules.

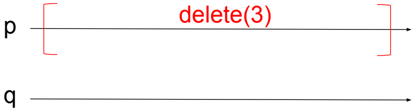
Intuitively: a concurrency-optimal BST employs as much synchronization as necessary for high-level correctness.

Interesting schedule

To illustrate the difficulty of designing a concurrency-optimal BST consider the following schedule.

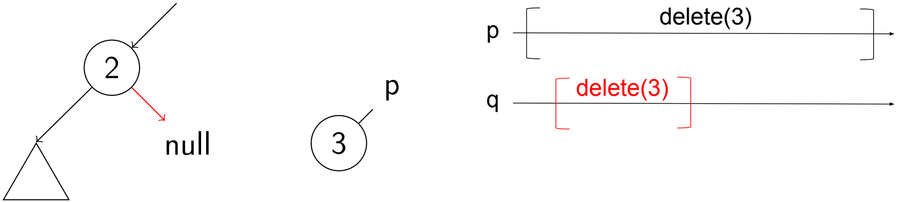
p invokes `Delete`(3), traverses to node 3 and falls asleep.





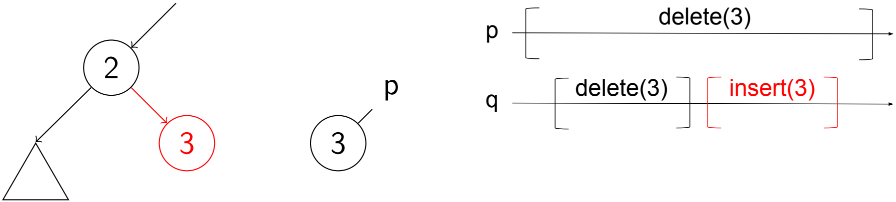
Interesting schedule

q invokes `Delete`(3), traverses to node 3, unlinks it and returns.



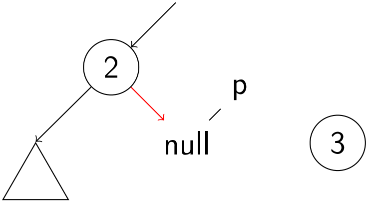
Interesting schedule

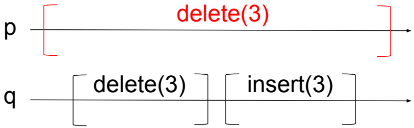
q invokes `Insert(3)`, links a new node 3 to node 2 and returns.



Interesting schedule

p wakes up and unlinks the **new node** from the tree.





Interesting schedule

- This schedule is **observably correct**.
- But it is not accepted by partially-external BSTs [Bronson et al., PPoPP 2010] and [Crain et al., Euro-Par 2013].
- External BSTs ([Ellen et al., PODC 2010], [Natarajan et al., PPoPP 2014]) and internal BST ([Drachsler et al., PPoPP 2014]) do not accept similar schedule.

Outline

Optimal implementation

- We perform everything optimistically: traverse, load all the necessary nodes and fields, such as state, choose the case.
- Right before the modification we take a lock on everything and check all the necessary conditions:
  - link is still present;
  - link goes to the node with the proper value;
  - proper state: DATA or ROUTING;
  - node is not removed, i.e., deleted mark is not set;
  - proper number of children.

Optimal implementation

- The critical section consists of one line of sequential implementation together with "wrapping block".
- Could be interleaved in any way if the conditions are satisfied.
- The conditions are satisfied if and only if the schedule is observably correct.
- **Such an implementation is concurrency-optimal.**

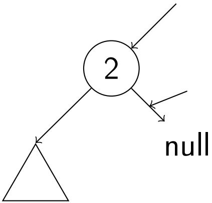
Additional optimizations.

- Can be optimized further.
- Accept more interleavings of the concurrent implementation.
  - Three locks: state, left and right children ([Natarajan et al., PPoPP 2014]);
  - Read/write locks.

Implementation

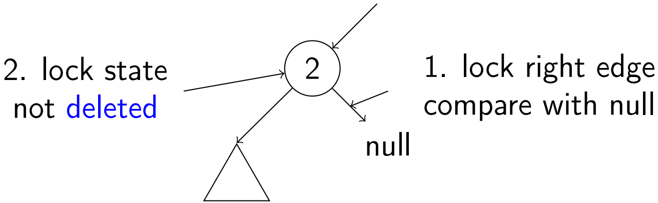
- Now, look into more details which locks are taken and which checks are performed in different cases.
- Assume that an update operation already traversed, read all the nodes and fields and chose the case.

Implementation. Insert leaf. Insert(3).

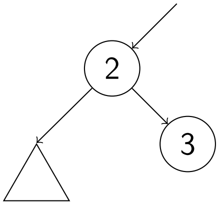


1. lock right edge
compare with null

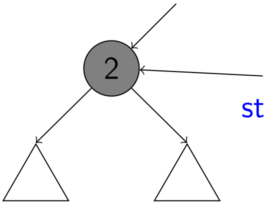
Implementation. Insert leaf. Insert(3).



Implementation. Insert leaf. Insert(3).

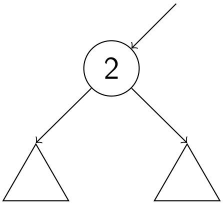


Implementation. Insert ROUTING. Insert(2).

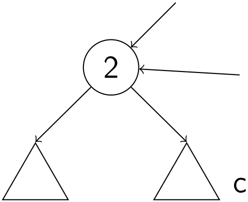


lock state
state == ROUTING
not deleted

Implementation. Insert ROUTING. Insert(2).

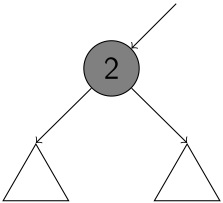


Delete node with two children. Delete(2)

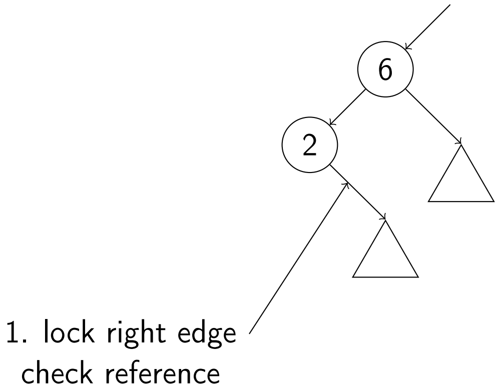


lock state
state == DATA
not deleted
check for 2 children

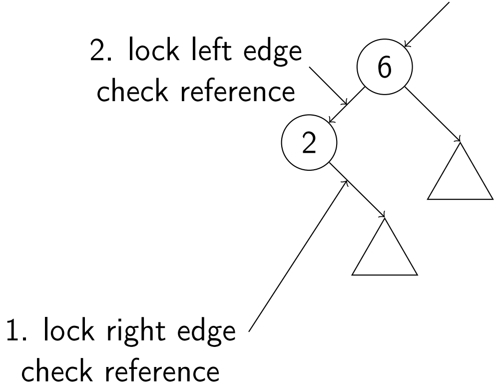
Delete node with two children. Delete(2)



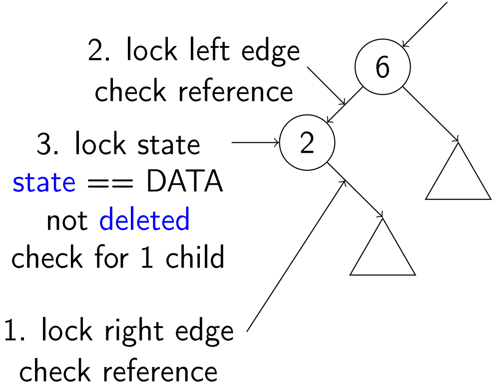
Delete node with one child. Delete(2)



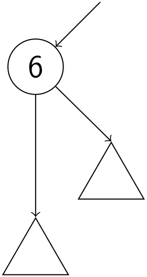
Delete node with one child. Delete(2)



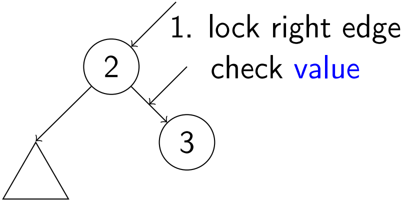
Delete node with one child. Delete(2)



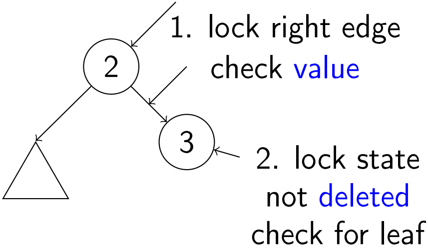
Delete node with one child. Delete(2)



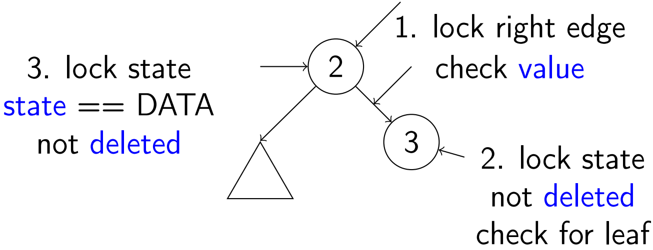
Delete leaf with DATA parent. Delete(3)



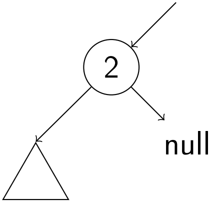
Delete leaf with DATA parent. Delete(3)



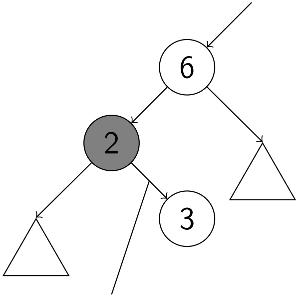
Delete leaf with DATA parent. Delete(3)



Delete leaf with DATA parent. Delete(3)

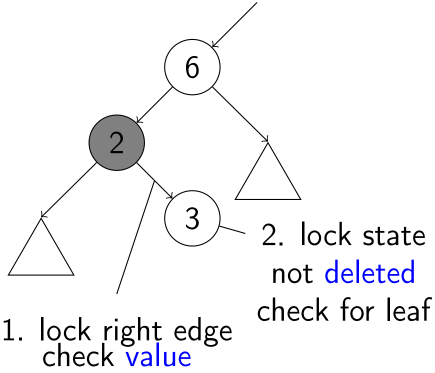


Delete leaf with ROUTING parent. Delete(3)

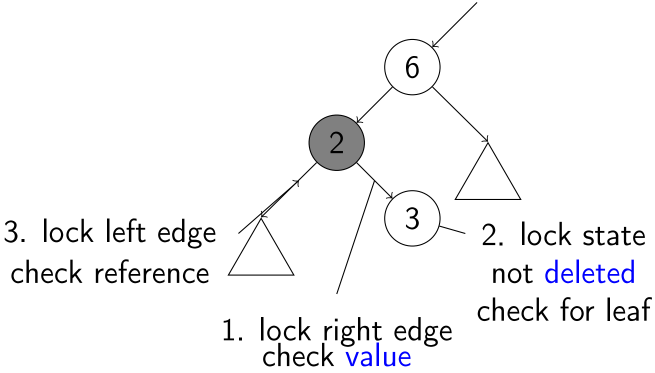


1. lock right edge
   check value

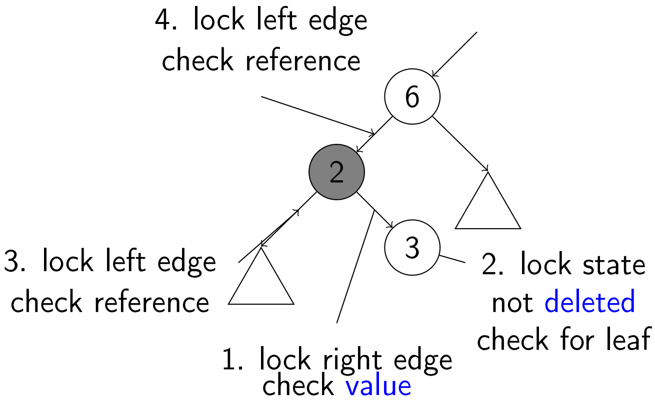
Delete leaf with ROUTING parent. Delete(3)



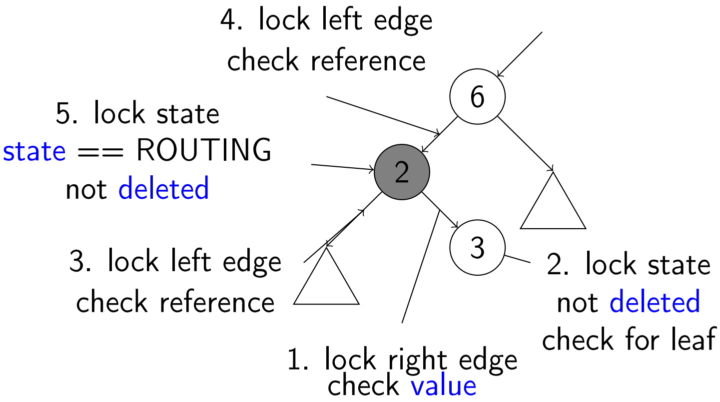
Delete leaf with ROUTING parent. Delete(3)



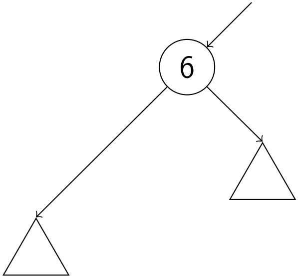
Delete leaf with ROUTING parent. Delete(3)



Delete leaf with ROUTING parent. Delete(3)



Delete leaf with ROUTING parent. Delete(3)



Outline

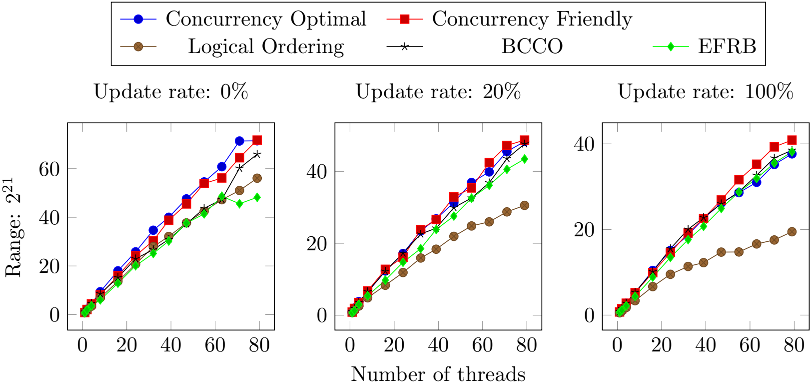
Settings

- 80-way Intel machine and 64-way AMD machine.
- Update ratio x: 0, 20, 100.
  - $\frac{x}{2}$% insert,
  - $\frac{x}{2}$% delete,
  - $100 - x$% contains.
- Value range: 2^{15}, 2^{19} and 2^{21}.
- The tree is prepopulated with range/2 values.
- Metric: throughput (operations per second).

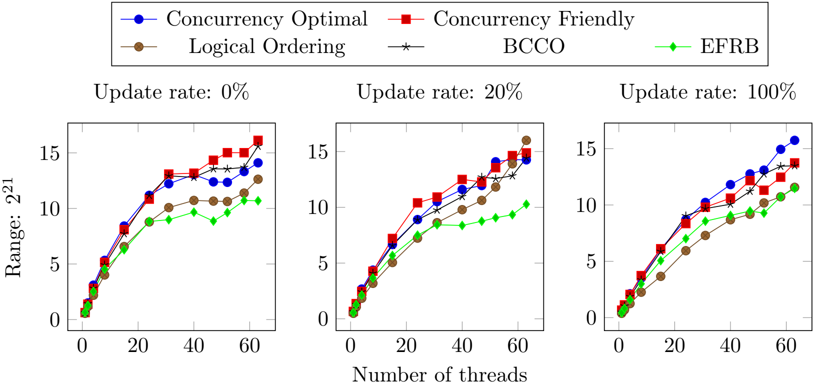
Algorithms

- Concurrency Friendly, [Crain et al., Euro-Par 2013];
- Logical Ordering, [Drachsler et al., PPoPP 2014];
- BCCO, [Bronson et al., PPoPP 2010];
- EFRB, [Ellen et al., PODC 2010].

Intel machine



AMD machine


Concurrency Optimal
Concurrency Friendly
Logical Ordering
BCCO
EFRB
Update rate: 0%
Update rate: 20%
Update rate: 100%
Range: 2^{21}
Number of threads
0
5
10
15
0
20
40
60

Outline

Conclusion

- Provably concurrency-optimal algorithm may perform well in practice.
- Concurrency-optimality could be an adequate design principle for efficient concurrent data structures. Besides BST, Linked List based Set [Gramoli et al., DISC 2015].
- Which other data structures could be optimized using this approach? What are the limitations?

Questions

Thank you for attention!