Balanced Non-blocking Binary Search Trees

Trevor Brown, supervised by Eric Ruppert
DisCoVeri Group, York University

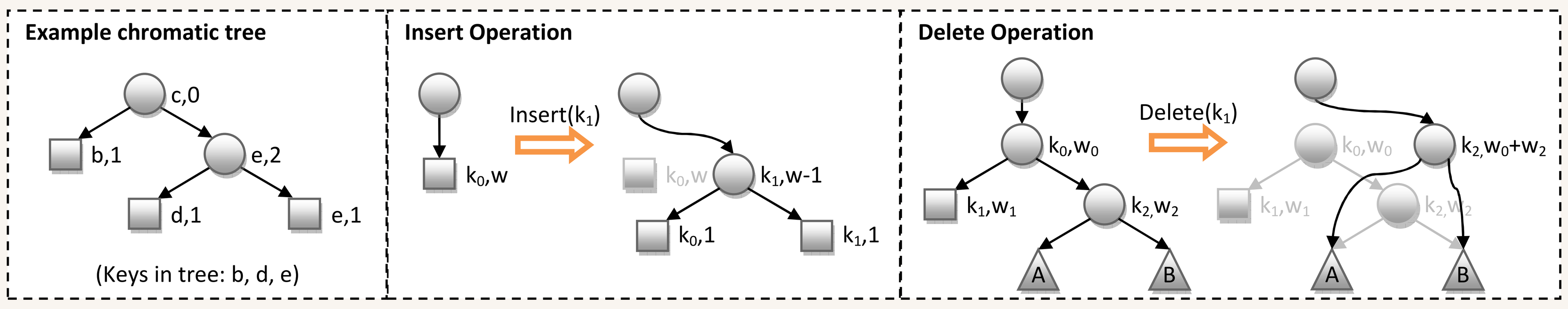
What We Created

- The first non-blocking, balanced binary search tree (BST) that is simultaneously accessible by many processes
  - **Balanced**
    Every leaf is $O(log(n) + F)$ steps from the root
    ($n = \#$ of keys in dictionary and $F = \#$ of accessing processes)
  - **Non-blocking**
    Guarantees that, infinitely often, some operation completes
    This implies the structure cannot use locks
  - **Linearizable**
    Every concurrent execution is consistent with some correct serial execution
- Model of computation
  - Asynchronous
  - Crash failures are possible
  - Shared memory with compare and swap instruction (CAS)

Related Work

- Non-blocking BST of Ellen, et al. [2]
  - *Helping:* Storing sufficient information at nodes to allow any process to complete an operation in progress (for non-blocking property)
  - *Flagging and marking:* CAS'ing helping information into a node before changing a child link or removing it from the tree (to coordinate processes)
- Chromatic search trees of Nurmi and Soisalon-Soininen [3]
  - Generalization of red-black trees allowing $weight > 1$
  - Decouple rebalancing from updates so that it can be delayed
- Amortized analysis of chromatic search trees by Boyar, et al. [1]
  - Presented a modified set of rotations
  - Proved any sequence of applicable rotations will restore balance
  - Proved bounds on number of rotations needed to restore balance

Tree Structure and Modifications

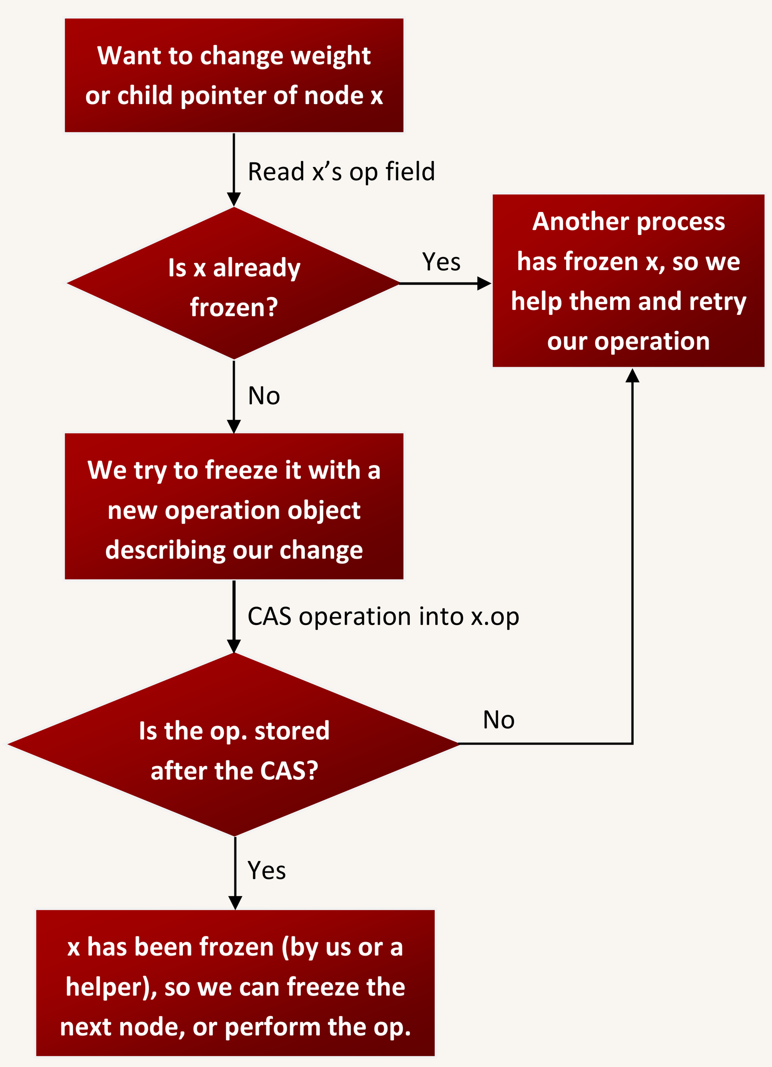


(In diagrams, square nodes are leaves; round nodes are leaves or internal nodes.)

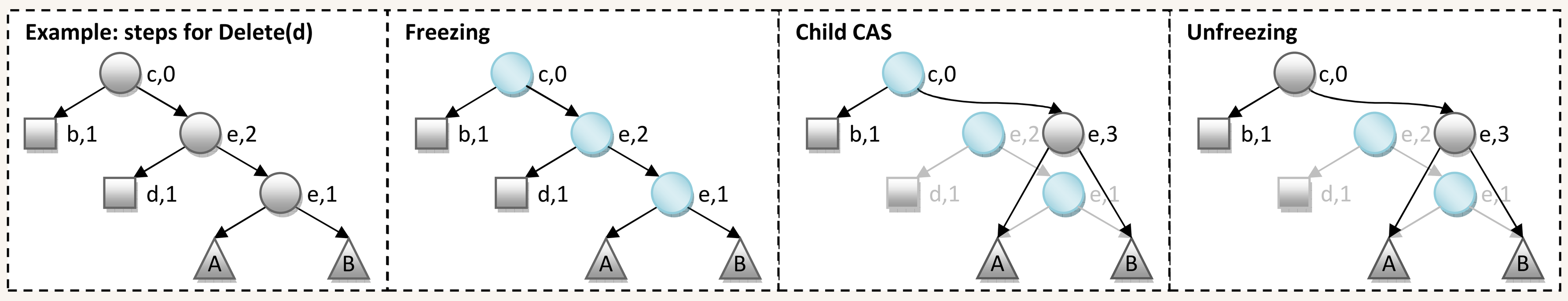
- Structure is a leaf-oriented tree
  - Full binary tree with internal nodes to guide searches to leaves
  - Set of keys in dictionary = set of keys in leaves
- Weights used to maintain balance
  - 0 and 1 correspond to red and black in a red-black tree
  - Red-red violations as in familiar red-black trees
  - Red-black trees have only limited overweight violations ($weight = 2$)
  - Chromatic trees allow arbitrary overweight violations ($weight > 2$)
- Operations create replacement sub-trees of entirely new nodes, and atomically splice them into the tree with a single CAS
  - Insert replaces a leaf with sub-tree containing three new nodes
  - Delete replaces a sub-tree with a new copy of the remaining node (It removes the deleted leaf and a deprecated internal routing node)
  - Rotations are similar, with replacement sub-trees of varying size
  - Maintains consistent data for "stranded" searching processes

Freezing

- Is a generalization of flagging and marking
  - Before an operation can modify a node, it must freeze it
  - A node is frozen when its op field refers to an active operation object
  - Guarantees atomicity of updates
- How is it used in the algorithm?
  - A node must be frozen before being modified or removed from the tree
  - All processes respect freezing, and once an operation's nodes are entirely frozen, it will be completed before any of its nodes are unfrozen
  - Searches can safely ignore freezing, updates, and rotations, and proceed exactly as in the sequential case.

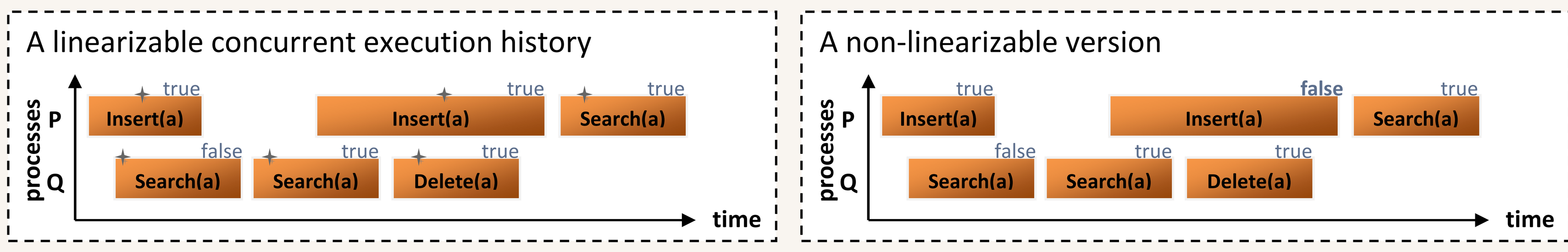


Algorithm for Insert and Delete



- **Algorithm overview**
  - Create an Operation object op
  - Call Help(op) to perform all helping steps
  - If the operation is flagged $Retry$, then restart the algorithm
  - Otherwise, Help successfully completed the operation, so
    - If the operation created any violations, call Clean-up
- **Help(op) steps for an Operation**
  - Any process can perform these steps to help the operation complete
  - *Freezing:* all affected nodes are frozen in sequence
  - *Child CAS:* a new replacement for the frozen sub-tree is swapped in
  - *Unfreezing:* the parent of the replaced sub-tree is unfrozen
- **What if a step cannot be completed?**
  - If the step was completed by another process, move to the next step
  - Otherwise
    - If another operation was blocking ours, call Help on it
    - Unfreeze all nodes, and set the operation's retry flag
- **The clean-up phase**
  - Initiated by the process P that invoked the INSERT or DELETE
  - As above, P creates op (appropriate rotation operation) and calls Help
  - However, instead of restart on a Retry flag, P attempts to fix all violations on the search path to the leaf updated by the INSERT or DELETE
  - This phase ends as soon as P does one of the following
    - Perform a rotation that does not create a violation
    - Encounter no violations along the path from the root to the leaf

Linearizability and Correctness



- Linearizability is a correctness condition for concurrent algorithms
- A *concurrent execution history* (sequence of operation invocations and responses) is linearizable if one can choose a "linearization point" within each operation (see stars in the above diagram) such that:
  - A correct, sequential execution of the operations, ordered by these points, produces operation responses consistent with the history
- An *algorithm* is linearizable if all possible execution histories are linearizable
- For instance, in this algorithm, the linearization points are:
  - For an INSERT or DELETE, the precise moment the new sub-tree is CAS'ed into the tree structure
  - For a SEARCH(K), a point when the leaf was on the search path to key k (It is proven by induction that every node visited by a search to key k was on the search path to k at some point during the search.)

Future Work

- Implementation and experimental performance evaluation
- Theoretical performance analysis
- Formal proof of correctness
- Performance improvements
- Making operations wait-free
- Adding more operations (predecessor, successor, clone, etc.)

Bibliography

[1] J. Boyar, R. Fagerberg, K. S. Larsen, Amortization results for chromatic search trees, with an application to priority queues. In *Proc. 4th Intl. Workshop on Algorithms and Data Structures*, WADS '95, pp. 270–281, 1995.

[2] F. Ellen, P. Fatourou, E. Ruppert, F. van Breugel, Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pp. 131–140, 2010.

[3] O. Nurmi, E. Soisalon-Soininen, Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica*, 33(6):547–557, 1996.