Relativistic Red-Black Trees

Philip Howard
4/28/2010
pwh@cecs.pdx.edu
Red-Black Trees

- **Root Property**: The root is black
- **External Property**: Every external node is black
- **Internal Property**: The children of red nodes are black
- **Depth Property**: All external nodes have the same black depth
Claim

• $O(1)$ restructures after insert/delete
• Possibly $O(\log(n))$ recolors
• $O(\log(n))$ performance
Insertion

• Search until we get to external node
• Insert at this location adding two empty, black, external nodes underneath
• If root, color black, else color red

• Preserves root, external, and depth properties, may violate internal property
Double Red case 1

- $z$ is the new node
- $v,u$ are parent and grandparent of new node
- Sibling $w$ of $v$ is black
Double Red case 1

• Re-label u,v,z as a,b,c in left to right order (in-order traversal order)

• Restructure as:
Double red (case 2)

• Sibling of v is red
Double red (case 2)

- Re-color u red, w,v black
- Recurse up the tree as needed
Delete

• Find node

• If node doesn’t have an external child, swap with next internal node in in-order traversal order (left most node of the right branch)

• Delete the node from the bottom
Swap example: Delete 5
Delete (cont)

- Remove $v$ with external child $w$
- $r$ is sibling of $w$
- $x$ is parent of $v$

- Remove $v$ and $w$
- Make $r$ a child of $x$
Delete (cont)

• v was red XOR r was red, color r black

• v was black AND r was black, color r double-black
Double Black case 1

• Sibling y of r is black and has a red child z
• restructure(z)

• a and c are black
• b is former color of x
• r is black
Double Black (case 2)

- Sibling \( y \) of \( r \) is black
- Both children of \( y \) are black
- Recolor

\[
\begin{align*}
\text{color } r \text{ black} \\
\text{color } y \text{ red} \\
\text{If } x \text{ is red, color it black; else color it double black}
\end{align*}
\]
Double Black (case 3)

- Sibling y of r is red
- if y is right child of x, z is right child of y
- if y is left child of x, z is left child of y
- restructure(z) (note y is always b)
Double Black (case 3)

- restructure(z)  (note y is always b)
- color y black, x red

- Repeat case 1 or 2 (note that case 2 can’t cascade)
Performance

• Find, Insert, Remove
  \( O( \log(n) ) \)
• At most 1 restructure on insert
• At most 2 restructurues on delete
• Possibly \( \log(n) \) recolors
Relativistic Requirements

• Readers don’t block, and are not inhibited by writers
• Lookups always find a node that exists in the tree
• Traversals always return nodes in the correct order without skipping any nodes that exist in the tree
nodes that “exist in the tree”

• Every node that was in the tree at the beginning of a read and was not deleted during the read

• Some nodes that were deleted during a read

• Some nodes that were inserted during a read
Assumptions

- Readers ignore the color of nodes
- Readers don’t need to access the parent pointers (note: some traversal algorithms require access to parent pointers)
- If keys are unique, temporarily having a duplicate node in the tree won’t affect reads
- Mutual exclusion used between writers
<table>
<thead>
<tr>
<th>Operations</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>No nodes are moved or freed so this operation is safe.</td>
</tr>
<tr>
<td>Recolor</td>
<td>Recoloring does not affect the read consistency of the tree.</td>
</tr>
<tr>
<td>Delete</td>
<td>Need to defer reclamation of memory until no thread has a reference to the deleted node.</td>
</tr>
<tr>
<td>Swap</td>
<td>Need to guarantee that the node which moves up isn’t missed by readers between the new and old position.</td>
</tr>
<tr>
<td>Restructures</td>
<td>Requires moving nodes. Need to ensure that no readers get lost in the transition.</td>
</tr>
</tbody>
</table>
Swap example: Delete 5
Swap algorithm

find node to delete
find swap candidate
create swap-prime
link swap-prime into tree
wait a grace period
remove swap from tree
Restructure
Restructure – diag right
Diag algorithm

Create copy of A. Label copy A’.
A’ children are child of a and child of B
link A’ into tree as child of B
link around A: B becomes child of up
Restructure – zig right
zig algorithm

Create copy of B. Label copy B’
B’ children are A and C
B’ becomes child of UP
Wait a grace period
Children of B become children of A and C
Traversals

• $O(n)$: requires structure of tree to be maintained (no updates allowed)
  – Use parent pointers to find next()

• $O(n \log(n))$: allows concurrent updates
  – Start search for next() at root of tree
## Performance

<table>
<thead>
<tr>
<th>lock</th>
<th>No synchronization – NOT A VALID IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>nolock</td>
<td>pthread mutex lock</td>
</tr>
<tr>
<td>rwlr</td>
<td>Reader-Writer lock that favors readers</td>
</tr>
<tr>
<td>rwlw</td>
<td>Reader-Writer lock that favors writers</td>
</tr>
<tr>
<td>rp</td>
<td>Relativistic Programming implementation</td>
</tr>
</tbody>
</table>

4/27/2010
Read Performance

Red/black read tree performance

- noLock read 64K
- noLock wread 64K
- rp read 64K
- rp wread 64K
- rwlr read 64K

Operations/sec vs. Threads

4/27/2010
Write Performance

Red/black write tree performance

- no lock write 64K
- rp write 64K
- rwlw write 64K
- lock write 64K
- rwr write 64K
Conclusions

What we gained

- Linear scalability for readers
- Writer doesn’t interfere with read performance
- Read performance approaches that of nolock
- Contended write performance exceeds that of other valid synchronization methods
Conclusions

What we gave up

- Uncontended write performance is worse than with other synchronization methods
- Readers may see updates in different orders
When is it OK for readers to see updates in different orders?

When updates are independent or commutative

Example: phone book

deleting a customer and adding a customer are independent UNLESS the new customer gets the old customer’s phone number
On going work

- Benchmark against a TM-ish, NBS-ish concurrent algorithm
- Remove the grace period from restructures
- Remove the grace period from swap
- Allow multiple concurrent updates
- $O(n)$ traversal with concurrent updates
- Answer the general question: What makes an algorithm/data structure RP friendly?