Left-Leaning Red-Black Trees Considered Harmful

(This is read.seas.harvard.edu/~kohler/notes/llrb.html)

Eddie Kohler

Robert Sedgewick's left-leaning red-black trees are supposedly simpler to implement than normal red-black trees:

> In this paper, we describe a new variant of red-black trees that meets many of the original design goals and leads to substantially simpler code for insert/delete, less than one-fourth as much code as in implementations in common use.

Sounds exciting, and a lot of LLRB implementations are kicking around.

But are they simpler? Jason Evans:

> ### Left-leaning red-black trees are hard to implement
>
> Last Monday, I started implementing left-leaning red-black trees, expecting to spend perhaps 15 hours on the project. I'm here more than 60 hours of work later to tell you that left-leaning red-black trees are *hard* to implement, and contrary to Sedgewick's claims, their implementation appears to require approximately the same amount of code and complexity as standard red-black trees. Part of the catch is that although standard red-black trees have additional cases to deal with due to 3-nodes that can lean left or right, left-leaning red-black trees have a universal asymmetry between the left and right versions of the algorithms.

Jason Evans is more right than Robert Sedgewick. Here are a couple things you should think about before implementing left-leaning red-black trees.

Tricky writing

Sedgewick's paper is tricky. As of 2013, the insert section presents 2–3–4 trees as the default and describes 2–3 trees as a variant. The delete implementation, however, *only works for 2–3 trees*. If you implement the default variant of insert and the only variant of delete, your tree won't work. The text doesn't highlight the switch from 2–3–4 to 2–3: not kind.

Parent pointers

Sedgewick's left-leaning red-black trees are implemented without parent pointers. "The most popular [deletion] method in common use [for conventional RB trees] is based on a parent pointers…, which adds substantial overhead and does not reduce the number of cases to be handled." His slides put "(??)" or "(!)" after mentions of parent pointers. But parent pointers add functionality as well as overhead, and they aren't that expensive.

Iteration. Parent pointers greatly simplify iteration over ranges of objects in a tree. Given parent pointers, it's simple to find the successor or predecessor of any node in the tree. Each successor or predecessor call

takes $O(1)$ amortized time, so traversing the entire tree takes the minimum $O(n)$ time.

You can get the same properties without parent pointers (or other links between nodes), but only sort of. $O(1)$ iteration requires the ability to step backwards in the tree, so an LLRB iterator would have to include an ancestry list up to the root. These "fat" iterators are much larger than node pointers.

And using iterators becomes harder too. C++ standard containers, such as map, offer an extremely useful *iterator validity* guarantee: a tree iterator remains valid even as the tree grows and shrinks. (The obvious exception is that removing an item invalidates all iterators pointing at that item.) Ensuring this guarantee with fat iterators would be very very painful; I doubt you ever would.

Performance. I have several red-black tree implementations based on the same choices (intrusive nodes, and a node's color bit is stored in *the pointer to that node* rather than the node itself). A simple benchmark: add a million random integers to the tree; find 4,000,000 random integers in that tree (all finds succeed); and then remove the million integers in random order. We compare an LLRB implementation without parent pointers, to the same implementation with parent pointers. (Insert and remove are implemented recursively.) Parent pointers are slower, but just by a bit.

- Insert phase: 0.556s without parents, 0.576s with parents (1.03x)
- Find phase: 1.656s without parents, 1.672s with parents (1.01x)
- Delete phase: 1.024s without parents, 1.108s with parents (1.08x)

Jason Evans reports that removing parent pointers reduces memory usage in jemalloc by 0.4%. In my benchmark it'd be larger, as nodes only contain ints.

Additional rotations

The red-black tree insert and delete algorithms presented in *Introduction to Algorithms* (CLRS) strictly bound rotations: insert performs at most two rotations, and delete performs at most three. What about left-leaning trees?

First, clearly (although Sedgewick doesn't emphasize this), LLRB trees' stricter invariants *must* cause more rotations. There are exponentially fewer valid LLRB trees than valid RB trees, so inserts and deletes that would have created perfectly fine RB trees will cause additional rotations to enforce the left-leaning invariant. Each rotation means additional memory writes. I would expect a constant factor more rotations: still the same computational complexity, but measurably slower.

But that's not all. Sedgewick's "simple" recursive delete algorithm rotates *both* on the way *down* the tree, towards the victim item, *and* on the way *back up* the tree, towards the root (in the "fixUp" procedure). Most of these proactive rotations are unnecessary, and I think many of the top-down rotations are undone on the way back up!

This matters for performance. I measured the number of rotations per operation in the insert–find–delete benchmark above.

- Insert phase: Normal RB 0.582 rotations/insert, LLRB 1.725 rotations/insert (2.96x—probably a constant factor)
- Delete phase: Normal RB 0.380 rotations/delete, LLRB **19.757** rotations/delete (51.99x!!—both a constant factor and a log-N factor)

Just crazy. (And there's a similar difference in the number of "flips" per operation.) Sedgewick's slides say

“Left-leaning red-black trees—less code implies faster insert, delete” (slide 68), unconventionally using “implies” to mean “provides the opposite of.”

And rotations aren’t the only delete performance boner either. Since there are no parent pointers, Sedgewick requires $O(\log n)$ *comparisons* to delete a item from the tree, while conventional RB trees with parent pointers require exactly 0. If comparisons are expensive (and they often are) this matters.

A more complex LLRB delete implementation could avoid many rotations. (Jason Evans’s rb-newer does, for instance, as does this one.) But such implementations still impose the inherently larger number of rotations required to lean left, and the $O(\log n)$ comparisons required when there are no parent pointers.

Performance

Here are some overall performance numbers.

- Insert phase: conventional RB 0.476s, LLRB 0.560s (1.18x)
- Find phase: conventional RB 1.648s, LLRB 1.680s (1.02x)
- Delete phase: conventional RB 0.612s, LLRB 1.032s (1.69x)

That is, conventional wins. The find code is exactly the same in both trees. The normal RB tree has parent pointers and is competitive with boost::intrusive::rbtree (which has insert 0.448s (0.94x my RB, i.e. faster), find 1.872s (1.14x slower), delete 0.676s (1.10x slower)). Extra rotations hurt LLRB insert and delete. Both insert and delete are implemented recursively in LLRB and iteratively in conventional RB, but the recursion itself doesn’t seem to be a big cost (for example, an iterative implementation of delete didn’t help).

Simplicity

Sedgewick doesn’t necessarily claim to care about performance (he should mention it more than he does). He does care about simplicity. But some of his LLRB implementation’s simplicity is independent of the central left-leaning idea, and some aspects of LLRB trees are *more* complex than conventional RB trees.

“Full implementations are rarely found in textbooks, with numerous ‘symmetric’ cases left for the student to implement.” This is true. But this symmetry indicates *simplicity*, not complexity. If left and right cases are handled identically, then one can code *one case* with boolean variables that distinguish between left and right. For instance, CLRS say, in RB-Insert,

```
 4   ... if p[x] = left[p[p[x]]]
 5         then y ← right[p[p[x]]]
 6              if color[y] = RED
 7                 then color[p[x]] ← BLACK
 8                      color[y] ← BLACK
 9                      color[p[p[x]]] ← RED
10                      x ← p[p[x]]
11              else if x = right[p[x]]
12                      then x ← p[x]
13                           Left-Rotate(T, x)
14                   color[p[x]] ← BLACK
15                   color[p[p[x]]] ← RED
16                   Right-Rotate(T, p[p[x]])
17       else (same as then clause with “right” and “left” exchanged)
```

with a symmetric case. But they could have written this instead. Assume that `child[x, side]` equals

`right[x]` if `side = TRUE` and `left[x]` otherwise (this is trivial to code; perhaps the child pointers are a two-element array); and assume that `Rotate(T, n, side)` behaves like `Rotate-Left(T, n)` or `Rotate-Right(T, n)` accordingly. Then this code is complete, with no so-called "case bloat":

```
 4   ... side ← (p[x] = right[p[p[x]]])
 5       y ← child[p[p[x]], !side]
 6       if color[y] = RED
 7          then color[p[x]] ← BLACK
 8               color[y] ← BLACK
 9               color[p[p[x]]] ← RED
10               x ← p[p[x]]
11          else if x = child[p[x], !side]
12                  then x ← p[x]
13                       Rotate(T, x, side)
14               color[p[x]] ← BLACK
15               color[p[p[x]]] ← RED
16               Rotate(T, p[p[x]], !side)
```

You don't need to implement Left-Rotate and Right-Rotate separately, a single Rotate function will do. Once the code works for one case it must work for the other symmetric case.

But left-leaning red-black trees *are asymmetric*, so the code for left and right *must differ*. There are not fewer cases to consider, there are *more*.

The LLRB implementation contains some nice simplifying ideas. Luckily, some of them apply just as well to conventional red-black trees. Assume that, as in Sedgewick's LLRB implementation, the `Rotate` operation introduces a red link, and a `Flip` operation flips a node's color and the colors of its children. Then the above code snippet reduces to:

```
 4   ... side ← (p[x] = right[p[p[x]]])
 5       y ← child[p[p[x]], !side]
 6       if color[y] = RED
 7          then Flip(p[p[x]])
 8               x ← p[p[x]]
 9          else if x = child[p[x], !side]
10                  then x ← p[x]
11                       Rotate(T, x, side)
12               Rotate(T, p[p[x]], !side)
```

For an algorithms textbook, "left" and "right" are concrete and useful; might as well keep the symmetric cases. In an actual implementation, though, boolean sides wins. Then again so do Sedgewick's helpers.

In practice iterative conventional RB code will be longer than recursive LLRB code, but not by much. In my implementation, the iterative conventional RB header file has 640 non-comment/non-blank lines. The LLRB header file with recursion and without parent pointers has ~562. Most of these lines are boilerplate shared by both implementations, I wrote the LLRB implementation first, and the conventional implementation has more functionality (iteration over ranges). 80 lines is not a huge difference.

Conclusions

The simplest code is the code you didn't write. So use someone else's red-black tree (or other search tree structure) if you can. But in life sometimes we need and/or want to write a red-black tree. For instance, maybe the boost::intrusive tree doesn't offer the "reshaping" hooks you'd need to implement an interval tree. Or maybe you just like coding.

Whether you're choosing or writing a red-black tree, go conventional, not left-leaning. The exception: if you very rarely or never iterate over tree ranges, you never need to combine iteration with modification, your nodes are small (so parent pointer overhead is significant), your comparison function is cheap (or perhaps you never delete anything), and your workload is very heavily biased towards find, then LLRB trees *might* win. But know that the optimized LLRB implementation you choose will be just as complex as a well-implemented conventional tree, if not more so.