Red-Black Trees Revisited

Cameron Moy

Northeastern University, Boston MA 02115, USA
camoy@ccs.neu.edu

Abstract. For more than two decades, functional programmers have studied and refined the persistent red-black tree—a data structure of unrivaled elegance. This paper presents another step in its evolution by optimizing insertion and simplifying deletion.

Keywords: Algorithms · Data Structures · Trees

1 A Quick Recap

A red-black tree is a *self-balancing* binary search tree [5,1]. Operations rebalance the tree so it never becomes too lopsided. To do so, every node carries an extra bit that "colors" it either red or black:

```
data Color = Red | Black
data Tree a = E | N Color (Tree a) a (Tree a)
```

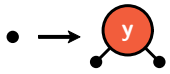
Insertion and deletion use chromatic information to maintain two invariants:

1. *Red-red invariant.* A red node may not have a red child.
2. *Black-height invariant.* The number of black nodes along all paths through the tree (the black height) is the same.

These two properties imply that the tree is roughly balanced. The longest possible path through the tree, alternating red and black nodes, is at most twice as long as the shortest path, containing just black nodes. Naively inserting or deleting nodes from the tree may violate these invariants. Hence, the challenge of implementing red-black trees is repairing the invariants after a modification.

2 Insertion à la Okasaki

Let's review the insertion algorithm of Okasaki [7]. In an ordinary binary search tree, insertion works by traversing the tree and replacing a leaf with the desired value. For a red-black tree, insertion's first step is the same, with the new node colored red:



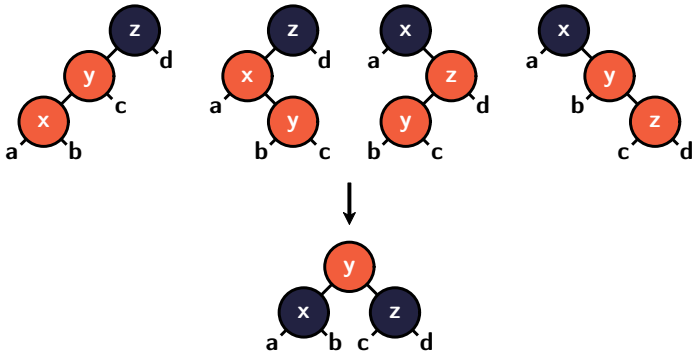


Fig. 1. Balance

Doing so may introduce a red-red violation if the leaf's parent happens to be red. A `balance` function resolves such red-red violations. A violation can come in one of four shapes that `balance` eliminates in the same way; see Figure 1.

To simplify the definition of `balance`, nodes of each color can be constructed and matched using pattern synonyms:

```
pattern R a x b = N Red a x b
pattern B a x b = N Black a x b
```

Realizing Figure 1 as code is now a straightforward, if tedious, exercise:

```
balance (B (R (R a x b) y c) z d) = R (B a x b) y (B c z d)
balance (B (R a x (R b y c)) z d) = R (B a x b) y (B c z d)
balance (B a x (R (R b y c) z d)) = R (B a x b) y (B c z d)
balance (B a x (R b y (R c z d))) = R (B a x b) y (B c z d)
balance s = s
```

Since `balance` can turn the top-most black node red, this may induce a red-red violation one level up the tree. Thus, `insert` must `balance` at every level. This "bubbles" the violation all the way up the tree. At the end, `insert` blackens the root to resolve the last potential violation:

```
insert x s = (blacken . ins) s
  where ins E = R E x E
        ins (N k a y b)
          | x <  y = balance (N k (ins a) y b)
          | x == y = N k a y b
          | x >  y = balance (N k a y (ins b))

blacken (N _ a y b) = B a y b
```

3 Insertion, Faster

When `balance` produces a black node, it can't possibly induce a red-red violation further up the tree. Since the rest of the tree satisfies the red-red invariant, there's no more work to be done; every subsequent `balance` is unnecessary.

If there were a way to short circuit those balances, `insert` could avoid much unneeded pattern matching. A new data type[1] makes this possible:

```
data Result a b = D a | T b
```

A `Result` contains a tree where either the work is **d**one, constructed with `D`, or there is more work **t**o do, constructed with `T`. Trees marked with `D` don't violate a red-black tree invariant, while trees marked with `T` may. Trees marked with `D` can be passed forward unaffected, while trees marked with `T` must be fixed by calling functions like `balance`.

A `Monad` instance for `Result` supports this use case. A tree where more work needs to be done will be given to the function `f`, while a tree that's done will propagate:

```
instance Monad (Result a) where
  return x = T x
  (D x) >>= f = D x
  (T x) >>= f = f x
```

Two functions on `Result` values will also prove useful. The `fromResult` function extracts trees from a `Result`

```
fromResult (D x) = x
fromResult (T x) = x
```

and `<$$>` applies a function to both sides of a `Result`[2]

```
f <$$> (D x) = D (f x)
f <$$> (T x) = T (f x)
```

Equipped with `Result`, suspended calls to `balance` further up the tree can be bypassed by wrapping a subtree in `D`. As mentioned before, it's safe to do so whenever `balance` produces a black node. After this point, no red-red violations are possible; see the highlighted case below. Here is the new `balance` function:

```
balance (B (R (R a x b) y c) z d) = T (R (B a x b) y (B c z d))
balance (B (R a x (R b y c)) z d) = T (R (B a x b) y (B c z d))
balance (B a x (R (R b y c) z d)) = T (R (B a x b) y (B c z d))
balance (B a x (R b y (R c z d))) = T (R (B a x b) y (B c z d))
balance (B a x b) = D (B a x b)
balance (R a x b) = T (R a x b)
```

[1] This type is the same as `Either`, but with more convenient constructors.

[2] Note that `<$$>` is not `fmap`. The functor instance implied by the monad applies a function only to `T` values.

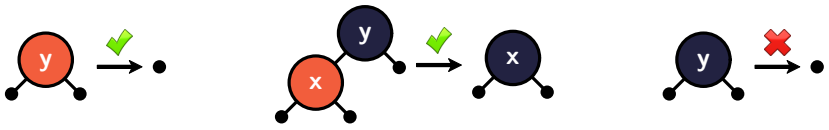
Now that `balance` returns a `Result` value, `insert` must handle it. The essence of this function, however, remains the same:

```
insert x s = (blacken . fromResult . ins) s
  where ins E = T (R E x E)
        ins (N k a y b)
          | x <  y = balance =<< (\a -> N k a y b) <$$> ins a
          | x == y = D (N k a y b)
          | x >  y = balance =<< (\b -> N k a y b) <$$> ins b
```

What's to be gained for all this trouble? Without much effort, insertion can be up to 1.5× faster using this approach. See Section 5 for details.

4 Deletion, Simpler

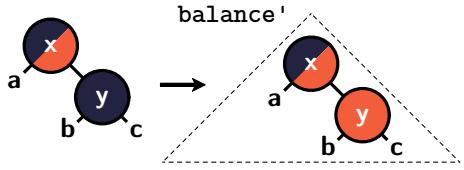
Let's turn our attention to `delete`. As with `insert`, the function starts off like any old binary search tree algorithm. If possible, it replaces the target node with its in-order successor. Otherwise, it's at a base case where no right child exists. The following diagram shows all three base cases:



Deleting a red node doesn't introduce a black-height violation, but deleting a black node might if its left child can't be blackened. Recall that `insert` always adds a red node, *possibly* causing a red-red violation. Subtrees are wrapped in `T` if there *might* be a violation and `D` if there isn't. For `delete`, the right-most base case *always* causes a black-height deficit. Thus, subtrees are wrapped in `T` if there *definitely* is a deficit and `D` if there isn't.

Just like `insert` needs `balance`, `delete` needs a function that can repair the black-height invariant at every level of the tree. That's the job of `raiseL` and `raiseR`. These functions attempt to increase the black height of the left and right child, respectively. If it can't, it equalizes the black heights and bubbles the violation up.

Consider `raiseL`, where the left child, labeled `a`, is black-height deficient. There are two cases to consider: when its sibling is black and when its sibling is red. Here's the first case, where the sibling is black and the root is any color:



To equalize the black heights, `raiseL` reduces the black height of the right child by reddening it. Now the whole tree is deficient. Not only that, but this can introduce a red-red or even the dreaded *red-red-red* violation. A variant of `balance`, called `balance'`, is responsible for handling both of these issues:

```
raiseL (N k a y (B c z d)) = balance' (N k a y (R c z d))
raiseR (N k (B a x b) y c) = balance' (N k (R a x b) y c)
```

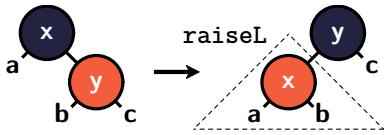
The purpose of `balance'` is threefold: resolve red-red violations, resolve red-red-red violations, and increase the black height by one if possible. To accomplish this, it acts like `balance` except the root color is preserved in the case of a violation and blackened otherwise. Differences compared to `balance` are highlighted:

```
balance' (N k (R (R a x b) y c) z d) = D (N k (B a x b) y (B c z d))
balance' (N k (R a x (R b y c)) z d) = D (N k (B a x b) y (B c z d))
balance' (N k a x (R (R b y c) z d)) = D (N k (B a x b) y (B c z d))
balance' (N k a x (R b y (R c z d))) = D (N k (B a x b) y (B c z d))
balance' s = blacken' s
```

Unlike `balance`, `balance'` never introduces a red-red violation further up the tree since it never turns a black node red. Additionally, if the top-most node is red, then `balance'` always returns a `D` result. It only returns `T` when `blacken'` encounters a black node:

```
blacken' (R a y b) = D (B a y b)
blacken' s = T s
```

Next, let's look at the case where the sibling is red. Here, `raiseL` applies a rotation that doesn't affect any black heights and then raises the left child recursively:



After the rotation, **a** is still deficient and the other subtrees are unchanged. However, as noted before, `balance'` resolves a black-height violation when called on a red node. Thus, it's guaranteed that the recursive call to `raiseL` will successfully increase the black height of **a**, yielding a valid red-black tree:

```
raiseL (N k a y (R c z d)) = (\a -> B a z d) <$$> raiseL (R a y c)
raiseR (N k (R a x b) y c) = (\b -> B a x b) <$$> raiseR (R b y c)
```

That's it. Figure 2 contains the rest of the code, which composes the presented functions into a complete algorithm.

```
delete x s = (fromResult . del) s
  where del E = D E
        del (N k a y b)
          | x <  y = raiseL =<< (\a -> N k a y b) <$$> del a
          | x == y = delCur (N k a y b)
          | x >  y = raiseR =<< (\b -> N k a y b) <$$> del b

delCur (R a y E)   = D a
delCur (B a y E)   = blacken' a
delCur (N k a y b) = raiseR =<< (\b -> N k a min b) <$$> b'
   where (b', min) = delMin b

delMin (R E y b)   = (D b, y)
delMin (B E y b)   = (blacken' b, y)
delMin (N k a y b) = (raiseL =<< (\a -> N k a y b) <$$> a', min)
   where (a', min) = delMin a
```

Fig. 2. Delete

5 Performance Evaluation

Using monads to communicate balancing information yields a unified and elegant presentation of both insertion and deletion; critically though, these variants also perform as well or better than existing algorithms. Figure 3 and Figure 4 summarize the results of a performance evaluation for several red-black tree implementations.

These measurements were collected on a Linux machine running an Intel Xeon E3 processor at 3.10 GHz with 32 GB of RAM. Since different implementations were originally written in different languages, they were all ported to Racket [3] and run with Racket 8.3 CS. Every sample ran the entire sequence of operations 5 times and 100 such samples were collected for each configuration.

Each implementation was tested with inputs in ascending order and in a random order. The line plots show the execution time across several tree sizes, while the box plots show the execution time for 2^{20} elements. Lower is better.

Monadic insertion is about 1.16× faster than Okasaki's [7]'s original when inserting 2^{20} elements in a random order. When the input sequence is in ascending order, this improvement increases to about 1.57× faster.

Monadic deletion performs the same, or a tad better, than the best existing algorithm of Filliâtre and Letouzey [2]. On a randomly distributed deletion sequence, their performance exactly coincides. The monadic algorithm is significantly faster than the approaches of Kahrs [6] and Germane and Might [4].

6 Related Work

Okasaki [7] gave a beautiful account of insertion, but omitted any discussion of deletion. Deletion is more difficult because black-height invariance is a global

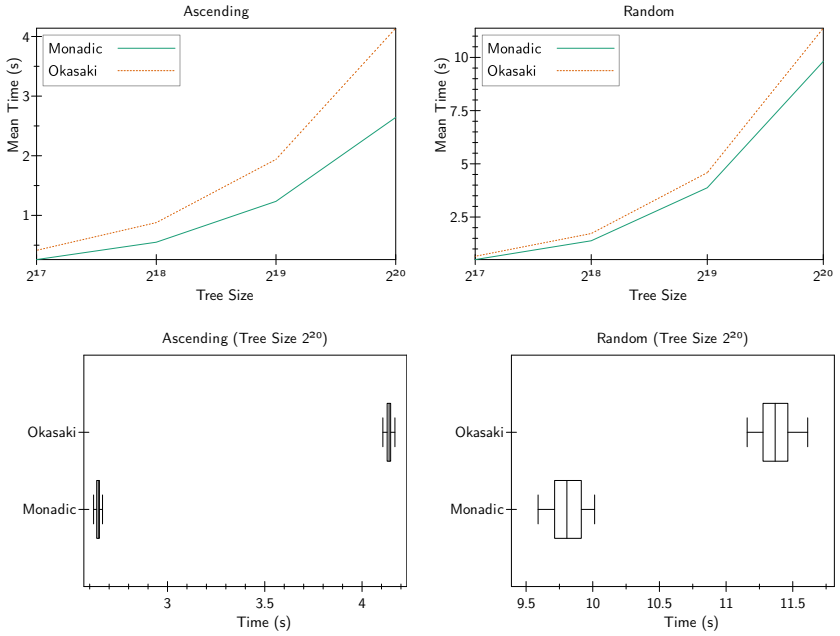


Fig. 3. Performance of red-black tree insertion (lower is better).

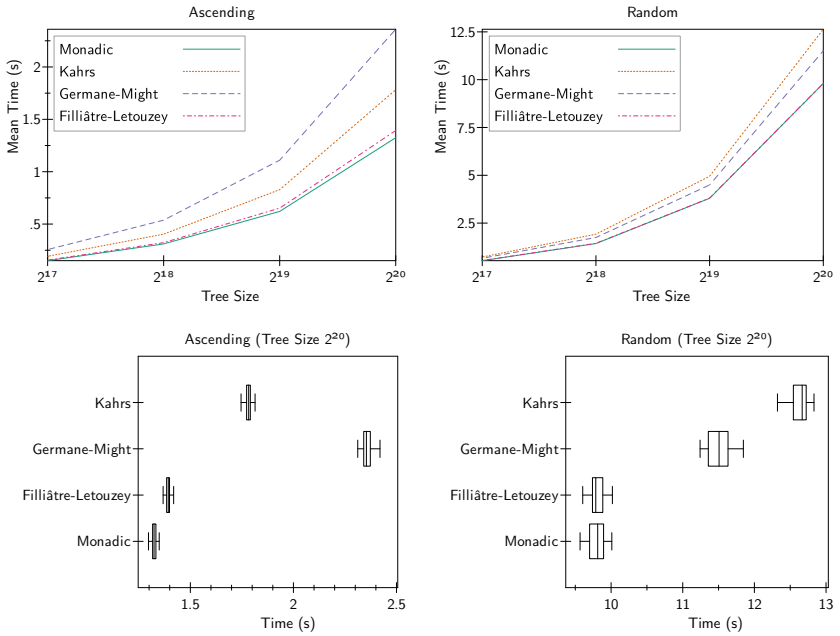


Fig. 4. Performance of red-black tree deletion (lower is better).

property; whether a subtree violates the black-height invariant can be determined only through inspection of the entire tree. To avoid this, a subtree must somehow indicate that its black height is too small—that it's black-height deficient. Every paper on red-black trees does this differently.

Filliâtre and Letouzey [2] develop an implementation where black-height deficiency is handled in an ad-hoc way using a threaded Boolean. Germane and Might [4] use a "double-black" color to serve the same function. Kahrs [6] describes a significantly different approach that maintains an additional invariant during the deletion process: black nodes are always deficient and red nodes are never deficient. Thus, the information is communicated implicitly instead of explicitly.

Germane and Might report that their double-black algorithm has poor performance—substantially worse than the one given by Kahrs. However, their evaluation was fatally flawed; it measured a version of the double-black algorithm with a suboptimal order of conditional branches. Reordering these branches greatly improves performance. Figure 4 uses a variant of Germane and Might's code without this pathology.

7 Conclusion

Given the beauty of red-black tree insertion, the absence of a deletion algorithm that is simultaneously efficient and simple has been unfortunate. Using the `Result` monad to indicate black-height deficiency yields an implementation that is fast, and remains easy to understand. The same monadic style can be applied to insertion as well. This yields a faster algorithm, without compromising on elegance.

Acknowledgements. Thanks to Matthias Felleisen for his feedback and encouragement. Also, thanks to Ben Lerner, Jason Hemann, Leif Andersen, Michael Ballantyne, Mitch Gamburg, and Sam Caldwell, for providing valuable comments that significantly improved the exposition.

References

1. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. MIT Press (2009)
2. Filliâtre, J.C., Letouzey, P.: Functors for proofs and programs. In: European Symposium on Programming (ESOP) (2004). https://doi.org/10.1007/978-3-540-24725-8_26
3. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Design Inc. (2010), https://racket-lang.org/tr1/
4. Germane, K., Might, M.: Deletion: The curse of the red-black tree. Journal of Functional Programming (JFP) (2014). https://doi.org/10.1017/S0956796814000227
5. Guibas, L., Sedgewick, R.: A dichromatic framework for balanced trees. In: IEEE Symposium on Foundations of Computer Science (1978). https://doi.org/10.1109/SFCS.1978.3

6. Kahrs, S.: Red-black trees with types. Journal of Functional Programming (JFP) (2001). https://doi.org/10.1017/S0956796801004026
7. Okasaki, C.: Red-black trees in a functional setting. Journal of Functional Programming (JFP) (1999). https://doi.org/10.1017/S0956796899003494