Faster, Simpler Red-Black Trees

Cameron Moy

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

Abstract. For more than two decades, functional programmers have refined the persistent red-black tree—a data structure of unrivaled elegance. This paper presents another step in its evolution. Using a monad to communicate balancing information yields a fast insertion procedure, without sacrificing clarity. Employing the same monad plus a new decomposition simplifies the deletion procedure, without sacrificing efficiency.

Keywords: Algorithms · Data Structures · Trees

1 A Quick Recap

A red-black tree is a *self-balancing* binary search tree [3, 8]. Insertion and deletion operations rebalance the tree so it never becomes too lopsided. To this end, every node carries an extra bit that "colors" it either red or black. In Haskell [12]:

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

For convenience, nodes of each color can be constructed and matched using the pattern synonym extension of the Glasgow Haskell Compiler:

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

Insertion and deletion use chromatic information to maintain two invariants:

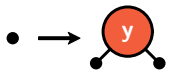
1. The **red-child invariant** states that a red node may not have a red child.
2. The **black-height invariant** states that the number of black nodes along all paths through the tree—the *black height*—is the same.

These two properties imply that the tree is roughly balanced. Naively inserting or deleting nodes from the tree may violate these invariants. Hence, the challenge of implementing red-black trees is to repair the invariants during a modification.

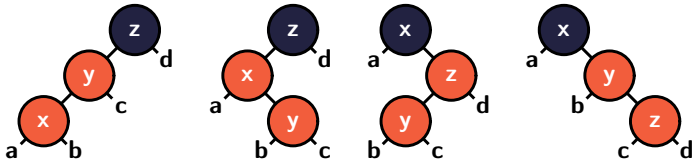
This paper improves on existing work with four contributions: (1) a faster way to implement insertion by avoiding redundant pattern matching; (2) a simpler way to implement deletion by employing two new auxiliary operations; (3) a monad instance that communicates information across recursive calls; (4) an evaluation that compares the performance of several red-black tree implementations. The algorithms are presented in Haskell since it provides a convenient notation for monads, but the approach is not language specific. Appendix A provides a Racket version.

2 Insertion à la Okasaki

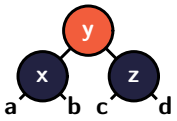
Recall the insertion algorithm of Okasaki [11]. For an ordinary binary search tree, insertion traverses the tree and replaces a leaf with the desired value. For a red-black tree, insertion's first step is the same, with the new node colored red:[1]



Doing so does not introduce a black-height violation but it may introduce a red-child violation if the leaf's parent happens to be red. A `balance` function resolves such red-child violations. A violation can only come in one of four shapes



and `balance` fixes the violation by transforming each into:



Realizing this diagram as code is a straightforward, if tedious, exercise:

```
-- Pre: Child may have a red-child violation.
-- Post: Satisfies the red-child invariant;
--       BH(out) = BH(in).[2]
balance :: Tree a -> Tree a
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 a black node into a red node, this may induce a red-child violation one level up the tree. Thus, `insert` must `balance` at every level. This process "bubbles" violations up the tree. At the end, `insert` blackens the root to resolve the last possible violation:

[1] The diagrams use the letters **x**, **y**, **z** for values; the letters **a**, **b**, **c**, **d** for subtrees; and • for the empty tree.

[2] Where BH computes the black height of a tree.

```
insert :: Ord a => a -> Tree a -> Tree a
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 :: Tree a -> Tree a
blacken (N _ a y b) = B a y b
blacken s = s
```

3 Insertion, Faster

The `balance` operation is applied at every level of a tree during insertion. Each time, `balance` pattern matches four specific shapes. Often, however, this pattern matching is unnecessary.

Suppose `balance` returns a black node. No more red-child violations can occur further up the tree, since the rest of the tree satisfies the red-child invariant. In other words, when `balance` produces a black node, the "bubbling" stops. No more work needs to be done and every subsequent `balance` is redundant.

For a mutable data structure, a `break` statement could eliminate the extra work. For an immutable data structure, a different solution is needed. An additional data type[3] makes short circuiting future operations possible:

```
type Result a = Result' a a
data Result' a b = D a | T b
```

A `Result` contains a tree where either the work is *done*, constructed with `D`, or there is more *to do*, constructed with `T`. Trees marked with `D` do not violate the red-child invariant, while trees marked with `T` may. Trees marked with `D` can pass forward unaffected, while trees marked with `T` must be fixed by calling `balance`.

A `Monad` instance for `Result` makes this use case easy to express. A tree where more work needs to be done is given to a function `f`, while a tree that is done propagates:

```
instance Monad (Result' a) where
  return x = T x
  (D x) >>= f = D x
  (T x) >>= f = f x
```

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

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

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

and `<$$>` applies a function to the contents of both `T` and `D` values[4]

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

Equipped with `Result`, suspended calls to `balance` further up a tree can be bypassed by wrapping a subtree in `D`. As mentioned before, it is safe to do so whenever `balance` produces a black node. Here is the new `balance` function:

```
balance :: Tree a -> Result (Tree a)
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)
```

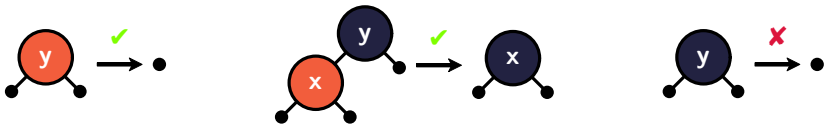
Now that `balance` returns a `Result` value, `insert` must handle it. The essence of the function, however, remains the same:

```
insert :: Ord a => a -> Tree a -> Tree a
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
```

Using this approach, insertion can be up to 1.56× faster than the original one.

4 Deletion, Simpler

As with `insert`, the `delete` function is similar to ordinary deletion on a binary search tree. For an internal node, `delete` replaces the target node with its in-order successor. Only the base cases, where no in-order successor exists, are interesting. The following diagram shows all three:



Deleting a red node does not introduce a black-height violation, but deleting a black node might if its left child is empty; an empty left child provides no opportunity to maintain the black height. In other words, the subtree becomes *short* with respect to black height.

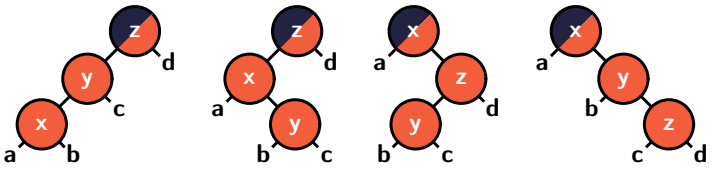
[4] Note that `<$$>` is not `fmap`. The functor instance implied by the monad applies a function only to `T` values.

Two auxiliary functions are needed to repair short subtrees: `balance'` and `eq`. Like `balance`, the `balance'` function resolves red-child violations. Unlike `balance`, it simultaneously increases a tree's black height. The `eq` function takes a tree where one child is short and equalizes the children's black heights. Both auxiliary functions use `Result` to communicate shortness information. Here, `Result` has a slightly different interpretation than `Result` for `insert`.

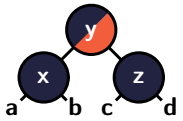
Recall that `insert` always adds a red node, *possibly* causing a red-child violation. Subtrees are wrapped in `T` if there *might* be a violation and `D` if there is not. In contrast, the final base case for `delete` *always* causes a black-height violation. Thus, a subtree is wrapped in `T` if it is *definitely* short and `D` otherwise.

4.1 `balance'`

The purpose of `balance'` is to resolve red-child violations and, if possible, increase the black height by one. To accomplish this, the function acts like `balance`, except the root color is preserved. So the following four shapes[5]



are transformed into



Whether the root is black or red, all red-child violations are resolved and the black height is increased by one. If none of the four shapes match, then the tree is blackened. Differences compared to `balance` are highlighted:

```
-- Pre: Root or child may have a red-child violation.
-- Post: Satisfies the red-child invariant;
--       BH(out) = BH(in) + 1 or BH(out) = BH(in).
balance' :: Tree a -> Result (Tree a)
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

blacken' :: Tree a -> Result (Tree a)
blacken' (R a y b) = D (B a y b)
blacken' s = T s
```

[5] The half-colored nodes indicate that the color could be either red or black.

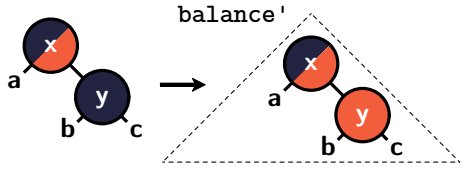
Three facts are worth noting. First, not only can `balance'` resolve trees with two red nodes in a row, but also trees where there are three red nodes in a row. Second, `balance'` never induces a red-child violation further up the tree because it never turns a black node into a red node. Finally, when provided a red node, `balance'` always returns a `D` result.

4.2 `eq`

Just as `insert` needs `balance`, `delete` needs a function that can repair the black-height invariant at every level of the tree. That is the purpose of `eq`. Although it is possible to define a single function to get the job done, it is convenient to split the function in two: `eqL` and `eqR`.[6]

Given a short left (right) child, `eqL` (`eqR`) returns a tree where the black heights of the children are equal. If the function can raise the black height of the left (right) child, it does so. If it cannot, it lowers the black height of the sibling and bubbles the violation up.

Consider `eqL`, where the left child, labeled **a**, is short. There are two cases to consider: when its sibling is black and when its sibling is red. Here is the first case, where the sibling is black and the root is any color:[7]



To equalize the black heights, `eqL` reduces the black height of the right child by reddening it. Now the whole tree is short. Not only that, but this can introduce red-child violations. If **b** is red, there may even be three red nodes in a row. The `balance'` function is designed to handle all of these issues simultaneously:

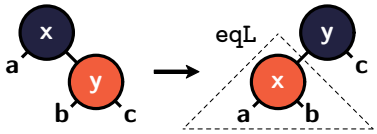
```
-- Pre: BH(left) = BH(right) - 1.
-- Post: BH(left) = BH(right).
eqL :: Tree a -> Result (Tree a)
eqL (N k a x (B b y c)) = balance' (N k a x (R b y c))

-- Pre: BH(right) = BH(left) - 1.
-- Post: BH(right) = BH(left).
eqR :: Tree a -> Result (Tree a)
eqR (N k (B a x b) y c) = balance' (N k (R a x b) y c)
```

Next, consider the case where the sibling is red. Here, `eqL` applies a rotation that does not affect any black heights and calls itself recursively on the left child:

[6] Some split `balance` into `balanceL` and `balanceR` [10, exercise 3.10]. For `balance`, splitting is done for performance rather than convenience.

[7] The dotted triangle encloses the tree that `balance'` is applied to.



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

```
eqL (N k a x (R b y c)) = (\a -> B a y c) <$$> eqL (R a x b)
eqR (N k (R a x b) y c) = (\b -> B a x b) <$$> eqR (R b y c)
```

4.3 Putting it Together

Here is the rest of the code, which composes the presented functions into a complete algorithm:

```
delete :: Ord a => a -> Tree a -> Tree a
delete x s = (fromResult . del) s
  where del E = D E
        del (N k a y b)
          | x <  y = eqL =<< (\a -> N k a y b) <$$> del a
          | x == y = delCur (N k a y b)
          | x >  y = eqR =<< (\b -> N k a y b) <$$> del b

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

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

5 Performance Evaluation

Using monads to communicate balancing information yields a unified and elegant presentation of both insertion and deletion; critically though, these variants perform as well as or better than existing algorithms. The next two pages summarize a performance evaluation for several functional red-black tree algorithms. Figure 1 and Table 1 present the data for insertion. Figure 2 and Table 2 present the data for deletion.

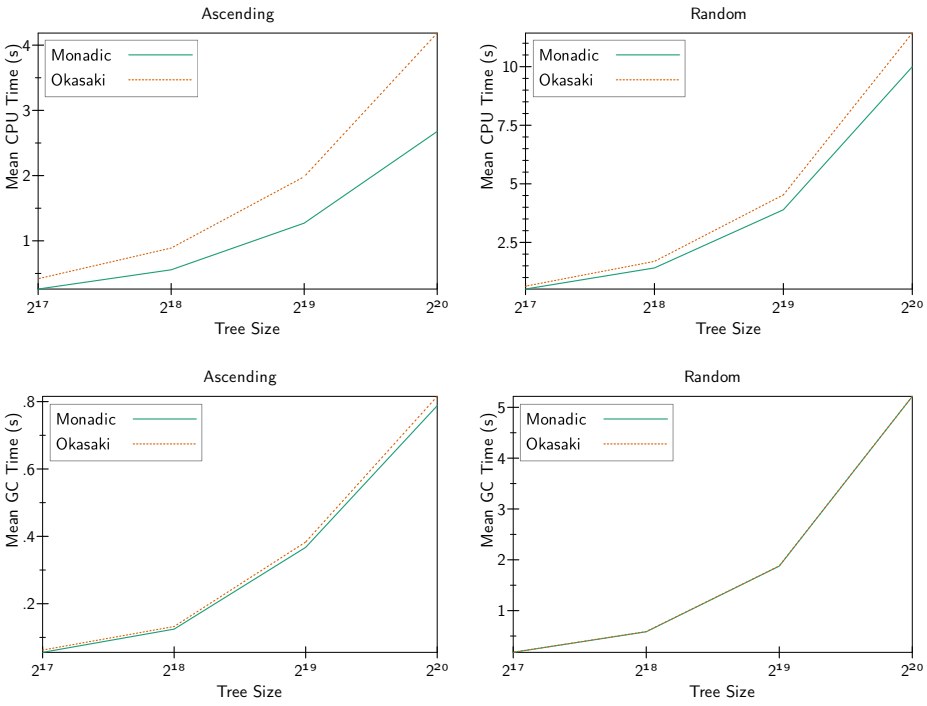


Fig. 1: Insertion Line Plots

Benchmark	Algorithm	CPU (s)	GC (s)	Memory (MB)
ASCENDING (2^{20})	Monadic	2.68 ± 0.01	0.79 ± 0.01	7807 ± 1
	Okasaki	4.19 ± 0.01	0.82 ± 0.01	7810 ± 1
RANDOM (2^{20})	Monadic	10 ± 0.07	5.21 ± 0.03	5619 ± 7
	Okasaki	11.43 ± 0.1	5.22 ± 0.05	5622 ± 8
SUFFIXTREE	Monadic	4.99 ± 0.05	0.33 ± 0.01	2720 ± 1
	Okasaki	5.35 ± 0.03	0.33 ± 0.01	2720 ± 1

Table 1: Insertion Measurements (Mean ± Standard Deviation)

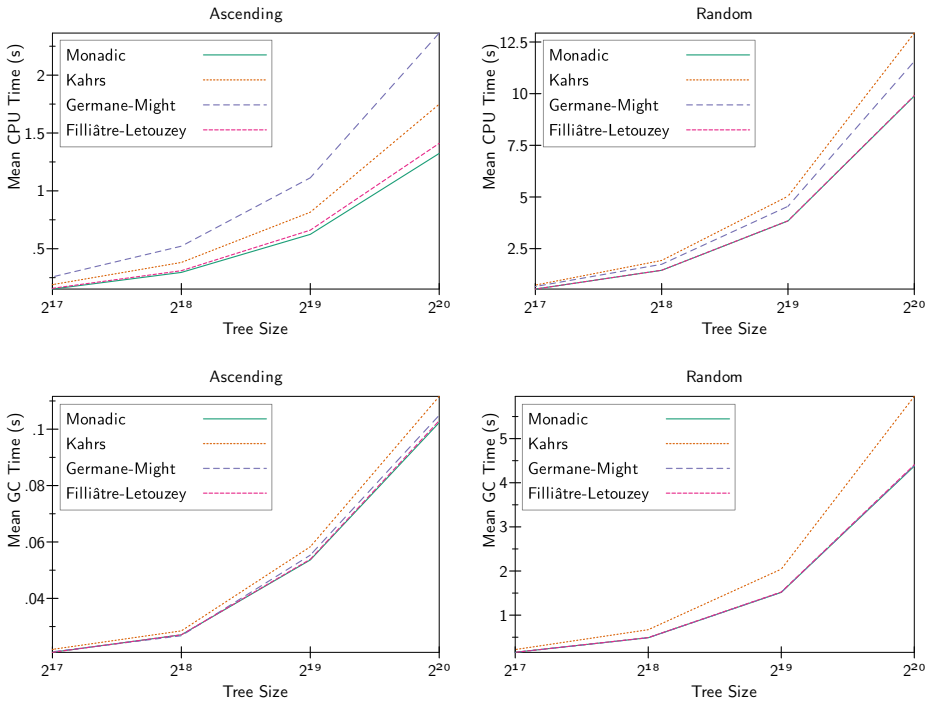


Fig. 2: Deletion Line Plots

Benchmark	Algorithm	CPU (s)	GC (s)	Memory (MB)
ASCENDING (2^{20})	Monadic	1.32 ± 0.01	0.1 ± 0	4685 ± 34
	Kahrs	1.75 ± 0.02	0.11 ± 0	7324 ± 58
	Germane-Might	2.36 ± 0.03	0.1 ± 0	5377 ± 64
	Filliâtre-Letouzey	1.41 ± 0.01	0.1 ± 0	4918 ± 33
RANDOM (2^{20})	Monadic	9.88 ± 0.11	4.38 ± 0.03	5376 ± 15
	Kahrs	12.93 ± 0.13	5.96 ± 0.03	8635 ± 32
	Germane-Might	11.56 ± 0.13	4.39 ± 0.04	5301 ± 11
	Filliâtre-Letouzey	9.9 ± 0.1	4.41 ± 0.02	5328 ± 11

Table 2: Deletion Measurements (Mean ± Standard Deviation)

These measurements were collected on a Linux machine running an Intel Xeon E3 processor at 3.10 GHz with 32 GB of RAM. Since the different algorithms were originally implemented in different languages, they were all ported to Racket [5] and run with Racket 8.7 CS. Racket is a strict language, so the performance characteristics should generalize better than that of a lazy language like Haskell. Every sample ran the entire sequence of operations 5 times and 100 such samples were collected for each configuration.

A configuration consists of a specific choice for input size, input order, and algorithm. For a given size n, the input values are the first n natural numbers, ordered in two different ways: ascending and random. The random order is a random permutation of the input data.

To test insertion and deletion, each red-black tree algorithm was used to implement sets. Insertion was tested by adding all the input values to an empty set. Deletion was tested by removing all the input values from a set containing them already.

As an additional benchmark for insertion, the SUFFIXTREE program from the gradual typing benchmark suite [7] was adapted to use red-black trees instead of hash tables. This program uses Ukkonen's algorithm to calculate the suffix tree of a text—in this case T.S. Eliot's "Prufrock."

The line plots in Figure 1 and Figure 2 show mean execution time, both total execution time including garbage collection (GC) and just GC time, across several tree sizes. Note that the plots are log scale. Table 1 and Table 2 give the mean and standard deviation of total execution time, GC time, and memory consumption for the same benchmarks.

Monadic insertion is about 1.14× faster than Okasaki's original [11] when inserting 2^{20} elements in a random order. When the input sequence is in ascending order, this improvement increases to about 1.56× faster. On the SUFFIXTREE benchmark it is 1.07× faster, demonstrating that the optimization has a measurable impact on the end-to-end performance of a real-world program.

Monadic deletion performs the same, or a tad better, than the algorithm of Filliâtre and Letouzey [4], currently the best known approach. On a randomly distributed deletion sequence, their performance exactly coincides. The monadic approach is significantly faster than that of Kahrs [9] and Germane and Might [6]. This evaluation demonstrates that the simplicity of the monadic deletion algorithm does not come at the cost of performance.

6 Related Work

Okasaki [11] gave a beautiful account of insertion, but omitted any discussion of deletion. Deletion is more difficult than insertion because black-height invariance is a global 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 is short. Every paper on red-black trees does this differently.

Filliâtre and Letouzey [4] develop an implementation where shortness is handled in an ad-hoc way using a threaded Boolean. Germane and Might [6] use a "double-black" color to serve the same function. The `Result` monad has the same purpose, but eliminates the manual bookkeeping necessary in both of them. Kahrs [9] describes a significantly different approach; it maintains an additional invariant during the deletion process: black nodes are always short and red nodes are never short. Thus, the information is communicated implicitly rather than explicitly.

The deletion algorithm presented here is substantially simpler to understand than prior work for two reasons. First, all prior algorithms have three cases for `eq` instead of just two. By factoring out `balance'`, two special cases collapse into one. Second, all prior algorithms require contortions to deal with the red sibling case. Specifically, each uses a three-level pattern match combined with a nested `balance` operation. The `eq` function presented here uses a two-level pattern match and recursion instead.

Germane and Might report that their double-black algorithm has poor performance—substantially worse than the one given by Kahrs. However, their evaluation is fatally flawed; it measures a version of the double-black algorithm with a suboptimal order of conditional branches. Reordering these branches improves performance. Section 5 evaluates a corrected variant of Germane and Might's code. See Appendix B for an explanation of this modification.

A related line of work focuses on proving the correctness of red-black tree algorithms using proof assistants [1, 4] and GADTs [9, 13]. These techniques should easily be applicable to this paper, and doing so is left as an exercise to the reader.

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 yields an algorithm that, along with `balance'` and `eq`, achieves both goals. The same monadic style can be applied to insertion, yielding a faster algorithm, without compromising simplicity.

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

Much of the code in this paper was directly adapted or at least heavily influenced by the code of Okasaki [11] (for insertion) and Germane and Might [6] (for deletion). They deserve a great deal of credit for the final product.

This work was partially supported by NSF grant SHF 2116372.

Bibliography

[1] Appel, A.: Efficient verified red-black trees. https://www.cs.princeton.edu/~appel/papers/redblack.pdf (2011)

[2] Ashley, J.M., Dybvig, R.K.: An efficient implementation of multiple return values in scheme. LISP and Functional Programming (LFP) (1994). https://doi.org/10.1145/182590.156784

[3] Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. MIT Press (2009)

[4] 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

[5] Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Design Inc. (2010), https://racket-lang.org/tr1/

[6] Germane, K., Might, M.: Deletion: The curse of the red-black tree. Journal of Functional Programming (JFP) (2014). https://doi.org/10.1017/S0956796814000227

[7] Greenman, B., Takikawa, A., New, M.S., Feltey, D., Findler, R.B., Vitek, J., Felleisen, M.: How to evaluate the performance of gradual typing systems. Journal of Functional Programming (JFP) (2019). https://doi.org/10.1017/S0956796818000217

[8] 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

[9] Kahrs, S.: Red-black trees with types. Journal of Functional Programming (JFP) (2001). https://doi.org/10.1017/S0956796801004026

[10] Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1999)

[11] Okasaki, C.: Red-black trees in a functional setting. Journal of Functional Programming (JFP) (1999). https://doi.org/10.1017/S0956796899003494

[12] Peyton Jones, S.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003)

[13] Weirich, S.: Red black trees (redux). https://www.seas.upenn.edu/~cis5520/21fa/lectures/stub/06-GADTs/RedBlackGADT0.html (2021)

A Racket Implementation

This section shows a Racket port of the Haskell code. Monads can be implemented in many ways, but the following code uses macros and multiple return values [2] to do so. This choice yields excellent performance.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; insert

(define (insert t x)
  (define (ins t)
    (match t
      [(E) (todo (R (E) x (E)))]
      [(N k a y b)
       (cond
         [(< x y) (=<< balance (<$$> (λ (a) (N k a y b)) (ins a)))]
         [(> x y) (=<< balance (<$$> (λ (b) (N k a y b)) (ins b)))]
         [else    (done t)])]))
  (blacken (from-result (ins t))))

(define (balance t)
  (match t
    [(or (B (R a x (R b y c)) z d)
         (B (R (R a x b) y c) z d)
         (B a x (R (R b y c) z d))
         (B a x (R b y (R c z d))))
     (todo (R (B a x b) y (B c z d)))]
    [(B _ _ _) (done t)]
    [_         (todo t)]))

(define (blacken t)
  (match t
    [(R a x b) (B a x b)]
    [_         t]))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; delete

(define (delete t x)
  (define (del t)
    (match-define (N k a y b) t)
    (cond
      [(< x y) (=<< del-left  (<$$> (λ (a) (N k a y b)) (del a)))]
      [(> x y) (=<< del-right (<$$> (λ (b) (N k a y b)) (del b)))]
      [else    (del-root t)]))
  (from-result (del t)))
```

```
(define (del-root t)
  (match t
    [(B a y (E)) (blacken* a)]
    [(R a y (E)) (done a)]
    [(N k a y b)
     (define m (box false))
     (=<< del-right (<$$> (λ (b) (N k a (unbox m) b)) (del-min b m)))]))

(define (del-min t m)
  (match t
    [(B (E) y b) (set-box! m y) (blacken* b)]
    [(R (E) y b) (set-box! m y) (done b)]
    [(N k a y b)
     (=<< del-left (<$$> (λ (a) (N k a y b)) (del-min a m)))]))

(define (del-left t)
  (match t
    [(N k a y (R c z d))
     (<$$> (λ (a) (B a z d)) (del-left (R a y c)))]
    [(N k a y (B c z d))
     (balance* (N k a y (R c z d)))]))

(define (del-right t)
  (match t
    [(N k (R a x b) y c)
     (<$$> (λ (b) (B a x b)) (del-right (R b y c)))]
    [(N k (B a x b) y c)
     (balance* (N k (R a x b) y c))]))

(define (balance* t)
  (match t
    [(or (N k (R a x (R b y c)) z d)
         (N k (R (R a x b) y c) z d)
         (N k a x (R (R b y c) z d))
         (N k a x (R b y (R c z d))))
     (done (N k (B a x b) y (B c z d)))]
    [_ (blacken* t)]))

(define (blacken* t)
  (match t
    [(R a x b) (done (B a x b))]
    [_         (todo t)]))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; monad

(define-syntax-rule (todo x)
  (values true x))

(define-syntax-rule (done x)
  (values false x))

(define-syntax-rule (from-result x)
  (let-values ([(_ y) x])
    y))

(define-syntax-rule (<$$> f x)
  (let-values ([(a d) x])
    (values a (f d))))

(define-syntax-rule (=<< f x)
  (let-values ([(ax dx) x])
    (if ax (f dx) (values ax dx))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; data

(struct E ())
(struct N (color left value right))

(define-syntax-rule (define-color name)
  (begin
    (define-for-syntax (transf stx)
      (syntax-case stx ()
        [(_ a x b) #'(N 'name a x b)]))
    (define-match-expander name transf transf)))

(define-color R)
(define-color B)
```

B Performance Evaluation Correction

Germane and Might [6] incorrectly conclude that their algorithm is always significantly slower than other approaches. This conclusion is due to a subtle confounding factor that put their algorithm at an unfair disadvantage.

To understand the flaw, consider this skeleton of their `delete` function:

```
delete :: Ord a => a -> Tree a -> Tree a
delete x s = del (redden s)
  where del E = E
        del (R E y E)
          | x == y = ...
          | x /= y = ...
        del (B E y E)
          | x == y = ...
          | x /= y = ...
        del (B (R E y E) z E)
          | x < z  = ...
          | x == z = ...
          | x > z  = ...
        del (N k a y b)
          | x < y  = ...
          | x == y = ...
          | x > y  = ...
```

It highlights the two most common cases during deletion, when the current node does not match the target and the function recurs on either the left or right side. However, the structure of the code forces each of the base cases to be checked first—before the most common cases.

To favor the common cases, the skeleton should look as follows:

```
delete :: Ord a => a -> Tree a -> Tree a
delete x s = del (redden s)
  where del E = E
        del (N k a y b)
          | x < y  = ...
          | x == y =
              case s of
                R E y E -> ...
                B E y E -> ...
                B (R E y E) z E -> ...
          | x > y  = ...
```

The base cases are only checked at the target node. This simple modification improves the performance of the double-black algorithm by 2×.