JFP **24** (4): 423–433, 2014. © Cambridge University Press 2014
doi:10.1017/S0956796814000227

FUNCTIONAL PEARL
Deletion: The curse of the red-black tree

KIMBALL GERMANE and MATTHEW MIGHT
University of Utah, UT, USA
(*e-mail:* `krgermane@gmail.com`)

Abstract

Okasaki introduced the canonical formulation of functional red-black trees when he gave a concise, elegant method of persistent element insertion. Persistent element deletion, on the other hand, has not enjoyed the same treatment. For this reason, many functional implementations simply omit persistent deletion. Those that include deletion typically take one of two approaches. The more-common approach is a superficial translation of the standard imperative algorithm. The resulting algorithm has functional airs but remains clumsy and verbose, characteristic of its imperative heritage. (Indeed, even the term *insertion* is a holdover from imperative origins, but is now established in functional contexts. Accordingly, we use the term *deletion* which has the same connotation.) The less-common approach leverages the features of advanced type systems, which obscures the essence of the algorithm. Nevertheless, foreign-language implementors reference such implementations and, apparently unable to tease apart the algorithm and its type specification, transliterate the entirety unnecessarily. Our goal is to provide for persistent deletion what Okasaki did for insertion: a succinct, comprehensible method that will liberate implementors. We conceptually simplify deletion by temporarily introducing a "double-black" color into Okasaki's tree type. This third color, with its natural interpretation, significantly simplifies the preservation of invariants during deletion.

1 Introduction

Red-black trees are an efficient representation of ordered sets, and many common operations, such as search and insertion, are possible in logarithmic time. Their efficiency stems from their mostly-balanced nature, which is guaranteed by their structural invariants.

A red-black tree is a binary tree in which each node is colored red or black, and whose construction satisfies two properties:

1. the local property that every red node has two black children, and
2. the global property that every path from the root to a leaf[1] node contains the same number of black nodes.

[1] For our purposes, leaf nodes do not house a value and are colored black.

These conditions guarantee that the longest path from root to leaf can be no more than twice the shortest (the only difference being individual red nodes interspersed along the way), so the penalty of locating an element in a red-black tree is minor relative to that in a perfectly-balanced tree.

Okasaki properly introduced red-black trees into the functional world when he gave a concise, elegant method of element insertion (Okasaki, 1999). In Okasaki's formulation, insertion of an element begins by traversing the tree, in typical recursive fashion, to find the location on the fringe to place it (or find that insertion unnecessary, in the case that the element is encountered). The newly-added node is colored red, an assignment which may introduce a local property violation. (This act characterizes Okasaki's algorithm: the preservation of the global property at the expense of the local one.) To account for the local property violation, the tree is re-*balanced* as the traversal recedes. This operation rearranges trees of one of the forms

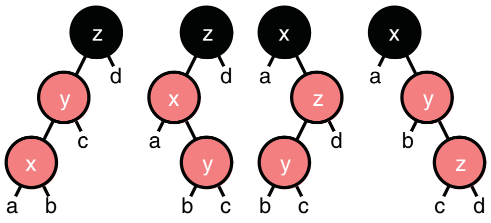


Fig. 1. (Colour online)

to obtain

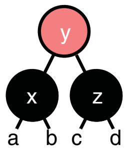


Fig. 2. (Colour online)

The final step of Okasaki's insertion algorithm is to blacken the root of the tree, which may resolve a red-red violation outside the scope of *balance*.

The concision and elegance of insertion can be manifest in code as well. We use the Racket language (Flatt & PLT, 2010) for this purpose, but any modern functional programming language will do.[2]

First, we define a datatype to represent red-black trees with

(**struct** *RBT* {})

Its variants include internal nodes, defined by

(**struct** **N** *RBT* {*color left-child value right-child*})

and leaf nodes, defined by

(**struct** **L** *RBT* {})

[2] We also provide a Haskell (Peyton Jones, 2003) implementation in the appendix.

We define the macro (**R** *a x b*) to match and construct **r**ed nodes and the macros (**B** *a x b*) and (**B**) to match and construct **b**lack internal nodes and leaves. We also define (**R?** *a*) and (**B?** *a*) which match nodes without deconstructing them, and bind each to *a*.

With this syntax in hand, Okasaki's insertion algorithm is indeed concise.

```
(define (insert t v)
  (define-match ins
    [(B) (R (B) v (B))]
    [(N c a x b)
     (switch-compare
       (v x)
       [< (balance (N c (ins a) x b))]
       [= (N c a x b)]
       [> (balance (N c a x (ins b)))])])
  (blacken (ins t)))
```

```
(define-match balance
  [(or (B (R (R a x b) y c) z d)
       (B (R a x (R 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))))
   (R (B a x b) y (B c z d))]
  [t t])

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

2 Deletion

Deletion is dual to insertion: where insertion is found unnecessary at the presence of an element, deletion is found unnecessary at its absence. The deletion of any element from the empty tree yields, of course, the empty tree.

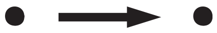


Fig. 3

However, insertion into binary trees has the advantage that a new node is added only to the fringe, whereas deletion might also target an interior node. We accommodate this complication by replacing the value in a targeted interior node with its inorder successor[3], if it exists, and deleting its fringe node. If its inorder successor doesn't exist, we can replace the targeted interior node with its left subtree. This substree may be empty, to be handled by the following example on the left, or may be a red singleton, to be handled by the same example on the right.

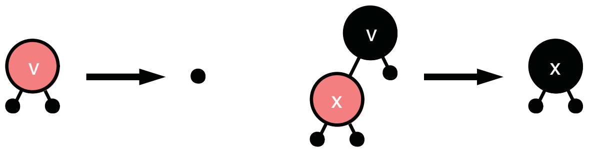
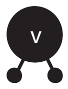


Fig. 4. (Colour online)

In order to preserve the global property, the root color of the resulting tree must be a combination of the root colors of the original tree and its left child that preserves

[3] That is, the value of the leftmost node of the interior node's right child.

their respective path contributions. For pre-transformation root colors of red and black, a post-transformation black root satisfies this requirement. So constrained, we are naturally led to wonder how the tree root of



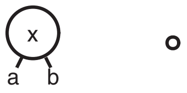
should be colored under deletion of v.

In response, we introduce a transitory color, *double-black*, to temporarily preserve the global invariant which both nodes and leaves can take on. We permit double-black leaves by another variant of the red-black tree datatype with

(**struct L2** *RBT* {})

and double-black nodes by another color ’BB. As before, we also define macros to match and construct double-black (**BB**) nodes and leaves. The -*B* function demotes a double-black node or leaf to its black counterpart and will prove useful.

Double-black nodes and leaves, depicted with “reversed polarity” as



contribute two black nodes to any path through them. With a color with this property made available, it becomes obvious how to handle a singleton black node:

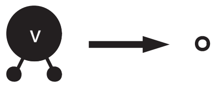
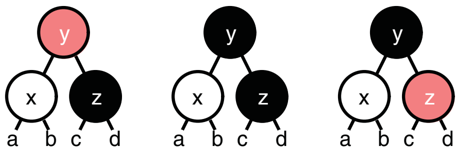


Fig. 5

Of course, a tree which includes double-black nodes is not a red-black tree, but just as Okasaki balanced red-red violations away by pulling them higher in the tree, we can *rotate* double-black nodes (and leaves) away in the same fashion. We attempt to discharge a double-black node, if one exists, at each step of the traversal unwinding. If discharge isn’t possible at a particular step, we arrange for it to be considered by the next step by pulling it higher in the tree. In this way, the double-black node bubbles upward until it is extinguished. At each of these steps, we encounter one of only three tree arrangements (and their reflections) which require double-black node treatment.



In the first case, we can discharge the double-black node immediately with the rotation

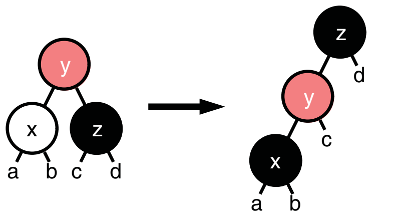


Fig. 6. (Colour online)

but, if the subtree *c* is red-rooted, this rotation introduces a red-red violation. We compose *balance* with the rotation to rectify the violation if it occurs. We match this case and its reflection within the *rotate* function by

[(**R** (**BB?** *a-x-b*) *y* (**B** *c z d*))
 (*balance* (**B** (**R** (*-B a-x-b*) *y c*) *z d*))]
[(**R** (**B** *a x b*) *y* (**BB?** *c-z-d*))
 (*balance* (**B** *a x* (**R** *b y* (*-B c-z-d*))))]

It's simple to verify that this rotation preserves the global property: count the number of black nodes in each path through the original subtree and see that the rotation preserves each.

In the next case, a similar rotation

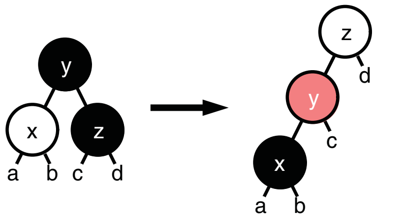


Fig. 7. (Colour online)

does not discharge the double-black node. This case is susceptible to the same red-red violation as the previous, but, rooted by a double-black node, cannot be handled by *balance*. We solve this simply by extending *balance* to include the transformations

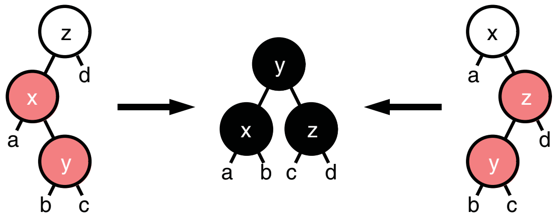


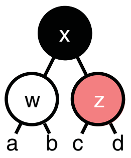
Fig. 8. (Colour online)

i.e., by adding

[(**or** (**BB** (**R** *a x* (**R** *b y c*)) *z d*)
 (**BB** *a x* (**R** (**R** *b y c*) *z d*)))
 (**B** (**B** *a x b*) *y* (**B** *c z d*))]

to its case analysis. Unlike the original cases of its design, these cases of *balance*, introducing no red nodes themselves, cannot introduce red-red violations.

It is hopeless to attempt to rearrange the final case



to satisfy the global and local properties simultaneously (not to mention ordering), even with the help of *balance* and *rotate*. However, we observe that in order for each path through this tree to have the same number of black nodes, the red node must have black nodes as children. Including the inner child in our consideration gives us just enough to satisfy every property.

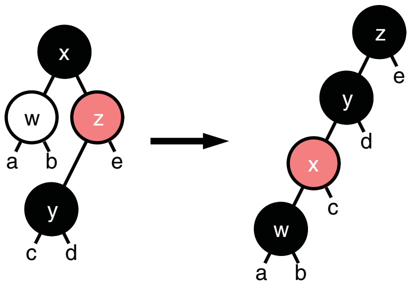


Fig. 9. (Colour online)

The possibility of a red-red violation is present here, but occurs deeper in the tree, and so cannot readily be handled after the rotation. Instead, we must integrate *balance* into the process. As cases for the *rotate* function, this is expressed by

[(**B** (**BB?** *a*-*w*-*b*) *x* (**R** (**B** *c* *y* *d*) *z* *e*))
 (**B** (*balance* (**B** (**R** (-*B* *a*-*w*-*b*) *x* *c*) *y* *d*)) *z* *e*)]
[(**B** (**R** *a* *w* (**B** *b* *x* *c*)) *y* (**BB?** *d*-*z*-*e*))
 (**B** *a* *w* (*balance* (**B** *b* *x* (**R** *c* *y* (-*B* *d*-*z*-*e*)))))]

At the final step of unwinding, a double-black node might reach the root of the tree, outside the domain of *rotate*. Such an occurrence would not be fatal since the newly-colored root node could be soundly demoted by unilaterally *blacken*ing it after deletion, just as Okasaki does for insertion. This would expose *blacken* to the transient double-black color. In the interest of containing double-black to *balance* and *rotate*, we adopt a coloring policy which prevents a double-black node from ever reaching the root.

Observe that, in any non-empty red-black tree, either one of the root's children is red or the root can be colored red without violating any invariant. So colored, this configuration guarantees a bubbling double-black node will be discharged before it reaches the root, if only just before. We codify this strategy by prefixing deletion with a *redden* operation, which provides that a tree's root may be *blacken*ed after an insertion only to be *redden*ed before a deletion. To avoid such unnecessary operations, we admit red roots in red-black trees—our invariants in fact never excluded them—and weaken *blacken* to do so only if the red-black construction demands it. This reveals another view of the duality of insertion and deletion: just as the final step of insertion is to *blacken* the root if necessary, the initial step of deletion is to *redden* it if possible.

Deleting the inorder successor from the right child can be accomplished by invoking *delete* directly, but this requires many unnecessary comparisons and that the value of the successor be retrieved beforehand. We define *min/delete* to extract and delete a tree's minimum element efficiently.

```
(define-match min/delete
  [(B) (error 'min/delete "empty tree")]
  [(R (B) x (B)) (values x (B))]
  [(B (B) x (B)) (values x (BB))]
  [(B (B) x (R a y b)) (values x (B a y b))]
  [(N c a x b) (let-values ([(v a*) (min/delete a)])
                 (values v (rotate (N c a* x b))))])
```

With these definitions, the deletion algorithm can be expressed succinctly.

```
(define-match balance
  ; Figures 1 and 2
  [(or (B (R (R a x b) y c) z d)
       (B (R a x (R 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))))
   (R (B a x b) y (B c z d))]
  ; Figure 8
  [(or (BB (R a x (R b y c)) z d)
       (BB a x (R (R b y c) z d)))
   (B (B a x b) y (B c z d))]
  [t t])

(define-match rotate
  ; first case, Figure 6
  [(R (BB? a-x-b) y (B c z d))
   (balance (B (R (-B a-x-b) y c) z d))]
  [(R (B a x b) y (BB? c-z-d))
   (balance (B a x (R b y (-B c-z-d))))]
  ; second case, Figure 7
  [(B (BB? a-x-b) y (B c z d))
   (balance (BB (R (-B a-x-b) y c) z d))]
  [(B (B a x b) y (BB? c-z-d))
   (balance (BB a x (R b y (-B c-z-d))))]
  ; third case, Figure 9
  [(B (BB? a-w-b) x (R (B c y d) z e))
   (B (balance (B (R (-B a-w-b) x c) y d)) z e)]
  [(B (R a w (B b x c)) y (BB? d-z-e))
   (B a w (balance (B b x (R c y (-B d-z-e)))))]
  ; fall through
  [t t])
```

```
(define-match -B
  [(BB) (B)]
  [(BB a x b) (B a x b)])

(define (delete t v)
  (define-match del
    ; Figure 3
    [(B) (B)]
    ; Figure 4
    [(R (B) (== v) (B))
     (B)]
    ; Figure 4
    [(B (R a x b) (== v) (B))
     (B a x b)]
    ; Figure 5
    [(B (B) (== v) (B))
     (BB)]
    [(N c a x b)
     (switch-compare
      (v x)
      [< (rotate (N c (del a v) x b))]
      [= (let-values ([(v* b*) (min/delete b)])
           (rotate (N c a v* b*)))]
      [> (rotate (N c a x (del b v)))])])
  (del (redden t)))

(define-match redden
  [(B (B? a) x (B? b))
   (R a x b)]
  [t t])
```

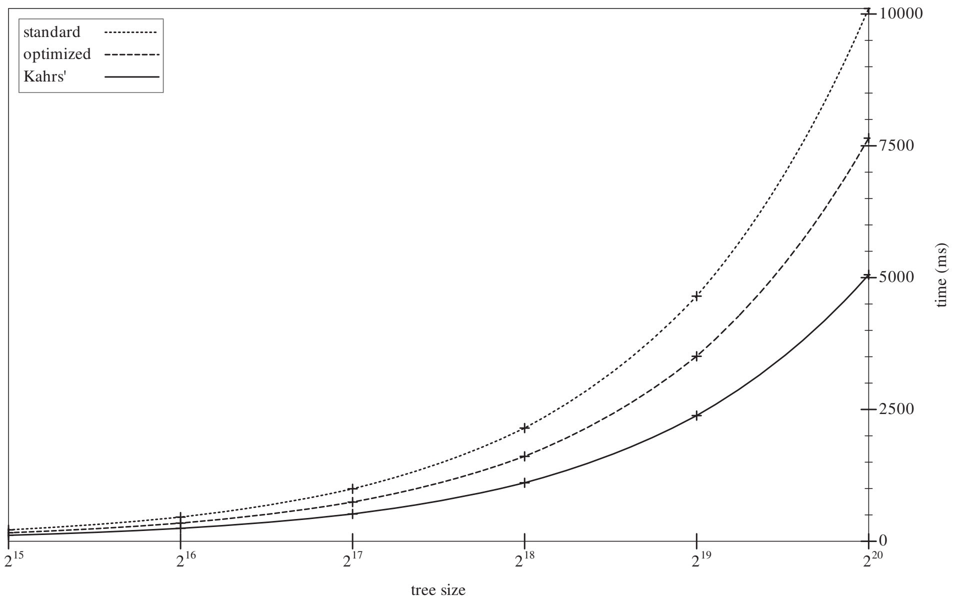


Fig. 10. Time benchmark comparison.

3 Evaluation

The primary goal of the preceding account of persistent deletion is comprehensibility. Once the algorithm is understood, many opportunities become apparent to make the algorithm more efficient (but, we contend, less clear). For instance, the *balance* cases can be partitioned by the root node color—black or double-black–and in which child the red-red violation may occur—left or right.

We evaluated the implementation, both as presented and so-"optimized", against the so-called untyped formulation of Kahrs (Kahrs, 2001).[4] In order to make a more direct comparison, we translated Kahrs' formulation into Racket. While originally implemented in Haskell, his untyped formulation, as one would expect, does not leverage Haskell's type system in any appreciable way, so we are confident our port to Racket is faithful.

We performed a benchmark consisting of a sequence of cascading deletions from trees increasing exponentially in size, repeated five times for each size.

Figure 10 illustrates the execution time of each implementation as a graph of execution time over log-scaled tree size. Kahrs' algorithm performed twice as fast as the presented implementation and still significantly faster than the optimized version. This advantage may be due to his extraction method: instead of replacing the target node's value with that of its inorder successor, it stitches the subtrees of the target node together.

Each implementation uses essentially the same representation for red-black trees, so memory usage for a given tree in each is the same. An instrumented garbage

[4] We intended to include a zipper-based implementation in the evaluation, but the Scheme port as found was incorrect.

collector revealed revealed differences in the behavior of each implementation with regard to it. For each benchmark, the presented and optimized implementations allocated in total 26G and 25G, and 22% and 32% of execution time was under control of the garbage collector, respectively. In contrast, Kahrs' implementation allocated 34G in total, and 32% of execution time was under control of the garbage collector.

4 Related work

The standard imperative algorithm can be found in many textbooks, such as Cormen *et al.* (Cormen *et al.*, 2001). Translations of this algorithm have been found in SML/NJ (Appel & MacQueen, 1991) and some Scheme implementations (Dybvig, 2003). Despite the use of Huet's zippers (Huet, 1997) and Hinze's linear-time construction method (Hinze *et al.*, 1999), these translations retain the imperative character of their ancestry.

Static type systems have been used to great effect to encode the red-black tree invariants. Kahrs (Kahrs, 2001) leverages Haskell's type system to promote correctness and Appel (Appel, 2011) outright proves Kahrs' approach correct using Coq (Bertot & Castéran, 2004), as Filliâtre does with his own implementation (Filliâtre & Letouzey, 2004). Kahrs' algorithm has been transliterated into languages with ill-suited type systems such as Scala's collections library (Odersky, 2009) and several Scheme implementations.

5 Conclusion

In spite of the existence of faster methods of persistent deletion, the numerous transliterated and even incorrect implementations underscore the need for a comprehensible account. We have endeavored to give such an account. Once the method is understood, clarity can be spent for efficiency, and the performance of the resulting algorithm is competitive with other methods.

Acknowledgments

This material is partially based on research sponsored by DARPA under agreement number FA8750-12-2-0106 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

Appel, A. W. (2011) Efficient verified red-black trees.

Appel, A. & MacQueen, D. (1991) Standard ml of new jersey. In *Programming Language Implementation and Logic Programming*. Springer, pp. 1–13. Springer Berlin Heidelberg.

Bertot, Y. & Castéran, P. (2004) *Interactive Theorem Proving and Program Development: Coq'art: the Calculus of Inductive Constructions.* Springer. Springer-Verlag Berlin Heidelberg.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2001) *Introduction to Algorithms*, vol. 2. The MIT press Cambridge. The MIT Press Cambridge.

Dybvig, R. K. (2003) *The Scheme Programming Language*. The MIT Press. The MIT Press Cambridge.

Filliâtre, J.-C. & Letouzey, P. (2004) Functors for proofs and programs. In *Programming languages and systems*. pp. 370–384.

Flatt, M. & PLT. (2010) *Reference: Racket*. Tech. rept. http://racket-lang.org/tr1/.

Hinze, R. *et al.* (1999) Constructing red-black trees. In *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages*, vol. 99, pp. 89–99.

Huet, G. (1997) The zipper. *J. Funct. Program.* **7**(05), 549–554.

Kahrs, S. (2001) Red-black trees with types. *J. Funct. Program.* **11**(04), 425–432.

Odersky, M. (2009) The scala language specification, version 2.8. *Epfl lausanne, switzerland.*

Okasaki, C. (1999) Red-black trees in a functional setting. *J. Funct. Program.* **9**(04), 471–477.

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

Appendix A. Haskell code

```
data Color = R | B | BB deriving (Show)

data Tree elt = E | EE | T Color (Tree elt) elt (Tree elt) deriving (Show)

type Set a = Tree a

empty :: Set elt
empty = E

insert :: Ord elt => elt -> Set elt -> Set elt
insert x s = blacken (ins s)
  where ins E = T R E x E
        ins (T color a y b) | x <  y = balance color (ins a) y b
                            | x == y = T color a y b
                            | x >  y = balance color a y (ins b)

        blacken (T R (T R a x b) y c) = T B (T R a x b) y c
        blacken (T R a x (T R b y c)) = T B a x (T R b y c)
        blacken t = t

balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
balance BB a x (T R (T R b y c) z d)  = T B (T B a x b) y (T B c z d)
balance BB (T R a x (T R b y c)) z d = T B (T B a x b) y (T B c z d)
balance color a x b = T color a x b

delete :: Ord elt => elt -> Set elt -> Set elt
delete x s = del (redden s)
  where del E = E
        del (T R E y E) | x == y = E
                        | x /= y = T R E y E
        del (T B E y E) | x == y = EE
                        | x /= y = T B E y E
        del (T B (T R E y E) z E) | x <  z = T B (del (T R E y E)) z E
                                  | x == z = T B E y E
                                  | x >  z = T B (T R E y E) z E
        del (T c a y b) | x <  y = rotate c (del a) y b
                        | x == y = let (y',b') = min_del b
                                   in rotate c a y' b'
                        | x >  y = rotate c a y (del b)

        redden (T B (T B a x b) y (T B c z d)) =
          T R (T B a x b) y (T B c z d)
        redden t = t

rotate R (T BB a x b) y (T B c z d) = balance B (T R (T B a x b) y c) z d
rotate R EE y (T B c z d) = balance B (T R E y c) z d
rotate R (T B a x b) y (T BB c z d) = balance B a x (T R b y (T B c z d))
rotate R (T B a x b) y EE = balance B a x (T R b y E)
rotate B (T BB a x b) y (T B c z d) = balance BB (T R (T B a x b) y c) z d
rotate B EE y (T B c z d) = balance BB (T R E y c) z d
rotate B (T B a x b) y (T BB c z d) = balance BB a x (T R b y (T B c z d))
rotate B (T B a x b) y EE = balance BB a x (T R b y E)
rotate B (T BB a w b) x (T R (T B c y d) z e) =
  T B (balance B (T R (T B a w b) x c) y d) z e
rotate B EE x (T R (T B c y d) z e) = T B (balance B (T R E x c) y d) z e
rotate B (T R a w (T B b x c)) y (T BB d z e) =
  T B a w (balance B b x (T R c y (T B d z e)))
rotate B (T R a w (T B b x c)) y EE = T B a w (balance B b x (T R c y E))
rotate color a x b = T color a x b

min_del (T R E x E) = (x,E)
min_del (T B E x E) = (x,EE)
min_del (T B E x (T R E y E)) = (x,T B E y E)
min_del (T c a x b) = let (x',a') = min_del a
                      in (x',rotate c a' x b)
```