Draft:
Exorcising the curse : Deletion in Red-black tree

Pierre-Emmanuel Wulfman

November 18, 2023

Abstract

Chris Okasaki showed how to implement red-black trees in a functional programming language [4], but his implementation missed an important function : the deletion of a node from the red-black tree. Late work by Matt Might proposed an algorithm for deletion which required adding two colors to our red-black tree [1], thus changing our data structure and adapting all algorithm. Other works by Ralf Hinze and Stefan Khars [3], showed how to implement such tree with deletion by combining phantom types, existential types, and nested datatypes. In this paper, we will show how to implement the deletion algorithm for the standard red-black tree in a functional setting, using only simple datatypes without adding extra colors. By removing these extra requirements, this algorithm can be implemented in language that doesn't support such advanced type system and can be easily adapted to other languages. This algorithm is inspired by Okasaki's insertion algorithm and is proved correct using the Coq proof assistant.

1 Introduction

Red-black trees are a kind of self-balancing binary search trees (BST), with additional properties over simple BST that makes it impossible for them to degenerate into combs and thus guaranties the performance of look-ups, insertions and deletions in the worst case.

In a red-black tree, each node has an additional color bit (red or black), and the relative balance of the tree is achieved by enforcing the following properties :

1. Each node is either red or black.
2. The root is black
3. All leaves are black
4. A Red node cannot have red child
5. Every path from a given node to any of its leaves goes through the same number of black nodes. (this number is referred as the black height of the subtree at the node)

The last two, that we will refer as the red property and the black property, imply that at every node the higher subtree emerging from its children is at most twice as high as the other subtree. Thus, it ensures that there is at worst a factor two between the deepest leaf and the other leaf, keeping it sufficiently balance to guaranty the algorithmic performance (log (n)).

While the first three properties are easily enforced in the definition of the datatype. The standard BFS insertion and deletion algorithm break both black and red property. In order to preserve those invariant, the equivalent algorithms for red-black trees need to add extra step of rotation and recoloring. The imperative version of the algorithm was first presented by Guibas and Sedgewick in 1978 [2] and the purely functional version by Chris Okasaki in 1999 [4].

Oksaki adapted the insertion like this :

```
let insert cmp elt tree =
    let rec ins = function
        L -> T (Red, L, elt, L)
      | T (color, left, root, right) ->
         let diff = cmp elt root in
         if diff = 0 then T (color, left, root, right)
         else if diff < 0 then
            balance color (ins left) root right
         else
            balance color left root (ins right)
    in blacken (ins tree)
```

As in BST, the new node is inserted at the leaf. The inserted node is always red, which preserve the black invariant but breaks the red one if inserted as a chield of a red noe. For this reasons, Okasaki added a balance operation along the insertion path to restore the red porperty. This balance operation uses the rotation depicted in fig 1 to take care of a red child followed by a red grand-child. (Notice that this case appear only along the insertion path so not on both child at the same time). This rotation put a red node at the top of the subtree which was previously black by construction. This recoloring can also lead to a double red node, which is solved by recurrence, applying the same rotation while rewinding the insertion path until there is no more double red nodes or we eventualy reach the root. Sometime this will lead the root to be turned red. In this case, we simply repaint it black to restore the second property.

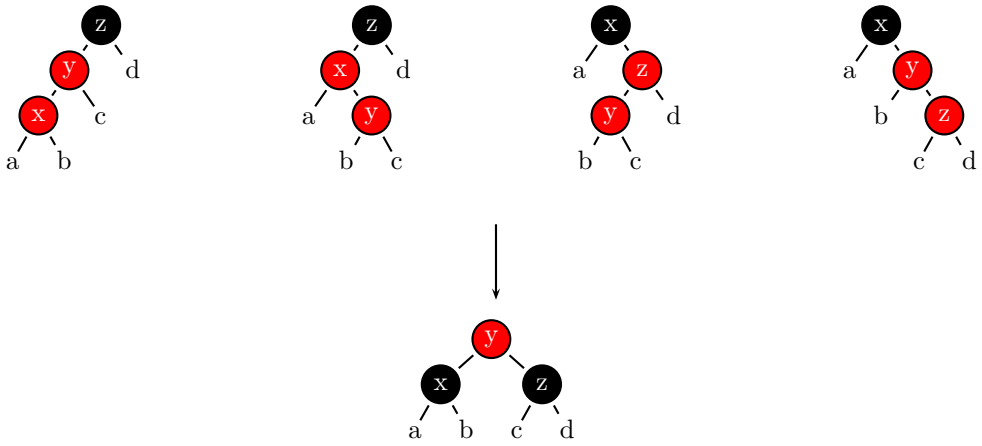

Figure 1: Okasaki's balance algorithm

Okasaki's balance algorithm is presented here :

```
let balance color left root right =
    match color, left, root, right with
        Black, T (Red, T (Red, a, x, b), y, c), z, d
      | Black, T (Red, a, x, T (Red, b, y, c)), z, d
      | Black, a, x, T (Red, T (Red, b, y, c), z, d
      | Black, a, x, T (Red, b, y, T (Red, c, z, d)) ->
         T (Red, T (Black, a, x, b), y, T (Black, c, z, d))
      | _ ->
         T (color, left, root, right)
```

To sum up, the insertion algorithm break the red property and Okasaki proposed the balance algorithm that restore it.

Similarly, the deletion algorithm may lead to the deletion of a black node which break the black property. But Okasaki's paper did not provide a functional deletion algorithm.

In 2104, Might proposed a deletion algorithm that never break the black invariant. He achieved this by adding a third color, a double black color, which count as two black nodes for the count of the black height. With this new color, deleting a black node trigger an "increase" of the color of the root node, hence the black

height stay the same, but it breaks the first property of a red-black tree, which is later restore using different balancing technique. While correct, this solution has drawbacks, coined "a curse" in the article's title, that is two added color and one added leaf node, a modification of Okasaki's balance algorithms, and an arguably complex deletion algorithm.

In this article we will present an algorithm that restore the black invariant during deletion without the addition of extra colors, thus breaking the "curse".

2 Deletion

The deletion for a BST algorithm is the following :

```
let delete cmp elt tree =
    let rec del = function
        L -> raise Not_found
      | T (left, root ,right) ->
        let c = cmp elt root in
        if c < 0 then (del left) root right
        else if c > 0 then left root (del right)
        else remove left right
    and remove l r = match l,r with
        left, L  -> left
      | L, right -> right
      | T(l,v,r), right ->
        let v', l' = remove_rightmost l v r in
        T (l', v', right)
    and remove_rightmost left value = function
        L -> value, left
      | T(l,v,r) -> T (left,value, remove_rightmost l v r)
```

We took a similar approach than Okasaki's by first adapting the BST standard deletion by adding a call to a balance function that will restore the black property if it is broken. To do so, this balance function needs to be provided with the information on "how" the property was broken (i.e. if one subtree is shorter and which one) In the case of Okasaki's insertion, this information is present local from the color of the node, its children and grand-children. In the case of deletion, we can't get this information on the reduction of the black height just by looking at the top of the subtree. We could compute the black eight of both subtrees, but this would degrade the performance of the algorithm. Instead, this information is dispatched through the structure of the algorithm itself. The information that we need is whether the deletion appeared in the left or right subtree, that we get from the chain of calls and if the deletion reduce the black height (by 1) or not, and we will get this information from our remove function

This lead to the following `delete` function :

```
let delete cmp elt tree =
    let rec del = function
        L -> raise Not_found
      | T (color, left, root ,right) ->
        let c = cmp elt root in
        if c < 0 then balance_left color (del left) root right
        else if c > 0 then balance_right color left root (del right)
        else remove color left right
     ...
```

The `remove` function also need to be modified, to take the color as input and to return, along with the new subtree, a boolean set to true if the tree is shorter after deletion. For this function, we discriminate between three cases on the node to remove.

1. The node has two leaf children, in this case, it is replaced by a leaf (fig. 2), which decrease the black height if the node was black.

2. The node has exactly one leaf child, then by construction, the other child is a red node with two leaf children, and the node to remove is black. In this case, we replace the node to remove by its child, repainted black (fig. 3) and the black height doesn't change.

3. In the last case the node has two non leaf children. We replace the value with the in-order predecessor (or successor) and then remove this other node, which has at most one child as the rightmost (resp. leftmost) value of a tree, so we end up with one of the previous cases.

Since the deletion in the left of right subtree can reduce the black height and thus break the black property, we have to add a balance function in the algorithm to restore it.

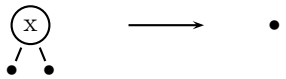

Figure 2: Deletion of a leaf

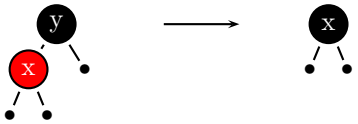

Figure 3: Deletion of a node with only one child

This algorithm is shown here:

```
and remove c l r = match c,l,r with
  (* Remove a leaf *)
  | Red   , L, L -> L, false
  | Black, L, L -> L, true
  (* Only one child implies the child is red and the parent is black *)
  | Black, T (Red, l, v, r), L)
  | Black, L, T (Red, l, v, r)) -> T (Black, l, v, r), false
  (* Two sub-trees*)
  | c, T(c',l',v',r'), r ->
    let v, l = remove_rightmost c' l' v' r' in
         balance_left c l v r

and remove_rightmost c l v = function
    L -> v, remove c l L
  | T (c', l', v', r') ->
    let rightmost_value, r = remove_rightmost c' l' v' r'
    rightmost_value, balance_right c l v r
```

Let's now take a look at the `balance_left` function. This function receives a node after the delete function was called on the left subtree. If the left subtree has the same black height as before, the `balance_left` function simply reconstitute the node. If the left subtree is shorter, then we need to rotate the tree so that the black property is restored at this node. Let looks at the cases that we might have when the subtree is shorter. Let assume that the left child is red, then we can repaint it black, and the problem is solved. In fact, the remove function should have done this already and return the child repainted black and false instead. Thus, we will assume that the left child is black. (property a) Also, because the black height of a tree is at least 1, and was reduced by one, the black height of the subtree at the right child, left's sibling, is at least 2, by construction. (property b)

The first case is when the sibling is red (fig. 4). Then the root is black (red property) and the sibling has two black children that are non-leaf (property b). If we name the left node (a), the first child's children (b) and (c) and the other child (d). We notice that (a), (b), and (c) have the same black height which is one less than

(d). Then with a left rotation at the sibling node, we put (a), (b), (c), and (d) at the same height in the tree. By coloring the sibling black and (d) red, we restore the black property, but we may break the red property. For this reason, we have to call Okasaki's balance function on the right sibling restoring the red property at the sibling subtree and keeping the black property intact.

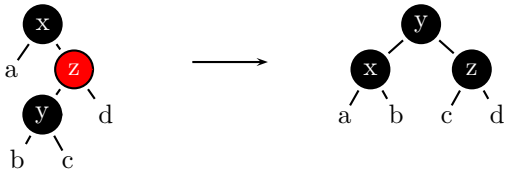

Figure 4: Case 1

The second case is when the sibling is black and has at least one red child (fig. 5). Still naming (a) the left node and (b), (c), (d) the red child's children and the other sibling's child, in this order. (a), (b), (c), and (d) have the same black height. Similarly, a left rotation at the right sibling (or a right rotation if the red child is at the right) and coloring the red child black fix the black property.

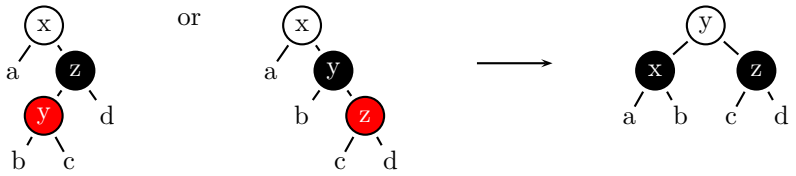

Figure 5: Case 2

The last case and the easiest one is when the sibling is black and has two black children (fig. 6). Because both children are black we can paint the right child red and restore the black property locally. This lead to the black height of the whole subtree to be reduced by one. Then, If the root is red, we paint it black, which increase the black eight and restore the black property globally. In the other case, we live the tree as is and propagate the information to the function caller. When unpiling the call stack, the balancing will be performed by recurrence until the black invariant is restored, or we eventually reach the root.

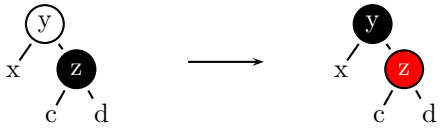

Figure 6: Case 3

The code of the `balance_left` function is presented here

```
let balance_left color (left, is_shorter) value right =
    if is_shorter then
        match color, left, value, right with
        (* fig 4 *)
        | Black, a, x, T (Red, T (Black, b, y, c), z, d)
          -> T (Black, T (Black, a, x, b), y, balance Black c z (redden d)),false
        (* fig. 5 *)
        | k, a, x, T (Black, T (Red, b, y, c), z, d)
        | k, a, x, T (Black, b, y, T (Red, c, z, d))
          -> T (k, T (Black, a, x, b), y, T (Black, c, z, d)),false
        (* fig 6*)
        | k, x, y, T (Black, c, z, d)
          -> T (Black, x, y, T (Red, c, z, d)),k=Black
        | _ -> failwith "Impossible cases by red property, or property b"
    else
        T (color, left, value, right), false
```

the full code of the delete function is given in appendix A.

The `balance_right` is the symmetric as `balance_left` with left and right siblings inverted.

3 Formal Proof

4 Conclusion

In essence, the algorithm presented here is the very similar as Might's deletion algorithm. In this previous algorithm, the author used extra colors which as the effect to transform the problem of a global property being broken into a local property being broken. Which allowed him to solve the problem similarly as in the insertion algorithm. We notice that the relevant effect of the added color was to store the information on the reduction of the black height in the tree, which can be done in a simpler way by using the call stack. This new implementation avoids the several drawbacks, and possible misinterpretation and understanding, of Might's algorithm. For these reasons, we believe that our implementation should be preferred.

References

[1] KIMBALL GERMANE and MATTHEW MIGHT. Deletion: The curse of the red-black tree. *Journal of Functional Programming*, 24(4):423–433, 2014.

[2] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21, 1978.

[3] STEFAN KAHRS. Red-black trees with types. *Journal of Functional Programming*, 11(4):425–432, 2001.

[4] CHRIS OKASAKI. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, 1999.

A Code of delete function

```
let balance_left color (left, is_shorter) value right =
    if is_shorter then
        match color, left, value, right with
        (* fig *)
        | Black, a, x, T (Red, T (Black, b, y, c), z, d)
          -> T (Black, T (Black, a, x, b), y, balance Black c z (redden d)),false
        (* fig.  *)
        | k, a, x, T (Black, T (Red, b, y, c), z, d)
        | k, a, x, T (Black, b, y, T (Red, c, z, d))
          -> T (k, T (Black, a, x, b), y, T (Black, c, z, d)),false
        (* fig *)
        | k, x, y, T (Black, c, z, d)
          -> T (Black, x, y, T (Red, c, z, d)),k=Black
        | _ -> failwith "Impossible cases by red property, or property b"
    else
        T (color, left, value, right), false
let balance_right color left value (right,is_shorter) =
   (* complemaentary as the above *)
   if is_shorter then
       match color, left, value, right with
        | Black, T (Red, a, x, T (Black, b, y, c)), z, d
          -> T (Black, balance Black (redden a) x b, y, T (Black, c, z, d)),false
        | k, T (Black, T (Red, a, x, b), y, c), z, d
        | k, T (Black, a, x, T (Red, b, y, c)), z, d
          -> T (k, T (Black, a, x, b), y, T (Black, c, z, d)),false
        | k, T (Black, a, x, b), y, z
          -> T (Black, T (Red, a, x, b), y, z),k=Black
        | _ -> failwith "Impossible cases by red property, or property b"
    else
        T (color, left, value, right), false

let delete cmp elt tree =
    let rec del = function
        L -> raise Not_found
      | T (color, left, root ,right) ->
        let c = cmp elt root in
        if c < 0 then balance_left color (del left) root right
        else if c > 0 then balance_right color left root (del right)
        else remove color left right
  and remove c l r = match c,l,r with
    (* Remove a leaf *)
    | Red  , L, L -> L, false
    | Black, L, L -> L, true
    (* Only one child implies the child is red and the parent is black *)
    | Black, T (Red, l, v, r), L)
    | Black, L, T (Red, l, v, r)) -> T (Black, l, v, r), false
    (* Two sub-trees*)
    | c, T(c',l',v',r'), r ->
      let v, l = remove_rightmost c' l' v' r' in
           balance_left c l v r
  and remove_rightmost c l v = function
      L -> v, remove c l L
    | T (c', l', v', r') ->
      let rightmost_value, r = remove_rightmost c' l' v' r'
      rightmost_value, balance_right c l v r
  (* Discard the boolean *)
  in fst (del tree)
```