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

Pierre-Emmanuel Wulfman

June 27, 2022

Abstract

Chris Okasaki showed how to implement red-black trees in a functional programing language, but his implementation missed an important function : the deletion of a node from the red-black tree. Matt Might then showed how we can perform such algorithm at the cost of adding two colors to our red-black tree. Other works by Ralf Hinze and Stefan Khars, showed how to implement such tree with deletion using higher-order datatypes. In this paper, we will show how to implement the deletion algorithm in a functional language, using only simple datatypes without adding extra colors.

1 Introduction

Red-black trees are a kind of self-balancing binary search trees, with additional properties over simple BST that makes it impossible for them to degenerate into comb and thus guarentie the performance of look-up, insertions and deletion in the worst case.

They ensure the balance of the tree in there structure by adding a color bit to each node and 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 fom a given node to any of its leaves goes through the same number of black nodes. (this number is refered 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 height of the higher subtree emerging from its child is at most twice the one emerging to its other child. Thus, it ensure 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 enforce in the definition of the datatypes. The standard BFS insertion and deletion algorithm break both black and red property. These algorithm has to be modified in order to restore such property. While this was done a while back for imperative data-structure, adapting these algorithms for persistent data-structure in a purely functional setting has been quite challenging. The insertion algorithm has been presented in [ref to Okasaki's] in 1999 and is now widely used.

Oksaki adapted the insertion like this :

```
let insert cmp elt tree =
    let rec ins = function
        L -> T (Red, L, elt, L)
      | T (colour, left, root, right) ->
         let diff = cmp elt root in
         if diff = 0 then T (colour, left, root, right)
         else if diff < 0 then
            balance colour (ins left) root right
         else
            balance colour 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 property but may break the red property which need to be restored. 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 cas 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 may also break the red property which will be restore again by the call to the balance function. Sometime this will lead to the root to be turn red. The algorithm finish by turning the root node black which does nothing if already black or increase the black height of the tree and restore the 2 property if it was red.

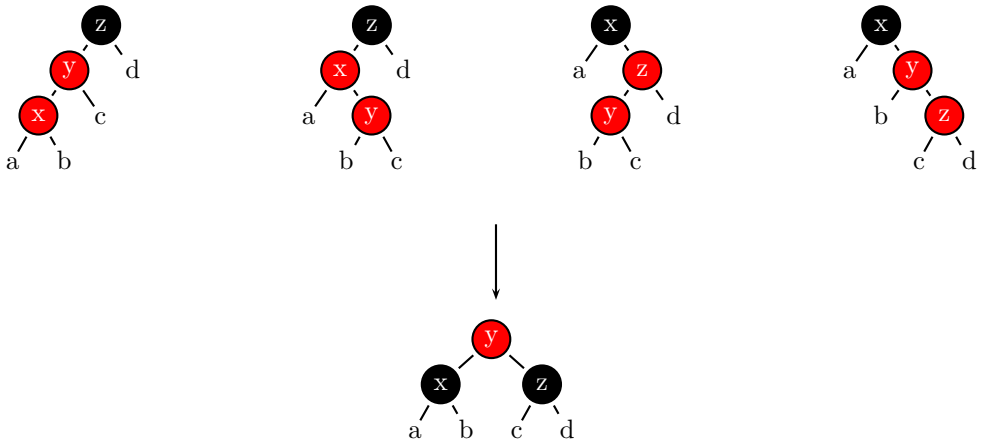

Figure 1: Okasaki's balance algorithm

Okasaki balance algorithm is presented here :

```
let balance colour left root right =
    match colour, 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 (colour, 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. In 2104 Might's find a way to avoid this issue by adding a third color, a double black color, which count as two black node for the count of the black height. With this new color, he is able to never break the black property during deletion, instead he break the first property which he later restore. While correct, this solutions has drawbacks, even called "a curse" in the article's title, that is two added color and added leaf node, a modification of Okasaki's balance algorithms, and a complex deletion algorithm.

In this article we will instead present an algorithm that restore the black property during deletion without using extra colours, then breaking the "curse" of an added color.

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 information on "how" the property was broken. In the case of Okasaki's insertion, this information is present in the color of the node, but in the case of deletion, we can't get the information on the reduction of the blackheight from the local coloring of the node. Instead, the information will be provide by the algorihtm itself. The information that we need is wither the deletion appeared in the left or right subtree, that we get from the chain of calls and if the deletion reduce the blackheight (by 1) or not, and we will get the information from our remove function Of course the balancing is symetric but different if the deletion is in the left or right subtree, so we split the balancing between `balance_right` and `balance_left`, to simplify the algorithm.

This lead to the following `delete` function :

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

The `remove` function also need to be modify, to take the colour as input and to return, along with the new subtree, a boolean set to true is the tree is shorter after deletion. For this function, we discriminate between three cases. Either the node has two leaf childs, in this case, it is replace by a leaf (fig. 2), which decrease the blackheight if the node was black. Either it has one exactly one leaf child, then by construction, the other child is a red node with two leaf childs, and the node to remove is black. In this cases, we replace the node to remove by its child, repainted black (fig. 3) and the length doesn't change. In the last case the node has two non leaf children. We replace the value with the inorder predecessor (or sucessor) and then remove this other node, which only has one child as the rightmost (resp. leftmost) value of a tree. Since we perform a deletion in the left of right subtree, we have to add a balance function after removal.

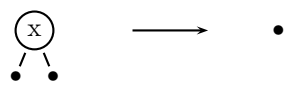

Figure 2: Deletion of a leaf

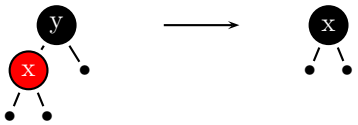

Figure 3: Deletion of a node with only one child

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 blackheight of a the tree is at least 1, and was reduce by one, the blackheight 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 colouring 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.

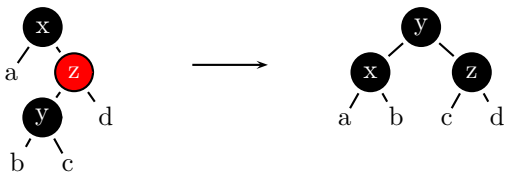

Figure 4: Case 1

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.

The last case which is 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 localy. Then if the root is red, we paint it black to restore the black property globaly, otherwise we are force to decrease the

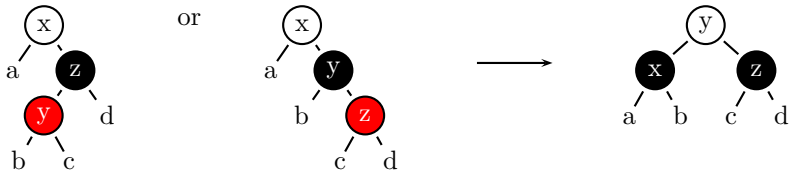

Figure 5: Case 2

size of the subtree and pass back this information. This algorithm will be call again on the root parent to try to restore the black property again. This may be push up to the root of the tree which will lead to the black height of the whole tree to be reduce by one, in the same way it is incease by one during Oksaki's insertion.

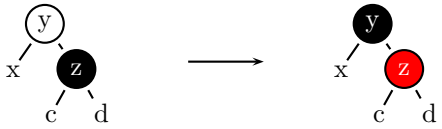

Figure 6: Case 3

The code of the `balance_left` function is presented here

```
let balance_left colour (left, is_shorter) value right =
    if is_shorter then
        match colour, 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 (colour, left, value, right), false
```

the full code of the delete function is given in appendix A. the `balance_right` is the same as `balance_left` with left and right siblings reversed.

3 Benchmark

4 Conclusion

In essence, the algorithm presented here is the same as Might's deletion algorithm. However, his choice to encode the information for rebalancing directly in the data-structure, as a new double-black color, leads to several drawback and possible misinterpretation and understanding, which are all avoided in our implementation. For this reason we believe that our implementation should be preferred.

A Code of delete function

```
let balance_left colour (left, is_shorter) value right =
    if is_shorter then
        match colour, 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 (colour, left, value, right), false
let balance_right colour left value (right,is_shorter) =
   (* complemaentary as the above *)
   if is_shorter then
       match colour, 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 (colour, left, value, right), false

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