Introduction to Red Black Trees

Helmut Brandl

Abstract

An introduction to red black trees.

Red Black Trees

An algorithm for insertion and deletion in a red black tree in a functional setting is described. The description provides evidence for the correctness of the algorithm.

A red black tree is not a trivial data structure. If you read the wikipedia article, you have the impression, that insertion and deletion are really very complicated, because there are many cases to distinguish. Especially deletion seems to be mind bending.

In this article we describe an algorithm which reduces the necessary case analysis to a minimum. We try to make the cases as orthogonal as possible, to make them comprehensible.

Furthermore we provide evidence for the correctness of each step.

> Link to an implementation of the described algorithm in Ocaml. **Implementation**

Basics

Definition

A red black tree is either an empty tree or a node with a color, a left child, an info element and a right child.

Each node has a color and the empty tree is considered black with a black height of zero.

A definition in a functional language like ocaml looks like

```
type info = ... (* type of the information element, must be sortable *)

type color = Red | Black

type t = Empty | Node of color * t * info * t
```

Insertion and deletion is always done at the bottom. We insert a node by replacing an empty tree with a singleton red node. We delete a node by replacing a singleton node (i.e. a node with two empty children) with an empty node.

Invariant

1. A red node has only black children (an empty tree counts as black).
2. Every path from the root to an empty node contains the same number of black nodes.
3. The inorder sequence is sorted i.e. a red black tree is a binary search tree.

Because the empty tree is considered black, a singleton red node does not violate the invariant. It has two empty black children.

Examples:

```
one red node:

        Rx

red node with children (only two black children possible):

        Ry
    Bx      Bz

black node with one or two children:

        By                  By                  By
    Rx                  Rx      Rz          Bx      Bz
```

If `h` is the black height of a red black tree, then the maximal height of the tree is `2 * h + 1`. E.g. a singleton red node has black height `0` but height `1`.

Insertion and deletion might create a violation of the invariant. Inserting a singleton red node below a red node might create a red violation because there are two red nodes in a row. Deleting a black node might create a black violation, because its sibling has a black height of one and the empty node has a black height of zero.

Insertion

Basics

A red black tree is sorted. If we want to insert an info element into a tree, we search for it following the order relation of the info element. There are two possiblilities.

1. We find a node with the info element we want to insert. There is nothing to be done.
2. Our search ends at an empty tree. This is the place to insert a new singleton red tree.

Let's look at the path from the insertion point to the root node.

```
Rnew    |   B R B B R ....

Rnew    |   R B R B B ...
```

In the first case we are ready, because no violation is created.

The second case is problematic, because we want to insert a red node below a red parent. We know that the grandparent must be black. We could swap the colors of the parent and the grandparent, creating a new red violation. This can bubble up until we reach two black nodes in a row.

However the solution is not that simple, because *stealing* the blackness of a parent might create two new other kind of violations. 1. The black parent might have a red child. 2. The black height of a path of the other branch to which the parent belongs reduces by one.

Algorithm

We can get a solution by satisfying the following insertion invariant: Inserting an element into a valid red black tree results in one of the states:

1. A nonempty tree where the root color has been changed from black to red and the new tree has the original black height.
2. A nonempty tree where the root color has not been changed and the new tree has the original black height.
3. Insertion into a red rooted tree ends with `a x b y c` where `a`, `b` and `c` are black rooted valid red black trees where the insertion has been successul in one of them and `x` and `y` are two info elements separating them. The black height of `a`, `b` and `c` is the black height of the original tree. Therefore we cannot form a valid red black tree of the original height without creating a red violation. The parent is black.

Note that all subtrees must be valid red black trees.

We have to prove that we can maintain the insertion invariant.

Initially we are in state 1. We insert the new element into an empty tree by creating a singleton red tree. Because the empty tree is considered black rooted, the root color has been changed from black to red. The new tree has the same black height as the initial tree (namely zero) and the new tree is a valid red black tree.

Now we have to consider inserting into a tree `Node(color, left, info, right)`. We only analyze the situation of inserting into the left child. Inserting into the right child is symmetrical. We assume that insertion into the left subtree ended in state 1, 2 or 3. We have to prove that insertion at the current level ends either in state 1, 2 or 3 as well.

1. Insertion into `left` created a new tree `Node(Red, a, x, b)` and `a` and `b` have the black height of the original tree `left`.

   - color = Black: We create `Node(Black, Node(Red,a,x,b), info, right)` and end in state 2.

   - color = Red: We cannot create a new valid red black tree. Therefore we end in state 3 with `a x b info right`. The next parent must be black, because the current node is red.

2. Insertion into `left` created a new tree `leftNew` whose color has not been changed. We remain in state 2 by returning `Node(color, leftNew, info,    right)`

3. Insertion into `left` ended up with `a x b y c`. The black height of `a`, `b` and `c` is the same as the black height of `left`. Because we are in state 3, the color of the current node must be black. We end in state 1 and return `Node (Red, Node (Black, a, x, b), y, Node (Black, c, info, right))`.

During insertion we have the following state diagramm.

```
               ++++++
               |    |
               v    |
1   ------->   2 ++++
^   \
|    \
|     ----->   3
|              |
\--------------/
```

If the insertion ends in state 1 or 2 at the root, then there is nothing to do. State 1 and 2 represent valid red black trees.

Ending in state 3 with `a x b y c` at the root we are allowed to introduce a new black level and generate either `Node (Black, a, x, Node (Red, b, y, c))` or `Node (Black, Node (Red, a, x, b), y, c)`.

Comparison to Chris Okasaki's Insertion Algorithm

In 1993 Chris Okasaki published an article named *Red-Black Trees in a Functional Setting* as a functional pearl in the journal of functional programming. The article described an insertion algorithm for red black trees in Haskell.

His algorithm is considered as the simplest possible insertion algorithm for red black trees in functional programming. I claim that the here presented insertion algorithm is even a little bit simpler than Chris Okasaki's and understandable without painting tree diagramms.

Furthermore it is more efficient, because

- It does in the rebalancing case only one case split and it does not need deep pattern match.
- It introduces the state 2 where nothing more has to be done until the root is reached.

Deletion

Deletion in a red black tree is more complicated than insertion. There are two difficulties to master:

- The info element to be deleted might be located in an interior node which has two non-empty children. We cannot just remove the node, because we cannot insert the two children into the parent.
- Deletion in one of the children of a node might reduce the black height of the child. Therefore the child with the deletion is no longer in balance with respect to black height with its sibling.

Fortunately we can separate the two issues and solve them without interference.

Deletion of an Interior Node

If the interior node has two non-empty children, we cannot deleted the node. However having two non-empty children, the leftmost element in the right child is a direct neighbor of the info element of the interior node in the order relation. We can delete the leftmost element and replace the info element of the interior node with the info element of the deleted leftmost node.

However the node with the leftmost info element in the order relation might not be the leftmost node in the right child. Let `Node(color, left, x, right)` be the interior node to be deleted (i.e. `x` is the info element to be removed), then we can have the following situation.

```
x
          right
         .
        .
       y
 empty   z
```

The leftmost info element in the order relation is `y`, but the node carrying the info element is not a singleton node. The solution is simple: We remove the bottom node carrying `z` and replace `y` with `z` and continue as if a node with info element `y` had been deleted.

This procedure leads to the following requirement:

A removal function which removes an info element of a red black tree must return an optional pair. The pair consists of a new tree where a node has been deleted and the info element which has been deleted.

For removal we need two functions:

- `remove_leftmost tree: optional (tree,deleted)`
- `remove element tree: optional (tree,deleted)`

We can call `remove_leftmost` on an empty tree. In that case the function returns nothing. I.e. we get an implicit test, if `right` is empty or not.

Deletion Invariant

Successful deletion of an element in a valid red black tree results in one of the following two states:

1. The new tree has the same black height as the original tree and its color remains the same or has been changed from red to black.
2. The new tree has a black height reduced by one. Its color is black and has not been changed.

Note that state 1 does not create any problems. We can insert the new tree into the parent node. It has the same black height, therefore both children still have the same black height. Its color might have changed from red to black, therefore cannot create a red violation.

State 2 is the problematic one. Since the black height has been reduced by one, we cannot reinsert the new tree into its parent node. Its sibling has a different black height. However the reduced tree is consistent. Since its color is black and unchanged, it does not create any red violation.

In state 2 we have to reorganize the sibling and the parent to end up in state 1 or 2.

Deletion of a Singleton Node

Deletion of a singleton node initializes the invariant.

If the deleted node is red, we start in state 1. The black height is the same (zero). Its color has changed from red to black.

If the deleted node is black, we start in state 2. Its black height has reduced by one (from 1 to 0) and its color is unchanged and black.

Integrate a Subtree with Deletion into the Parent

Let's assume we have `Node(color, left, info, right)` and have successfully deleted one info element in the right child getting back `rnew` either in state 1 or state 2 (we describe only deletion in the right child; deletion in the left child is symmetrical). The black height of `rnew` is `h >= 0` which depending on the state is either reduced or not.

State 1 is trivial. We return `Node(color, left, info, rnew)` in state 1.

In state 2 we have to analyze the sibling `left` in order to see, how we can reorganize the tree.

We know that the black height of `left` is `h + 1`, because we are in state 2. We split up `left` to get subtrees of black height `h` or `h + 1`. We want a right subtree of `left` which is black and has the black height `h` in order to combine it with `rnew` without problems.

We have to distinguish the following four cases for `left`.

```
    Bx                                  Bx
a       Bb                          a       Ry
                                          Bb   Bc
```

```
    Rx                                  Rx
Ba+     By                          Ba+     By
      b   Bc                              b     Rz
                                              Bc   Bd
```

Since `left` is a valid red black tree with black height `h + 1`, there are no other possibilities. So we have the four ordered sequences

```
    a x Bb

    a x y b Bc

    a+ x b y Bc

    a+ x b y z Bd
```

where the rightmost subtree is always black (indicated by `B`). The subtrees `a`, `b`, `c` and `d` have all a black height of `h` except for the last two sequences, where `a` has a black height of `h + 1` (indicated by `+`).

Now we have to do a case analysis distinguishing the two possible colors of the parent node and combine the cases with the possible four cases of the sibling.

Parent color is red:

The sibling cannot be red. Therefore we have to consider only the first two cases of the sibling.

We can create in both cases a new parent which has black height `h + 1` and whose color is either unchanged or changed from red to black. I.e. we end in state 1. Here are the two possibilities for the new parent node. It is not difficult to verify the invariants of the red black tree and the conditions for reaching state 1.

```
               Bx                          <-- h + 1; state 1
           a       Rinfo
                     Bb   rnew


               Ry                          <-- h + 1; state 1
           Bx      Binfo
          a  b      Bc   rnew
```

Parent color is black:

In that case all four cases for the sibling are possible. In order to end up in state 1 we have to build a new parent with black height `h + 2`. In order to reach state 2, it is sufficient to build a new parent node with black height `h + 1`.

```
               Bx                          <-- h + 1; state 2
           a       Rinfo
                  Bb    rnew


               By                          <-- h + 2; state 1
           Bx      Binfo
         a    b   Bc   rnew
```

```
      Bx                           <-- h + 2; state 1
  a+      By
        b     Binfo
            Bc   rnew

          Bx                       <-- h + 2; state 1
  a+              Rz
              By      Binfo
            b   c    Bd   rnew
```

Termination of Deletion

The deletion algorithm terminates if we reach the root. Since our deletion invariant guarantees valid red black trees in each state, there is nothing more to be done.

Summary

We have described insertion and deletion in a red black tree. The cases to be analyzed have been minimized.

- Insertion is done by maintaining an insertion invariant with 3 states. Only one state describes a red black tree with a red violation of the invariant. The insertion algorithm just needs a case split on the color of the parent node in one of the states. In the other two states, no case analysis is necessary.
- Deletion is done by maintaining a deletion invariant with 2 states. None of the states contains a tree with a violated invariant. However restructuring with the sibling might be necessary if the black height has been reduced. The restructuring requires only four cases in the sibling of the reduced node and two cases for the possible color of the parent.