J. Functional Programming **1** (1): 1–000, January 1993 © 1993 Cambridge University Press 1

FUNCTIONAL PEARLS

Red-Black Trees in a Functional Setting

CHRIS OKASAKI[†]
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, Pennsylvania, USA 15213
(e-mail: cokasaki@cs.cmu.edu*)*

1 Introduction

Everybody learns about balanced binary search trees in their introductory computer science classes, but even the stouthearted tremble at the thought of actually *implementing* such a beast. The details surrounding rebalancing are usually just too messy. To show that this need not be the case, we present an algorithm for insertion into red-black trees (Guibas & Sedgewick, 1978) that any competent programmer should be able to implement in fifteen minutes or less.

2 Red-Black Trees

A red-black tree is a binary tree where every node is colored either red or black. In Haskell (Hudak *et al.* , 1992), this might be represented as

```
data Color = R | B
data Tree elt = E | T Color (Tree elt) elt (Tree elt)
```

We will use this representation to implement sets. To implement other abstractions (e.g., finite maps) or fancier operations (e.g., find the ith largest element), we would augment the `T` constructor with extra fields.

As with all binary search trees, the elements in a red-black tree are stored in symmetric order, so that for any node `T color a x b`, `x` is greater than any element in `a` and less than any element in `b`. In addition, red-black trees satisfy two balance invariants:

Invariant 1. No red node has a red parent.
Invariant 2. Every path from the root to an empty node contains the same number of black nodes.

[†] This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050.

For the purposes of these invariants, empty nodes are considered to be black.

Taken together, these invariants ensure that every tree is balanced — and thus that most operations take no more than $O(\log n)$ time — because the longest possible path in a tree, one with alternating black and red nodes, is no more than twice as long as the shortest possible path, one with black nodes only.

3 Simple Set Operations

The simplest set operations are those requiring no rebalancing.

```
type Set a = Tree a

empty :: Set elt
empty = E

member :: Ord elt => elt -> Set elt -> Bool
member x E = False
member x (T _ a y b) | x <  y = member x a
                     | x == y = True
                     | x >  y = member x b
```

Except for the occasional wildcard, these are exactly the same as the equivalent operations on unbalanced search trees.

4 Insertions

Next, we consider the `insert` operation, which adds a new element to a set. This is where things start to get interesting because we need to add a new node without violating the red-black balance invariants. The skeleton of this function is

```
insert :: Ord elt => elt -> Set elt -> Set elt
insert x s = makeBlack (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)

        makeBlack (T _ a y b) = T B a y b
```

This is identical to the corresponding operation on unbalanced search trees, except for three things. First, when we create a new node (the `ins E` case), we initially color that node red. Second, we force the root of the final result to be black. Finally, instead of directly rebuilding the node after each of the two recursive calls to `ins`, we call the function `balance`. This balancing function is the heart of the algorithm.

By coloring the new node red, we maintain Invariant 2, but we might be violating Invariant 1. We make detecting and repairing such violations the responsibility of the black grandparent of the red node with the red parent. There are four dangerous

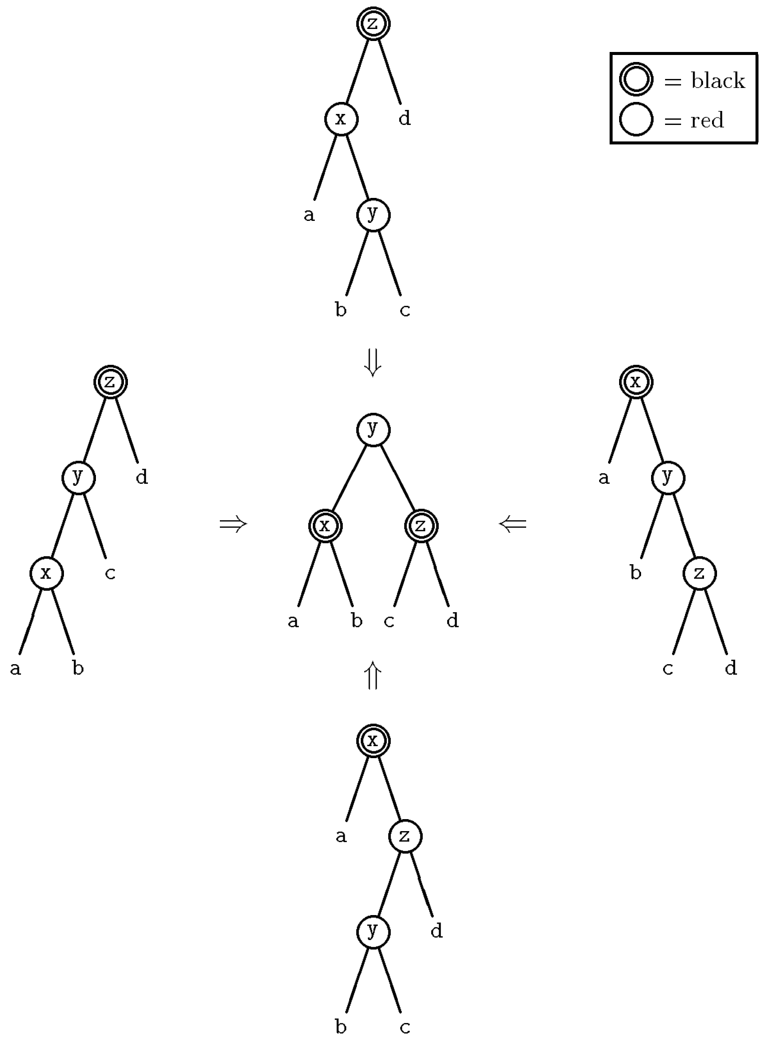

Fig. 1. Eliminating red nodes with red parents.

cases, depending on whether each red node is a left or right child. In all other cases, we simply rebuild the node with the given fields.

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

For each of the four dangerous cases, the solution is the same: rearrange the black node and the two red nodes into a tree with a red root and two black children, as shown pictorially in Figure 1. Note that there is only one way to do this that preserves the order of the elements. It is routine to verify that the red-black balance invariants both hold for the resulting tree.

The `balance` function now looks like

```
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 color a x b = T color a x b
```

Notice that the right-hand sides of the first four clauses are identical. Some functional languages support a feature known as *or-patterns* that allows multiple clauses with identical right-hand sides to be collapsed into a single clause (Fähndrich & Boyland, 1997). Inventing a syntax for or-patterns in Haskell, the `balance` function might be re-written as

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

After balancing a given subtree, the red root of that subtree might now be the child of another red node. Thus, we continue balancing all the way to the top of the tree. At the very top of the tree, we might end up with a red node with a red parent, but with no black grandparent to take responsibility for rewriting the tree. We handle this case by always recoloring the root to be black.

5 What happened to all the mess?

Why is this implementation so much simpler than other implementations of red-black trees? Certainly the use of algebraic datatypes and pattern matching leads to a particularly pleasant expression of the case analysis in `balance`, but a more significant reason is that this implementation of red-black trees uses subtly different transformations from previous implementations.

Most implementations separate each case of a red node with a red parent into two subcases according to the color of the red parent's sibling. This doubles the number of interesting cases from four to eight, but more importantly, it leads to substantially different actions for many of the cases, rather than the same action for all cases. Figure 2 illustrates the three kinds of actions: color flips, single rotations, and double rotations.

These alternative rules can be implemented as shown in Figure 3, using or-patterns to highlight similar actions. Instead of five cases and two different actions, there are now nine cases and five different actions. Furthermore, the order of the cases now becomes significant since the rules for single and double rotations assume that the red parent's sibling is black.

What is the point of all this extra complexity? Were the inventors of red-black

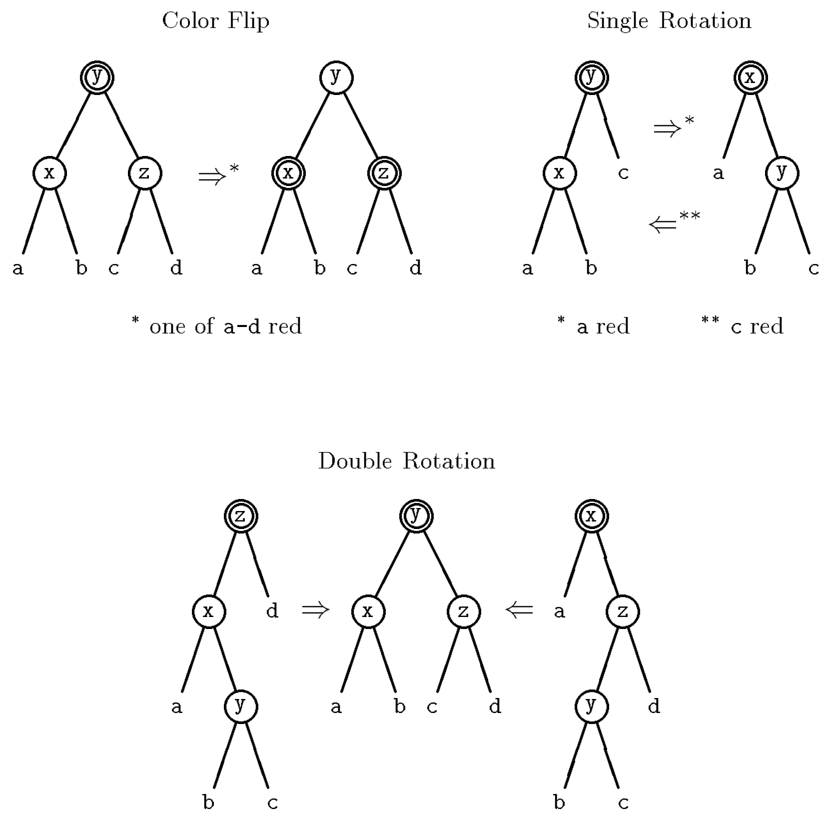

Fig. 2. Alternative balancing transformations. Subtrees `a-d` all have black roots unless otherwise indicated.

```
-- color flips
balance B (T R a@(T R _ _ _) x b) y (T R c z d)
     || B (T R a x b@(T R _ _ _)) y (T R c z d)
     || B (T R a x b) y (T R c@(T R _ _ _) z d)
     || B (T R a x b) y (T R c z d@(T R _ _ _)) = T R (T B a x b) y (T B c z d)
-- single rotations
balance B (T R a@(T R _ _ _) x b) y c = T B a x (T R b y c)
balance B a x (T R b y c@(T R _ _ _)) = T B (T R a x b) y c
-- double rotations
balance B (T R a x (T R b y c)) z d
     || B a x (T R (T R b y c) z d) = T B (T R a x b) y (T R c z d)
-- no balancing necessary
balance color a x b = T color a x b
```

Fig. 3. Alternative implementation of `balance`.

trees simply stupid? No. In an imperative setting, there are good reasons for preferring these alternative transformations. For example, each of the color flips can be implemented in three assignments to color fields, as opposed to seven or more assignments to color and pointer fields for the corresponding transformations in the earlier version of `balance`. In a functional setting, though, where we create new nodes rather than modifying old ones, these savings in assignments are illusory.

To understand the advantage of the rules for single and double rotations, recall that an imperative implementation of `insert` typically operates in two phases: a top-down *search* phase and a bottom-up *rebalancing* phase. Any rule that results in a subtree with a black root allows the rebalancing phase to terminate early, rather than continuing all the way to the top of the tree. In a functional setting, `insert` also operates in two phases: a top-down *search* phase and a bottom-up *construction* phase, which includes rebalancing. There is no good way to terminate the construction phase early, so there is no good reason to favor rules that generate black roots.

Of course, even in an imperative setting, one has to wonder whether these advantages are worth the extra mess, especially in introductory textbooks such as (Cormen *et al.*, 1990).

6 Conclusions

When existing imperative algorithms can be implemented in functional languages, the results are often much prettier than the original version. This has been amply demonstrated in the past for various kinds of balanced binary search trees, including 2-3 trees (Reade, 1992), weight-balanced trees (Adams, 1993), and AVL trees (Núñez *et al.*, 1995). But we should not stop there! Sometimes we can do even better by revisiting each design decision, and making choices appropriate for a functional setting rather than an imperative setting.

Of course, an elegant program that runs very slowly is worthless. But one of the things that makes computer science so much fun is that elegance and speed often go hand in hand. That is certainly true in this case. Even without further optimization, this implementation of balanced binary search trees is one of the fastest around. And with suitable optimizations — for example, replacing the three-way comparisons in `member` with two-way comparisons (Andersson, 1991) and specializing `balance` to inspect the colors of nodes only along the search path — this implementation really flies.

Acknowledgments

Thanks to Graeme Moss for his comments on an earlier draft of this paper.

References

Adams, S. (1993) Efficient sets—a balancing act. *Journal of Functional Programming* **3**(4):553–561.

Andersson, A. (1991) A note on searching in a binary search tree. *Software—Practice and Experience* **21**(10):1125–1128.

Cormen, T. H., Leiserson, C. E. and Rivest, R. L. (1990) *Introduction to algorithms.* MIT Press.

Fähndrich, M. and Boyland, J. (1997) Statically checkable pattern abstractions. *ACM SIGPLAN International Conference on Functional Programming* pp. 75–84.

Guibas, L. J. and Sedgewick, R. (1978) A dichromatic framework for balanced trees. *IEEE Symposium on Foundations of Computer Science* pp. 8–21.

Hudak, P., *et al.* . (1992) Report on the functional programming language Haskell, Version 1.2. *SIGPLAN Notices* **27**(5).

Núñez, M., Palao, P. and Peña, R. (1995) A second year course on data structures based on functional programming. *Functional Programming Languages in Education.* LNCS 1022, pp. 65–84. Springer-Verlag.

Reade, C. M. P. (1992) Balanced trees with removals: an exercise in rewriting and proof. *Science of Computer Programming* **18**(2):181–204.