Constructing Red-Black Trees

Ralf Hinze

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany

ralf@informatik.uni-bonn.de
http://www.informatik.uni-bonn.de/~ralf/

Abstract

This paper explores the structure of red-black trees by solving an apparently simple problem: given an ascending sequence of elements, construct, in linear time, a red-black tree that contains the elements in symmetric order. Several extreme red-black tree shapes are characterized: trees of minimum and maximum height, trees with a minimal and with a maximal proportion of red nodes. These characterizations are obtained by relating tree shapes to various number systems. In addition, connections to left-complete trees, AVL trees, and half-balanced trees are highlighted.

1 Introduction

Red-black trees are an elegant search-tree scheme that guarantees $O(\log n)$ worst-case running time of basic dynamic-set operations. Recently, C. Okasaki [10, 11] presented a beautiful functional implementation of red-black trees. In this paper we plunge deeper into the structure of red-black trees by solving an apparently simple problem: given an ascending sequence of elements, construct a red-black tree that contains the elements in symmetric order. Since the sequence is ordered, the construction should only take linear time.

There are at least two ways of approaching this problem. One can try to analyse and to improve the standard method, which works by repeatedly inserting elements into an empty initial tree. Or one can build upon well-known algorithms for constructing trees of minimum height [4, 5]. In the latter case one must solve the following related problem: given an arbitrary binary search-tree, is there a way of coloring the nodes such that a red-black tree emerges?

We follow both paths as each provides us with different insights into the structure of red-black trees. Along the way, we will encounter several extreme red-black tree shapes: trees of minimum and maximum height, trees with a minimal proportion of red nodes, and others with a maximal proportion. In addition, connections to left-complete trees [13], AVL trees [2], and half-balanced trees [12] are highlighted.

2 Functional red-black trees

Let us start with a brief review of C. Okasaki's functional red-black trees [10, 11]. A red-black tree is a binary tree whose nodes are colored either red or black.

$$\begin{array}{lcl}\textbf{data}\ \mathit{Color} & = & R \mid B \\ \textbf{data}\ \mathit{RBTree}\ a & = & E \mid N\ \mathit{Color}\ (\mathit{RBTree}\ a)\ a\ (\mathit{RBTree}\ a)\end{array}$$

The balance conditions are best explained if we take a look at their historical roots. Red-black trees were developed by R. Bayer [3] under the name *symmetric binary B-trees*. This term indicates that red-black trees were originally designed as binary tree representations of *2-3-4 trees*. Recall that a 2-3-4 tree consists of 2-, 3- and 4-nodes (a 3-node, for instance, has 2 keys and 3 children) and satisfies the invariant that all leaves appear on the same level. The idea of red-black trees is to represent 3- and 4-nodes by small binary trees, which consist of a black root and one or two auxiliary red children. This explains the following two balance conditions.

Red condition: Each red node has a black parent.

Black condition: Each path from the root to an empty node contains exactly the same number of black nodes (this number is called the tree's *black height*).

Note that the red condition implies that the root of a red-black tree is black.

The algorithm for inserting an element into a red-black tree is nearly identical to the standard algorithm for unbalanced binary trees. The main difference is that the constructor for building nodes, *N*, is replaced by a *smart constructor* [1] that maintains the invariants.

```
insert              ::  (Ord a) ⇒ a → RBTree a → RBTree a
insert a t          =   blacken (ins t)
  where ins E       =   N R E a E
        ins (N c l b r)
            | a < b     =   bal c (ins l) b r
            | a == b    =   N c l a r
            | a > b     =   bal c l b (ins r)
blacken (N _ l a r) =   N B l a r
```

Since a new node is colored red, only the red condition is possibly violated. The smart constructor *bal* detects and repairs such violations.

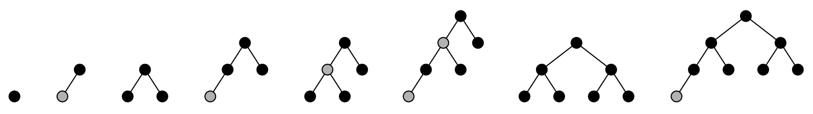
```
bal B (N R (N R t1 a1 t2) a2 t3) a3 t4   =  N R (N B t1 a1 t2) a2 (N B t3 a3 t4)
bal B (N R t1 a1 (N R t2 a2 t3)) a3 t4   =  N R (N B t1 a1 t2) a2 (N B t3 a3 t4)
bal B t1 a1 (N R (N R t2 a2 t3) a3 t4)   =  N R (N B t1 a1 t2) a2 (N B t3 a3 t4)
bal B t1 a1 (N R t2 a2 (N R t3 a3 t4))   =  N R (N B t1 a1 t2) a2 (N B t3 a3 t4)
bal c l a r                              =  N c l a r
```

The simplest way to construct a red-black tree is to repeatedly insert elements into an empty initial tree.

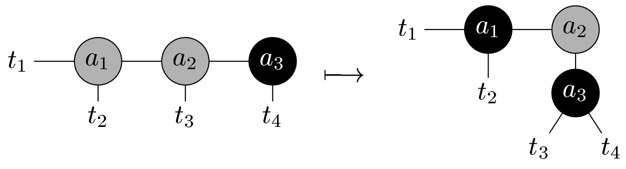
```
top-down  ::  (Ord a) ⇒ [a] → RBTree a
top-down  =   foldr insert E
```

3 A closer look at *top-down*

What tree shapes does *top-down* produce when the given sequence is ascending? It is instructive to peek at some small examples first. The following trees are generated by *top-down* $[1 \,..\, i]$ for $1 \leqslant i \leqslant 8$ ('◦' is a red node and '●' is a black node).

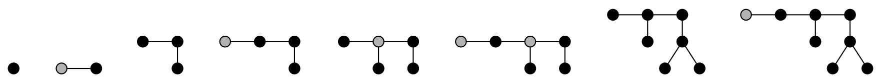


Note that we do not care to label the nodes as the keys are uniquely determined by the search-tree property. Since the list is processed from right to left, the elements are inserted in descending order. Consequently, *ins* always traverses the *left spine* of the tree to the leftmost leaf. The examples show that the color of the leftmost leaf alternates between black and red implying that the red condition is violated in every second step. Because *ins* always branches to the left, only the first equation of the smart constructor *bal* can possibly match. If we draw the left spine horizontally, the balancing operation takes the following form.



The balancing operation paints the first node black and combines the next two putting the black node below the second red one. Since this is the only operation applicable, all nodes not on the left spine must be black. We know even more: the black condition implies that the trees below the left spine (t_2, t_3 and t_4) must be perfectly balanced binary trees (perfect trees for short). Thus, the generated red-black trees correspond to sequences of *topped perfect trees*. A topped tree is a tree with an additional unary node on top. Topped perfect trees are, in fact, a widespread plant in the design and analysis of data structures. J.-R. Sack and T. Strothotte [13], who call them *pennants*, employ them to design algorithms for splitting and merging heaps in the form of *left-complete* binary trees. In a left-complete tree all leaves appear on at most two adjacent levels and the leaves on the lowest level are in the leftmost possible positions. We will exhibit further connections to their work in Section 6. The author recently showed that pennants also underly binomial heaps [7].

A topped perfect tree or a pennant of *rank* r is a perfect tree of height r with an additional node on top. It follows that a pennant of rank r contains exactly 2^r nodes. Turning to the analysis of *top-down* $[1\,..\,i]$ we are left with the task of determining the pennants' ranks. It is helpful to redraw our examples according to the *left-spine view*.



A pattern begins to emerge: let r be the rank of the rightmost pennant; the black condition implies that a pennant of rank i appears either once or twice for all $0 \leqslant i \leqslant r$. Since the size of a rank i pennant is 2^i, we have that the trees correspond to 'binary numbers' composed of the digits 1 and 2. It is worthwhile to study this number system, which we call 1-2 system, in more detail. Recall that the value of the radix-2 number $(b_{n-1} \ldots b_0)_2$ is $\sum_{i=0}^{n-1} b_i 2^i$. Since the number system abandons the digit 0 in favour of the digit 2, each natural number has, in fact, a unique representation. It is conceivable that the number system was already known in the middle ages when the number 0 was frowned upon and fell into oblivion later. Purists are probably attracted by the fact that there is no need to disallow leading zeros. Counting is easy:

$$()_2, (1)_2, (2)_2, (11)_2, (12)_2, (21)_2, (22)_2, (111)_2, (112)_2 \ldots$$

Note that 0 is represented by the empty sequence. The increment is similar to the ordinary binary increment; we have $s1 + 1 = s2$ and $s2 + 1 = (s + 1)1$. In the sequel we use the notation $(b_{n-1} \ldots b_0)_{1\text{-}2}$ to emphasize that the digits are drawn from the set $\{1, 2\}$.

We have seen that red-black trees generated by *top-down* $[1\,..\,n]$ are uniquely determined by the 1-2 decomposition of n. Let us examine some examples: $(1^{\{n\}})_{1\text{-}2} = 2^n - 1$ corresponds to a perfect tree of height n ($d^{\{n\}}$ means the digit d repeated n times); left-complete trees are produced for $(1^{\{n\}}2)_{1\text{-}2} = 2^{n+1}$ and $(21^{\{n\}}) = 3 \cdot 2^n - 1$. Hence, we know that *top-down* $[1\,..\,i]$ produces trees of minimum height for some values of i. This is good news. On the other hand $(2^{\{n\}})_{1\text{-}2} = 2^{n+1} - 2$ and $(12^{\{n\}})_{1\text{-}2} = 3 \cdot 2^n - 2$ correspond to *skinny trees* of height $2n$ and $2n + 1$, respectively. A skinny tree is a tree of smallest possible size for a given height. Fig. 1 depicts a skinny tree of height 7 and its 'successor', which is a left-complete tree. Note that we cannot remove a single node from the skinny tree without either lowering the tree's height

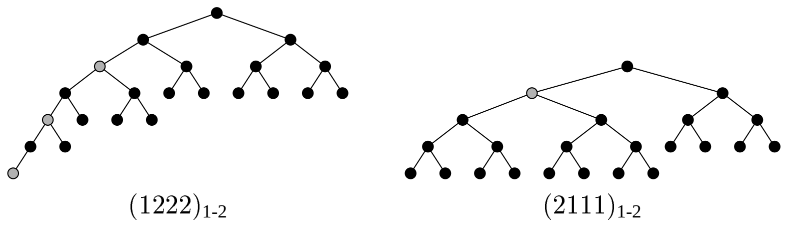


Figure 1: A skinny tree of height 7 and its 'successor'

or violating the black condition. Skinny trees give us a precise upper bound for the height of a red-black tree [14]:

$$\textit{height } t \leqslant 2\, \textit{lg}\ (\textit{size } t + 2) - 2\ .$$

So the bad news is that *top-down* $[1 \mathinner{..} i]$ produces red-black trees of maximum height for some values of i.

The trees generated by *top-down* have another intriguing property. They contain the minimal number of red nodes among all red-black trees of that size.

Sketch of proof. The central idea is to show that each tree of size i can be transformed into the shape generated by *top-down* $[1 \mathinner{..} i]$ and that the transformations do not increase the number of red nodes. We base the proof on 2-3-4 trees, which underly red-black trees, see Section 2. The shape of a 2-3-4 tree is uniquely represented by a sequence of level descriptions where each level description is a sequence of the numbers 2, 3 and 4. As an example, the trees of Fig. 1 are represented by

2	3
3 2	2 2 2
3 2 2 2 2	2 2 2 2 2 2
3 2 2 2 2 2 2 2 2 2 2	2 2 2 2 2 2 2 2 2 2 2 2 .

To simplify the proof we consider not only 2-3-4 trees but general multiway branching trees and transformations on these trees. To this end we generalize level descriptions to sequences of arbitrary natural numbers. We only require the following 'sanity' condition: A sequence of level descriptions $\ell_n \ell_{n-1} \ldots \ell_2 \ell_1$ is *valid* iff $|\ell_n| = 1$ and $\sum \ell_{i+1} = |\ell_i|$ for $1 \leqslant i < n$. We use two kinds of transformations. The first transformation replaces a subsequence of numbers in a certain level by another subsequence and is depicted $s \leadsto s'$. To ensure that the resulting tree is valid s and s' must satisfy $\sum s = \sum s'$ and $|s| = |s'|$. The second transformation is $s \overset{+k}{\leadsto} s'$ with $k \geqslant 1$, $\sum s = \sum s'$, and $|s| + k = |s'|$. Here, s is replaced by s' in a certain level and the level above is increased by k, ie one or more numbers in this level are increased by a total of k. If there is no level above, we silently create a new level consisting of a 1-node. The reader is invited to relate the two transformations to operations on multiway branching trees. Note that $s \leadsto s'$ does not affect the number of red nodes and that $s \overset{+k}{\leadsto} s'$ decreases the number of red nodes by k (recall that a node of size n consists of one black node and $n - 1$ red nodes). Now, an arbitrary 2-3-4 tree can be transformed into the desired shape by repeatedly applying the following transformations from the bottom level to the top level.

$$2\,3 \leadsto 3\,2 \qquad 3\,3 \overset{+1}{\leadsto} 2\,2\,2 \qquad (n+4) \overset{+1}{\leadsto} (n+2)\,2$$

In the resulting tree each level description has the form $[3]2^*$, ie an optional 3-node followed by an arbitrary number of 2-nodes. Since the tree generated by *top-down* satisfies the same property and since each number has a unique 1-2 decomposition, the claim follows. $\square$

Using the 1-2 number system we can even quantify the minimal number of red nodes: a red-black tree of size n contains at least k red nodes where k is the number of 2's in the 1-2 representation of n.

4 Improving *top-down*

The analogy to the 1-2 number system can be exploited to give a better implementation of *top-down* for the special case that the elements appear in ascending order. The digits become containers for pennants:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Digit}\ a & = & \mathit{One}\ a\ (\mathit{RBTree}\ a) \\ & \mid & \mathit{Two}\ a\ (\mathit{RBTree}\ a)\ a\ (\mathit{RBTree}\ a)\ \ . \end{array}$$

A red-black tree is represented by a list of digits in increasing order of size (the least significant digit comes first). Inserting an element corresponds to incrementing a 1-2 number. The function *incr*, which does the job, essentially implements the two laws $s1+1 = s2$ and $s2+1 = (s+1)1$ where s is any sequence of 1-2 digits.

$$\begin{array}{lcl} \mathit{incr} & :: & \mathit{Digit}\ a \to [\mathit{Digit}\ a] \to [\mathit{Digit}\ a] \\ \mathit{incr}\ (\mathit{One}\ a\ t)\ [\,] & = & [\mathit{One}\ a\ t] \\ \mathit{incr}\ (\mathit{One}\ a_1\ t_1)\ (\mathit{One}\ a_2\ t_2 : \mathit{ps}) & = & \mathit{Two}\ a_1\ t_1\ a_2\ t_2 : \mathit{ps} \\ \mathit{incr}\ (\mathit{One}\ a_1\ t_1)\ (\mathit{Two}\ a_2\ t_2\ a_3\ t_3 : \mathit{ps}) & = & \mathit{One}\ a_1\ t_1 : \mathit{incr}\ (\mathit{One}\ a_2\ (N\ B\ t_2\ a_3\ t_3))\ \mathit{ps} \end{array}$$

The reader is invited to relate *incr* to the definitions of *ins* and *bal* given in Section 2. The rest is easy: we repeatedly insert elements into the list of digits; the final result is converted to a red-black tree.

$$\begin{array}{lcl} \mathit{bottom\text{-}up} & :: & [a] \to \mathit{RBTree}\ a \\ \mathit{bottom\text{-}up} & = & \mathit{linkAll} \cdot \mathit{foldr}\ \mathit{add}\ [\,] \\ \mathit{add}\ a\ \mathit{ps} & = & \mathit{incr}\ (\mathit{One}\ a\ E)\ \mathit{ps} \\ \mathit{linkAll} & = & \mathit{foldl}\ \mathit{link}\ E \\ \mathit{link}\ l\ (\mathit{One}\ a\ t) & = & N\ B\ l\ a\ t \\ \mathit{link}\ l\ (\mathit{Two}\ a_1\ t_1\ a_2\ t_2) & = & N\ B\ (N\ R\ l\ a_1\ t_1)\ a_2\ t_2 \end{array}$$

It is a routine matter to prove *bottom-up* correct. We must essentially show that *add* implements *insert* on the left-spine view (*labels* lists the labels of a red-black tree):

$$\mathit{all}\ (a{<})\ (\mathit{labels}\ t) \implies \mathit{linkAll}\ (\mathit{add}\ a\ t) = \mathit{insert}\ a\ (\mathit{linkAll}\ t)\ \ .$$

The reimplementation of *top-down* is worth the effort: a standard amortization argument shows that *bottom-up* takes only linear time.

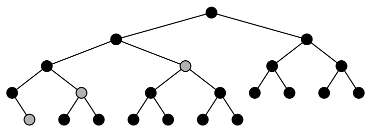
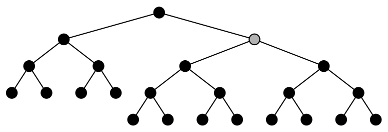
Remark. Red-black trees under the left-spine view correspond closely to *finger search-trees* [6]. A finger search-tree is a representation of an ordered list that allows for efficient insertion in the vicinity of certain points, termed fingers. Here we have a single static finger at the front end of the list. This data structure may be of further interest because it makes a nice implementation of updatable priority queues, which support deleting and decreasing a key.

5 Less height, please!

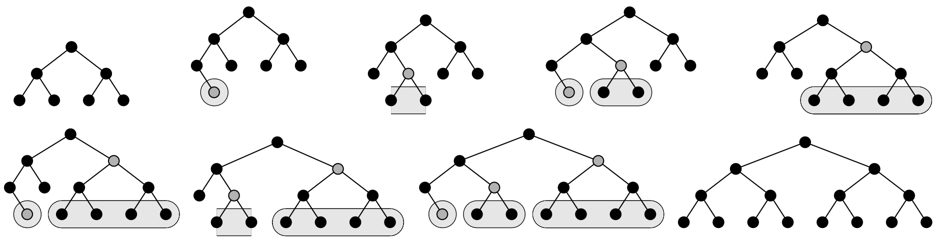
Having succeeded in implementing *top-down* efficiently, let us now try to reduce the height of the generated trees. It is well-known [8] that a binary search-tree is optimal if all leaves appear on at most two adjacent levels—under the assumption that all keys are equally likely. It turns out that it is almost trivial to modify *bottom-up* so that it produces trees of that shape. A simple *rotation to the right* suffices:

$$\mathit{link}'\ l\ (\mathit{Two}\ a_1\ t_1\ a_2\ t_2) \quad = \quad N\ B\ l\ a_1\ (N\ R\ t_1\ a_2\ t_2)\ .$$

Here are the shallow variants of the trees shown in Fig. 1.

It is interesting to see how the leaves on the bottom level are arranged. The pattern becomes apparent if we take a look at a longer sequence of trees.



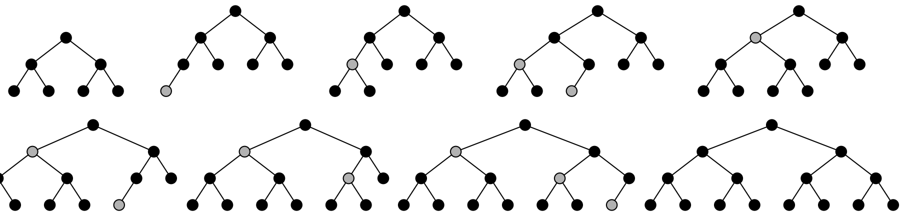
In each case the leaves appear on two adjacent levels. The definition of *link*′ brings about that the leaves on the lowest level are descendants of a red node (apart from perfect trees). Depending on the position of the red node we have either a group of 1, 2 or 4 nodes as indicated by the shading. If we convert the groups into binary digits, the binary numbers $(000)_2$, $(001)_2$, $(010)_2$, …, $(111)_2$ appear on the last level. The number system helps to explain why this is the case. The 1-2 number $(b_{n-1} \ldots b_0)_{\text{1-2}}$ can be decomposed into two binary numbers: $(b_{n-1} \ldots b_0)_{\text{1-2}} = (1 \ldots 1)_{\text{0-1}} + (b'_{n-1} \ldots b'_0)_{\text{0-1}}$ with $b'_i = b_i - 1$. The number $(1 \ldots 1)_{\text{0-1}}$ corresponds to the perfect tree on top; the residue $(b'_{n-1} \ldots b'_0)_{\text{0-1}}$ to the leaves on the bottom level.

6 Digression: Left-complete binary heaps

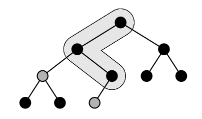
The generated trees, which we call *quasi left-complete trees*, are closely related to *left-complete trees*, which have the leaves on the lowest level in the leftmost possible positions. Consider the parents of the red nodes. If we *swap* their children, we obtain a left-complete tree. Again, it is trivial to modify *link* accordingly:

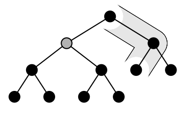
$$\mathit{link}''\ l\ (\mathit{Two}\ a_1\ t_1\ a_2\ t_2) \quad = \quad N\ B\ (N\ R\ t_1\ a_2\ t_2)\ a_1\ l\ .$$

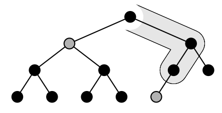
To complete the picture here are the left-complete colleagues of the trees above.



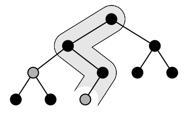
Of course, the above transformation does not preserve the search-tree property. So let us assume in this section that we deal with *binary heaps* instead. It is not difficult to adopt *incr* to the new situation. We change our point of view because this provides an interesting link to the work of J.-R. Sack and T. Strothotte [13]. The decomposition of a left-complete tree into a list of pennants also lies at the heart of their algorithms for splitting and merging heaps. There is, however, a slight difference. They decompose a left-complete heap along the path from the root to the last leaf, ie the rightmost leaf on the last level. This seems to be an obvious choice but as we shall see gives rise to a more complicated number system. Here are the two ways of decomposing a left-complete tree.

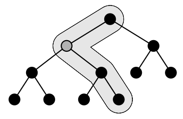

$(122)_{\text{1-2}} = (210)_{\text{0-1-2}}$

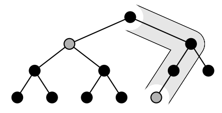

$(211)_{\text{1-2}} = (1011)_{\text{0-1-2}}$


$(212)_{\text{1-2}} = (1020)_{\text{0-1-2}}$

versus


$(202)_{\text{0-1-2}'}$


$(211)_{\text{0-1-2}'}$

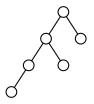

$(1012)_{\text{0-1-2}'}$

The difference is really minor: in the first row we follow the path to the first free position; in the second row to the last occupied position. Given a left-complete tree of size n, the first choice yields $\lfloor \lg(n+1) \rfloor$ pennants while the latter choice gives $\lceil \lg(n+1) \rceil$ pennants. Let us examine the number system corresponding to the latter choice, which we call 0-1-2′ system, for want of a better name. The examples show that the numbers are composed of the digits 0, 1 and 2. The digit d appears in the i-th position iff the path contains d pennants of size 2^i. For instance, the rightmost tree contains one pennant of size 8, one of size 2, and two of size 1. Consequently, the corresponding number is $(1012)_{\text{0-1-2}'}$. Without further restrictions the 0-1-2′ binary system is clearly *redundant*. It turns out that the number $(b_{n-1} \dots b_0)_{\text{0-1-2}'}$ corresponds to a left-complete tree iff $m+1 \leqslant \sum_{i=0}^{m} d_i \leqslant m+2$ for all $m < n$ [13]. This condition implies that we never have two successive 2's. In fact, 21^*2 cannot appear as a subsequence (d^* means d repeated arbitrarily often). Incrementing a 0-1-2′ number is funny: first make a 'normal' increment (this can be done in constant time since the subsequence 22 is forbidden); then apply the transformation $21^{\{n\}}2 \mapsto 101^{\{n\}}2$ (at most once). If a segmented representation is used [10, Section 9.2.4], the latter transformation can also be done in constant time.

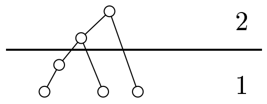
How do the number systems relate to each other? Well, we obtained the left-complete trees using rotations and swaps. A rotation corresponds to the carry propagation $2 \mapsto 10$ turning a 1-2 number into a 0-1-2 number. A swap does not affect the numeric representation. Thus, $(122)_{\text{1-2}}$ becomes $(210)_{\text{0-1-2}}$ and $(211)_{\text{1-2}}$ becomes $(1011)_{\text{0-1-2}}$. Note that the last digit of the 0-1-2 number is either 0 or 1. If we increment the last digit, we get the 0-1-2′ number corresponding to its successor. This relation is not too surprising since the path to the first free position corresponds to the path to the last element in the successor tree.

7 Coloring binary search-trees

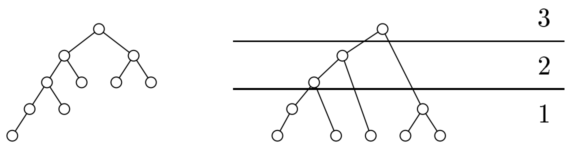
Let us now approach the problem of constructing red-black trees from a different perspective. Say, we are given an arbitrary binary search-tree and we are asked to color the nodes such that a red-black tree emerges or to report that it is not possible to do so. Here is the first test case.



The longest path from the root to an empty tree comprises 4 nodes; the shortest path consists of 2 nodes. Thus, the black height must be two and the nodes on the longest path must be colored black, red, black and red. We get a better picture of the situation if we draw the tree slightly different. In the picture below the vertical position of a node corresponds to its height rather than its distance from the root.



We additionally assign a *level number* to each node: a node of height h receives the level number $\lceil h/2 \rceil$. This way we divide the tree two-levelwise from the bottom to the top. Coloring is now easy: a node is colored red iff it has a parent with the same level number. Does this scheme work in general? Not quite, as the second test case shows.



All nodes of the right subtree must be colored black but our scheme colors the two leaves red. Fortunately, the picture on the right also contains an indication of the failure: an edge crosses two levels, ie the level numbers of two adjacent nodes differ by more than one. We can remedy this defect by lifting the right son of the root to the second level. Generally, it is possible to adjust the level numbers in a single top-down pass. It may, of course, happen that the leaves no longer appear on the same level. In this case the given tree is not colorable.

We are now ready to tackle the implementation. For simplicity, let us assume that the nodes of the input tree are decorated with the level number, ie trees are given as elements of the data type

```
type Level   =  Int
data Tree a  =  Empty | Node Level (Tree a) a (Tree a) ,
```

in which $\lceil height\ (Node\ h\ l\ a\ r)\ /\ 2 \rceil = h$. The algorithm takes the following form:

```
rbtree                         ::  Tree a → RBTree a
rbtree Empty                   =   E
rbtree (Node h l a r)          =   N B (rbtree′ h l) a (rbtree′ h r)
rbtree′                        ::  Level → Tree a → RBTree a
rbtree′ hp Empty | hp == 1     =   E
                 | otherwise   =   error "not a red-black tree"
rbtree′ hp (Node h l a r)      =   N color (rbtree′ h′ l) a (rbtree′ h′ r)
     where h′                  =   h ‘max‘ (hp − 1)
           color | hp == h     =   R
                 | otherwise   =   B .
```

The auxiliary function *rbtree′* receives two arguments: the uncolored tree and the level number, *hp*, of the tree's parent. If the tree is empty, the level number must necessarily be one. Otherwise, the given tree cannot be colored. The root of a non-empty tree is colored red iff its level number *h* coincides with *hp*. The adjusted level number of the root, which is passed to the recursive calls of *rbtree*′, is given by the expression h ‘*max*‘ $(hp - 1)$. Note that the level number of the input tree equals the black height of the generated tree. Furthermore, the longest path in the red-black tree contains alternating black and red nodes (if the height is odd and greater than one, the path starts with two black nodes). This in turn implies that *rbtree* produces trees of minimum black height.

It is relatively easy to see that *rbtree* yields a valid red-black tree. The converse is not so obvious: can we be sure that the given tree is not colorable if *rbtree* signals an error? It turns out that the correctness of the algorithm is best shown using an alternative characterization of red-black trees. Define the *min-height* of a tree as the length of the shortest path from the root to an empty node and the *max-height* as the length of the longest path. A binary tree *t* is said to be *half-balanced* [12] if for every subtree *u* of *t*,

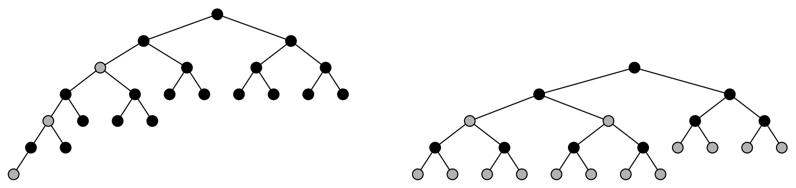
$$\tfrac{1}{2} \leqslant \textit{min-height } u \;/\; \textit{max-height } u \ .$$

Every red-black tree is half-balanced because $\textit{height } t \leqslant 2\,\textit{black-height}$ and $\textit{black-height } t \leqslant \textit{min-height } t$. The function *rbtree* can be viewed as a constructive proof of the reverse implication. One must essentially show that the first parameter of *rbtree*′ satisfies the following invariant

$$\tfrac{1}{2}\,(\textit{max-height } t + 1) \leqslant hp \leqslant \textit{min-height } t + 1 \ .$$

If the input tree satisfies the AVL property [2], the algorithm can be slightly simplified: the test $hp == 1$ becomes obsolete and h' may be safely replaced by h. The resulting function already appears in the seminal paper on red-black trees [3, p. 295].

It is high time to see the algorithm in action. Here are the colored variants of the trees shown in Fig. 1.



It is not hard to show that *rbtree* produces trees with a maximal proportion of red nodes. However, we already know that the skinny tree on the left hand side contains the smallest possible number of red nodes (see Section 3). Both results imply that there is exactly one way of coloring skinny trees.

If *rbtree* is applied to a left-complete tree or to a tree generated by *bottom-up*′, it produces a red-black tree that contains the maximal possible number of red nodes among all trees of that size. Note that it is actually desirable that a tree contains many red nodes since the balancing operation *bal* takes only black nodes into account. To summarize: let *quasi-left-complete* be a variant of *bottom-up*′ that constructs an uncolored tree of type *Tree a*. Then

$$\begin{array}{lcl} \textit{build} & :: & [a] \to \textit{RBTree } a \\ \textit{build} & = & \textit{rbtree} \cdot \textit{quasi-left-complete} \end{array}$$

builds a red-black tree that has minimum path length and a maximal portion of red nodes.

Sketch of proof. It remains to show that *build* constructs a red-black tree with a maximal portion of red nodes. The proof is largely analogous to the one in Section 3. This time we use transformations that do not decrease the number of red nodes. In particular, we employ the transformation $s \overset{-k}{\rightsquigarrow} s'$ with $k \geqslant 1$, $\sum s = \sum s'$, and $|s| - k = |s'|$. This transformation replaces s by s' in a certain level and decreases the level above by k (numbers must not become negative). A given 2-3-4 tree can be transformed into the desired shape by repeatedly applying the following transformations from bottom to top (permutations such as $3\,2 \rightsquigarrow 2\,3$ are omitted from the list).

$$0 \overset{-1}{\rightsquigarrow} \epsilon \qquad 1\,1 \overset{-1}{\rightsquigarrow} 2 \qquad 1\,2 \overset{-1}{\rightsquigarrow} 3 \qquad 1\,3 \overset{-1}{\rightsquigarrow} 4 \qquad 1\,4 \rightsquigarrow 2\,3$$
$$2\,2 \overset{-1}{\rightsquigarrow} 4 \qquad 3\,3 \rightsquigarrow 2\,4$$

We tacitly assume that levels that consist only of a singleton 1-node are silently removed. In the resulting tree each level has the form $[2][3]4^*$, ie an optional 2-node followed by an optional 3-node followed by an arbitrary number of 4-nodes. Using an induction on the length of the left spine one can show that the trees generated by *build* can be transformed into the same shape using only '$\rightsquigarrow$' transformations. Finally, trees of this shape are uniquely determined by the size since they correspond to quaternary numbers composed of the digits 1, 2, 3 and 4 and since this number system is non-redundant. ☐

Remark. In solving the problem of constructing red-black trees we have answered quite a few exercises to be found in textbooks on data structures and algorithms, most notably exercises 10.9, 10.10 and 10.14 in [14] and exercises 3.9 and 9.7 in [10].

8 Acknowledgement

I would like to thank Chris Okasaki for his valuable comments on an earlier draft of this paper. Adam Buchsbaum, Gerth Stølting Brodal, and two anonymous referees made several helpful suggestions. Thanks are also due to Joachim Korittky for implementing *functional* METAPOST [9], which was used for drawing the diagrams.

References

[1] Stephen Adams. Functional Pearls: Efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–561, October 1993.

[2] G.M. Adel'son-Vel'skiĭ and Y.M. Landis. An algorithm for the organization of information. *Doklady Akademiia Nauk SSSR*, 146:263–266, 1962. English translation in Soviet Math. Dokl. 3, pp. 1259–1263.

[3] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[4] Richard S. Bird. Functional algorithm design. *Science of Computer Programming*, 26:15–31, 1996.

[5] Richard S. Bird. Functional Pearl: On building trees with minimum height. *Journal of Functional Programming*, 7(4):441–445, July 1997.

[6] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*, pages 49–60, Boulder, Colorado, May 1977.

[7] Ralf Hinze. Functional Pearl: Explaining binomial heaps. *Journal of Functional Programming*, 9(1):93–104, January 1999.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, 2nd edition, 1998.

[9] Joachim Korittky. *Functional* METAPOST. Diplomarbeit, Universität Bonn, December 1998.

[10] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[11] Chris Okasaki. Functional Pearl: Red-Black trees in a functional setting. *Journal of Functional Programming*, 9(4), July 1999. To appear.

[12] H.J. Olivié. A new class of balanced search trees: Half-balanced search trees. *RAIRO Informatique théoretique*, 16:51–71, 1982.

[13] Jörg-Rüdiger Sack and Thomas Strothotte. A characterization of heaps and its applications. *Information and Computation*, 86(1):69–86, May 1990.

[14] Derick Wood. *Data Structures, Algorithms, and Performance.* Addison-Wesley Publishing Company, 1993.