CONSTRUCTING RED-BLACK TREES

RALF HINZE

Institut für Informatik III, Universität Bonn, Römerstraße 164
53117 Bonn, Germany
Email: ralf@informatik.uni-bonn.de
Hompage: http://www.informatik.uni-bonn.de/~ralf

September, 1999

Problem

Given an ascending sequence of elements, construct, in linear time, a red-black tree that contains the elements in symmetric order.

Approach #1: analyse and improve the standard method, which works by repeatedly inserting elements into an empty initial tree.

Approach #2: build upon well-known algorithms for constructing trees of minimum height. *Subproblem:* given an arbitrary binary search-tree, is there a way of coloring the nodes such that a red-black tree emerges?

Recap: Chris Okasaki's functional red-black trees

A red-black tree is a binary tree whose nodes are colored either red or black.

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

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*).

Recap: insertion

$$
\begin{array}{lcl}
insert & :: & (Ord\ a) \Rightarrow a \rightarrow RBTree\ a \rightarrow RBTree\ a \\
insert\ a\ t & = & blacken\ (ins\ t) \\
\quad \textbf{where} & & \\
\quad ins\ E & = & N\ R\ E\ a\ E \\
\quad ins\ (N\ c\ l\ b\ r) & & \\
\qquad \mid a < b & = & bal\ c\ (ins\ l)\ b\ r \\
\qquad \mid a == b & = & N\ c\ l\ a\ r \\
\qquad \mid a > b & = & bal\ c\ l\ b\ (ins\ r) \\
blacken\ (N\ _\ l\ a\ r) & = & N\ B\ l\ a\ r
\end{array}
$$

Recap: balancing

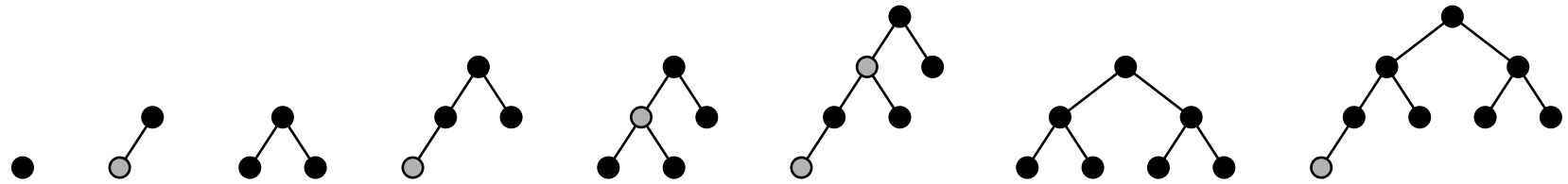
$$
\begin{aligned}
&bal\ B\ (N\ R\ (N\ R\ t_1\ a_1\ t_2)\ a_2\ t_3)\ a_3\ t_4 \\
&\qquad = \quad N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
&bal\ B\ (N\ R\ t_1\ a_1\ (N\ R\ t_2\ a_2\ t_3))\ a_3\ t_4 \\
&\qquad = \quad N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
&bal\ B\ t_1\ a_1\ (N\ R\ (N\ R\ t_2\ a_2\ t_3)\ a_3\ t_4) \\
&\qquad = \quad N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
&bal\ B\ t_1\ a_1\ (N\ R\ t_2\ a_2\ (N\ R\ t_3\ a_3\ t_4)) \\
&\qquad = \quad N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
&bal\ c\ l\ a\ r \quad = \quad N\ c\ l\ a\ r
\end{aligned}
$$

Recap: constructing a red-black tree

$$
\begin{array}{lcl}
top\text{-}down & :: & (Ord\ a) \Rightarrow [a] \rightarrow RBTree\ a \\
top\text{-}down & = & foldr\ insert\ E
\end{array}
$$

Approach #1: a closer look at $top\text{-}down$

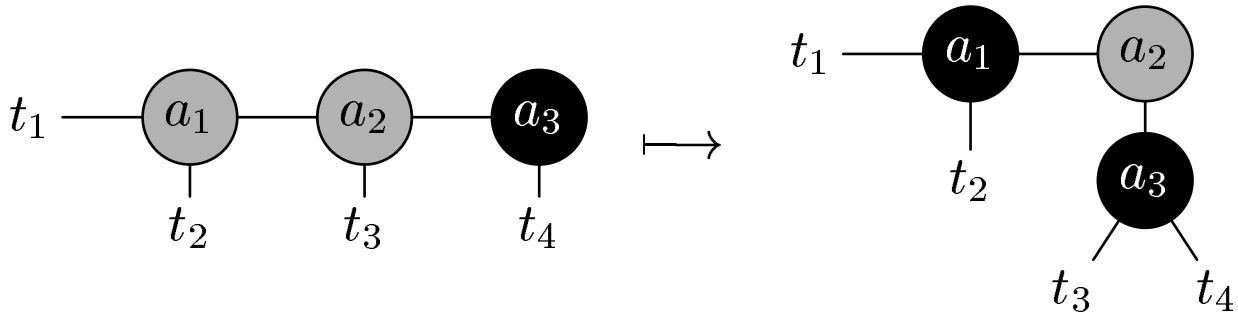
The following trees are generated by $top\text{-}down\,[1\,..\,i]$ for $1 \leqslant i \leqslant 8$ ('○' is a red node and '●' is a black node).



NB. ins always traverses the *left spine* of the tree to the leftmost leaf.

Approach #1: a closer look at $top\text{-}down$

If we draw the left spine horizontally, the balancing operation (first equation of bal) takes the following form.

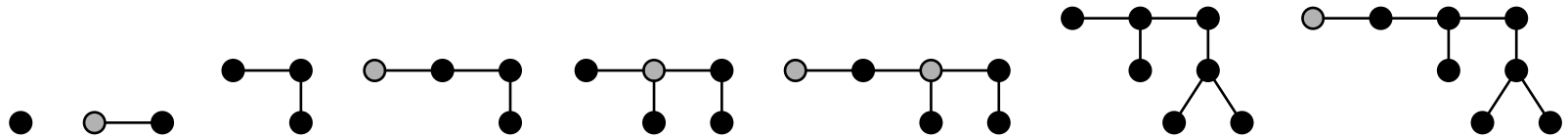


☞ The trees below the left spine (t_2, t_3 and t_4) must be perfectly balanced binary trees (perfect trees for short). Thus, the generated red-black trees correspond to sequences of *topped perfect trees* or *pennants*.

☞ A pennant of rank r contains exactly 2^r nodes.

Approach #1: a closer look at $top\text{-}down$

It is helpful to redraw the examples according to the *left-spine view*.



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 $0 \leqslant i \leqslant r$.

☞ The red-black trees generated by $top\text{-}down$ correspond to 'binary numbers' composed of the digits 1 and 2.

Approach #1: the 1-2 number system

Recall that the value of the radix-2 number $(b_{n-1} \ldots b_0)_2$ is $\sum_{i=0}^{n-1} b_i 2^i$.

Each natural number has a unique representation in the 1-2 number system.

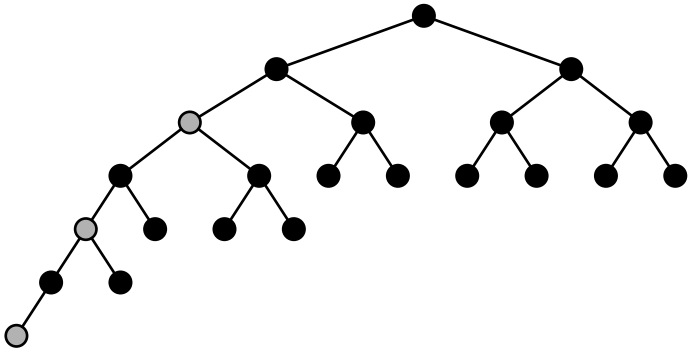
$$()_2, (1)_2, (2)_2, (11)_2, (12)_2, (21)_2, (22)_2, (111)_2, (112)_2 \ldots$$

Incrementing a 1-2 number:

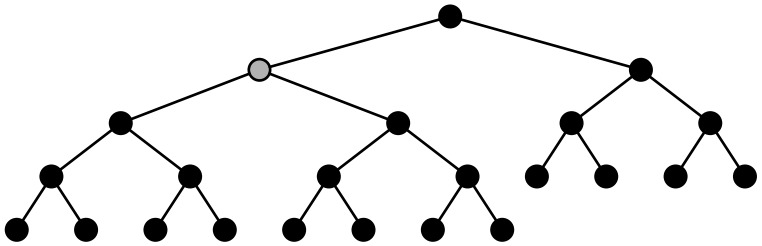
$$\begin{aligned} s1 + 1 &= s2 \\ s2 + 1 &= (s+1)\,1 \ . \end{aligned}$$

Approach #1: corollaries

The trees corresponding to $(12^{\{n\}})_{\text{1-2}}$ are *skinny trees* (a skinny tree is a tree of smallest possible size for a given height); the trees corresponding to $(21^{\{n\}})_{\text{1-2}}$ are *left-complete trees*.



$(1222)_{\text{1-2}}$



$(2111)_{\text{1-2}}$

Approach #1: 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:

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

Inserting an element corresponds to incrementing a 1-2 number.

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

Approach #1: improving $top\text{-}down$

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

If as is ordered, we have $top\text{-}down\ as = bottom\text{-}up\ as$.

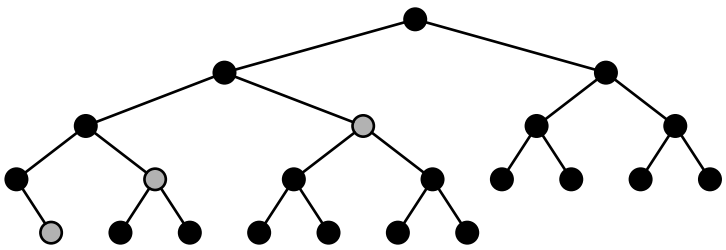
☞ A standard amortization argument shows that $bottom\text{-}up$ runs in linear time.

Approach #1: less height, please!

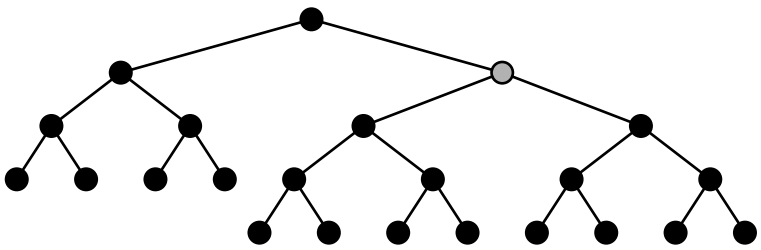
Interestingly, it is easy to modify $bottom\text{-}up$ so that it produces *optimal search-trees*. A simple *rotation to the right* suffices:

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

Here are the shallow variants of the trees above.



$(1222)_{\text{1-2}}$

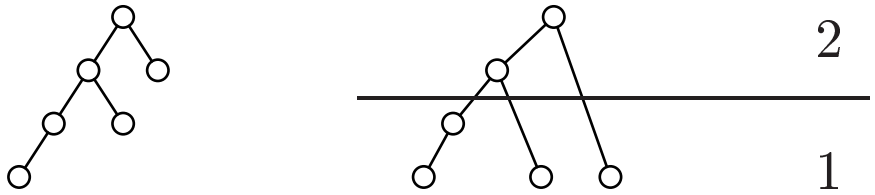


$(2111)_{\text{1-2}}$

Approach #2: coloring binary search-trees

Problem: given an arbitrary binary search-tree, is there a way of coloring the nodes such that a red-black tree emerges?

The first test case.

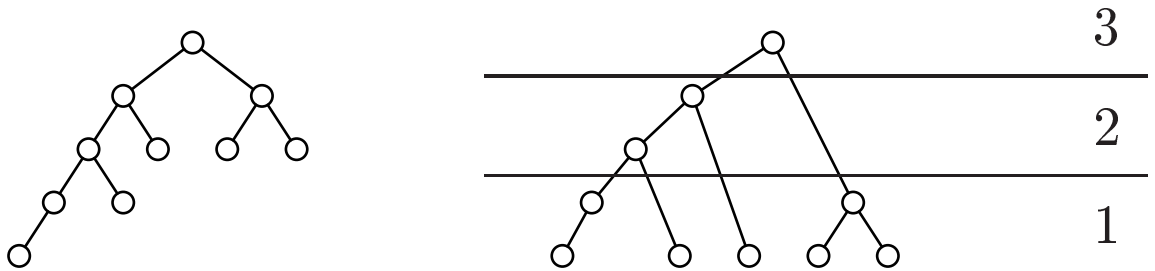


On the right-hand side the vertical position of a node corresponds to its height rather than its distance from the root. We assign a *level number* to each node: a node of height h receives the level number $\lceil h \,/\, 2 \rceil$.

☞ A node is colored red iff it has a parent with the same level number.

Approach #2: coloring binary search-trees

The second test case.



☞ The right son of the root must be 'lifted' to the second level.

Approach #2: coloring binary search-trees

We assume that trees are given as elements of the data type

$$\begin{array}{lcl} \textbf{type}\ \mathit{Level} & = & \mathit{Int} \\ \textbf{data}\ \mathit{Tree}\ a & = & \mathit{Empty} \mid \mathit{Node}\ \mathit{Level}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)\ \ , \end{array}$$

in which $\lceil \mathit{height}\ (\mathit{Node}\ h\ l\ a\ r) \,/\, 2 \rceil = h$.

Approach #2: coloring binary search-trees

```
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
```

Properties and cross connections

top-down **and** *bottom-up*: construct trees with a minimal number of red nodes among all trees of that size; depending on the number of elements they produce both trees of minimum and of maximum height.

bottom-up′: constructs trees of minimum height (optimal search-trees); relates red-black trees to *left-complete trees*.

rbtree: generalizes an algorithm for converting *AVL trees* into red-black trees; relates red-black trees to *half-balanced trees*; produces trees of minimum black height; applied to left-complete trees it constructs trees with a maximal number of red nodes among all trees of that size.

Résumé

- Solving the problem of constructing red-black trees gives many insights into their structure (and relates red-black trees to left-complete trees, half-balanced trees and AVL trees).

- Number systems are great for analysing and designing data structures and algorithms.

- Functional programming languages are an excellent tool for presenting algorithms and data structures.