CONSTRUCTING RED-BLACK TREES

(based on a talk presented at WAAAPL'99)

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`

Dezember, 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?

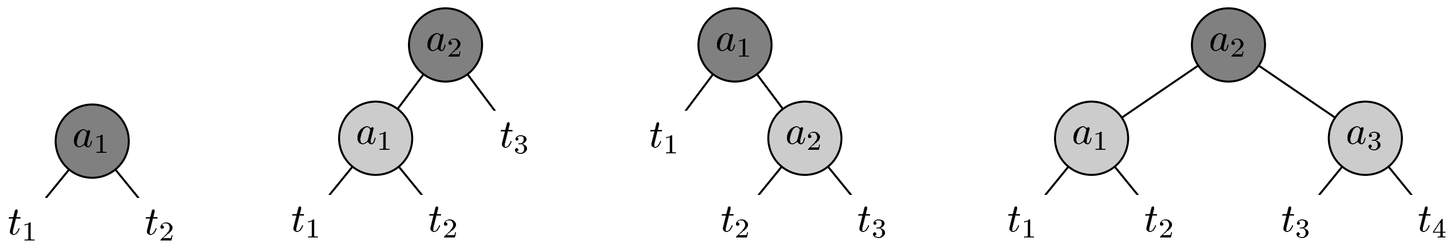
Chris Okasaki's functional red-black trees

A red-black tree is a binary tree whose nodes are colored either red or black (leaves are, by definition, black).

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

Historical roots

Red-black trees were developed by R. Bayer under the name *symmetric binary B-trees* as binary tree representations of *2-3-4 trees*. 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.



Balance conditions

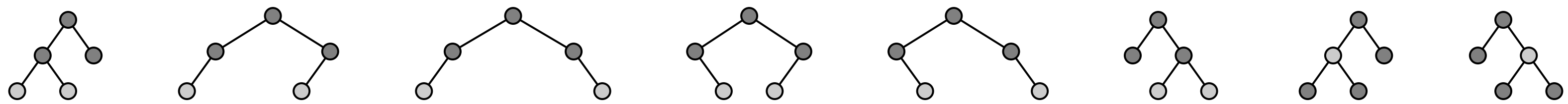
This explains the following two balance conditions.

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*).

Examples of red-black trees

There are 42 binary trees of size 5. Only 8 of them can be colored such that a red-black tree emerges.



Properties of red-black trees

The balance conditions imply the following properties (recall that $\sum_{k=0}^{n} x^k = (1 - x^{n+1})/(1 - x)$).

$$\begin{aligned} \textit{black-depth}\ t \leqslant \textit{depth}\ t &\leqslant 2\,\textit{black-depth}\ t \\ 2 \uparrow \textit{black-depth}\ t - 1 \leqslant \textit{size}\ t &\leqslant 4 \uparrow \textit{black-depth}\ t - 1 \\ \textit{depth}\ t &\leqslant 2\,\textit{lg}\ (\textit{size}\ t + 1) \end{aligned}$$

In other words, red-black trees guarantee $O(\log n)$ worst-case running time of basic dynamic-set operations.

Insertion

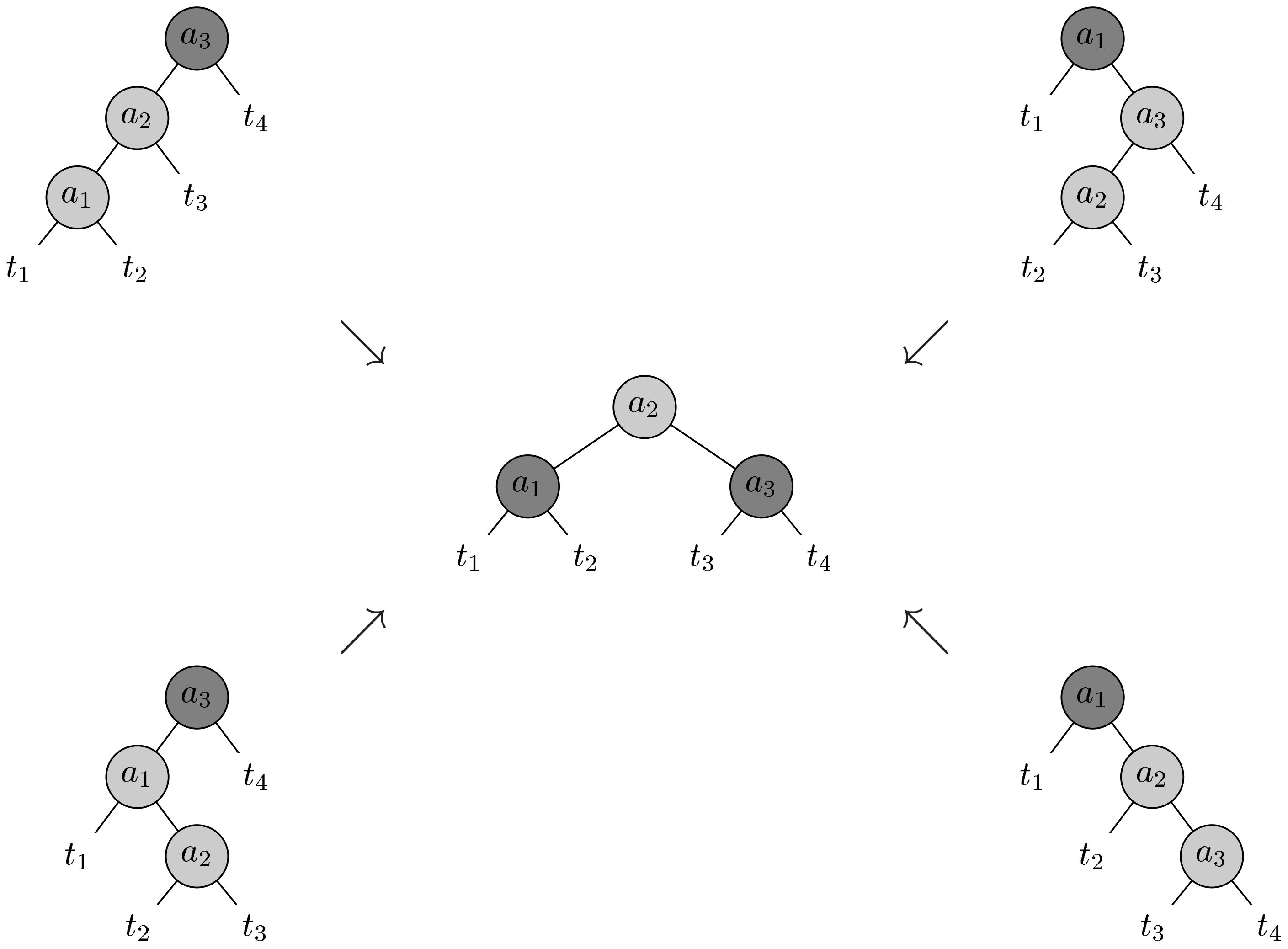
$$
\begin{array}{lcl}
\mathit{insert} & :: & (\mathit{Ord}\ a) \Rightarrow a \rightarrow \mathit{RBTree}\ a \rightarrow \mathit{RBTree}\ a \\
\mathit{insert}\ a\ t & = & \mathit{blacken}\ (\mathit{ins}\ t) \\
\quad\quad \mathbf{where} & & \\
\quad\quad \mathit{ins}\ E & = & N\ R\ E\ a\ E \\
\quad\quad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

NB. bal is a so-called *smart constructor*.

Balancing

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

$$
\begin{array}{lcl}
\multicolumn{3}{l}{bal\ B\ (N\ R\ (N\ R\ t_1\ a_1\ t_2)\ a_2\ t_3)\ a_3\ t_4} \\
 & = & N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
\multicolumn{3}{l}{bal\ B\ (N\ R\ t_1\ a_1\ (N\ R\ t_2\ a_2\ t_3))\ a_3\ t_4} \\
 & = & N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
\multicolumn{3}{l}{bal\ B\ t_1\ a_1\ (N\ R\ (N\ R\ t_2\ a_2\ t_3)\ a_3\ t_4)} \\
 & = & N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
\multicolumn{3}{l}{bal\ B\ t_1\ a_1\ (N\ R\ t_2\ a_2\ (N\ R\ t_3\ a_3\ t_4))} \\
 & = & N\ R\ (N\ B\ t_1\ a_1\ t_2)\ a_2\ (N\ B\ t_3\ a_3\ t_4) \\
bal\ c\ l\ a\ r & = & N\ c\ l\ a\ r
\end{array}
$$



a_3
a_2
t_4
a_1
t_3
t_1
t_2
a_1
t_1
a_3
a_2
t_4
t_2
t_3
a_2
a_1
a_3
t_1
t_2
t_3
t_4
a_3
a_1
t_4
t_1
a_2
t_2
t_3
a_1
t_1
a_2
t_2
a_3
t_3
t_4

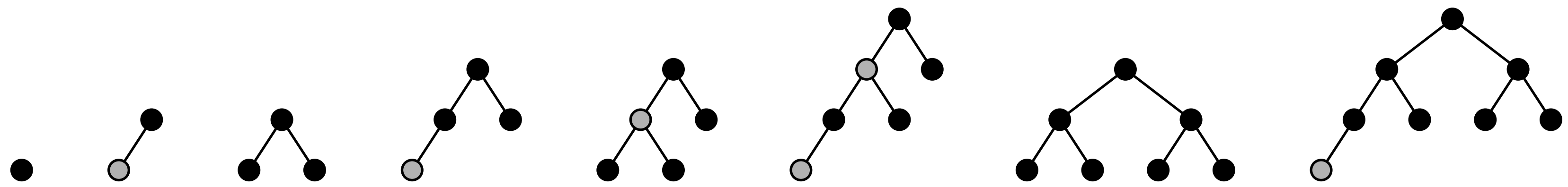
Constructing a red-black tree

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

NB. The elements are inserted from right to left.

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

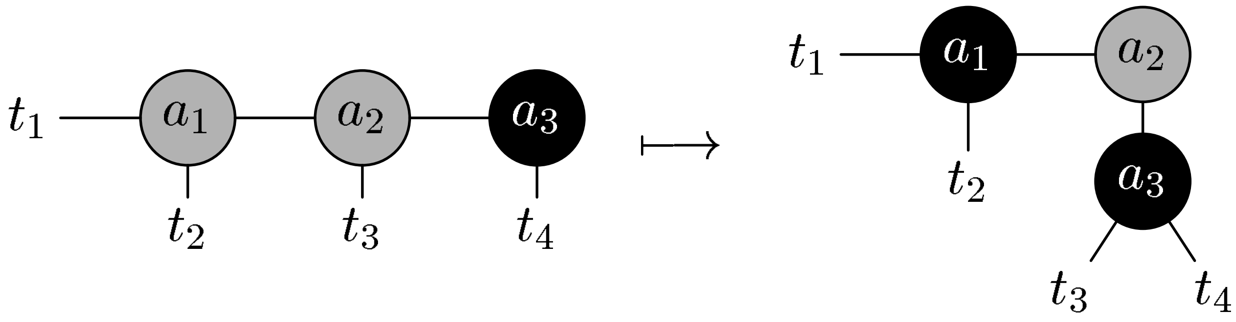
The following trees are generated by $top\text{-}down\,[1\,..\,i]$ for $1 \leqslant i \leqslant 8$ ('○' is a red node and '●' is a black node).



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

Approach #1: a closer look at *top-down*

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

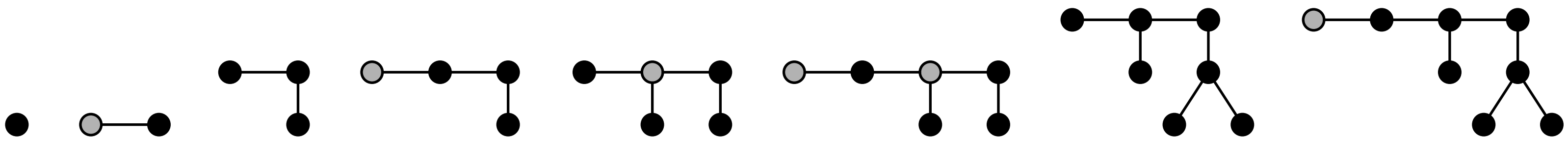


☞ The trees below the left spine (t_2, t_3 and t_4) must be perfectly balanced binary trees (perfect trees for short). Thus, the generated red-black trees correspond to sequences of *topped perfect trees* or *pennants*.

☞ 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.

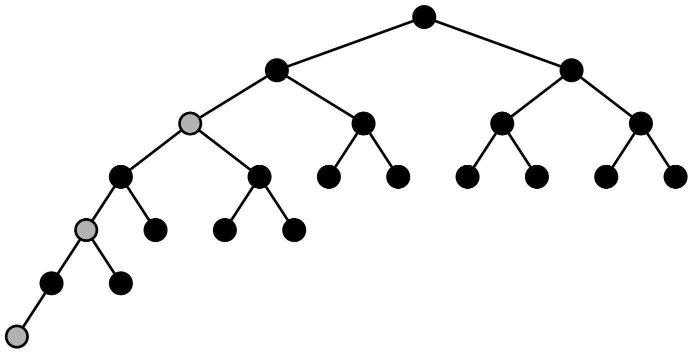
$$()_2, (1)_2, (2)_2, (11)_2, (12)_2, (21)_2, (22)_2, (111)_2, (112)_2 \ldots$$

Incrementing a 1-2 number:

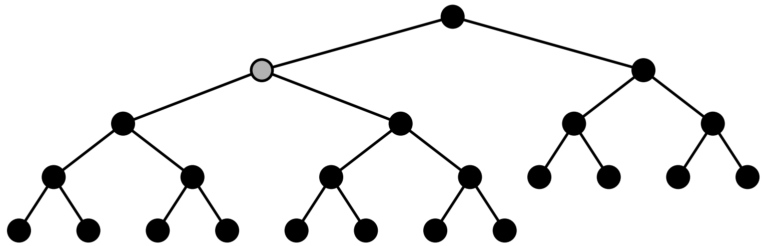
$$\begin{array}{lcl} \epsilon + 1 & = & 1 \\ s1 + 1 & = & s2 \\ s2 + 1 & = & (s+1)\,1 \quad . \end{array}$$

Approach #1: corollaries

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

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

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

$$
\begin{array}{lcl}
\mathit{incr}\ (\mathit{One}\ a\ t)\ [\,] & = & [\mathit{One}\ a\ t] \\
\mathit{incr}\ (\mathit{One}\ a_1\ t_1)\ (\mathit{One}\ a_2\ t_2 : \mathit{ps}) & = & \mathit{Two}\ a_1\ t_1\ a_2\ t_2 : \mathit{ps} \\
\multicolumn{3}{l}{\mathit{incr}\ (\mathit{One}\ a_1\ t_1)\ (\mathit{Two}\ a_2\ t_2\ a_3\ t_3 : \mathit{ps})} \\
\multicolumn{3}{l}{\quad =\ \ \mathit{One}\ a_1\ t_1 : \mathit{incr}\ (\mathit{One}\ a_2\ (N\ B\ t_2\ a_3\ t_3))\ \mathit{ps}}
\end{array}
$$

Approach #1: improving *top-down*

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

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

☞ A standard amortization argument shows that *bottom-up* runs in linear time.

☞ *top-down* and *bottom-up* construct trees with a minimal number of red nodes among all trees of that size.

Approach #1: less height, please!

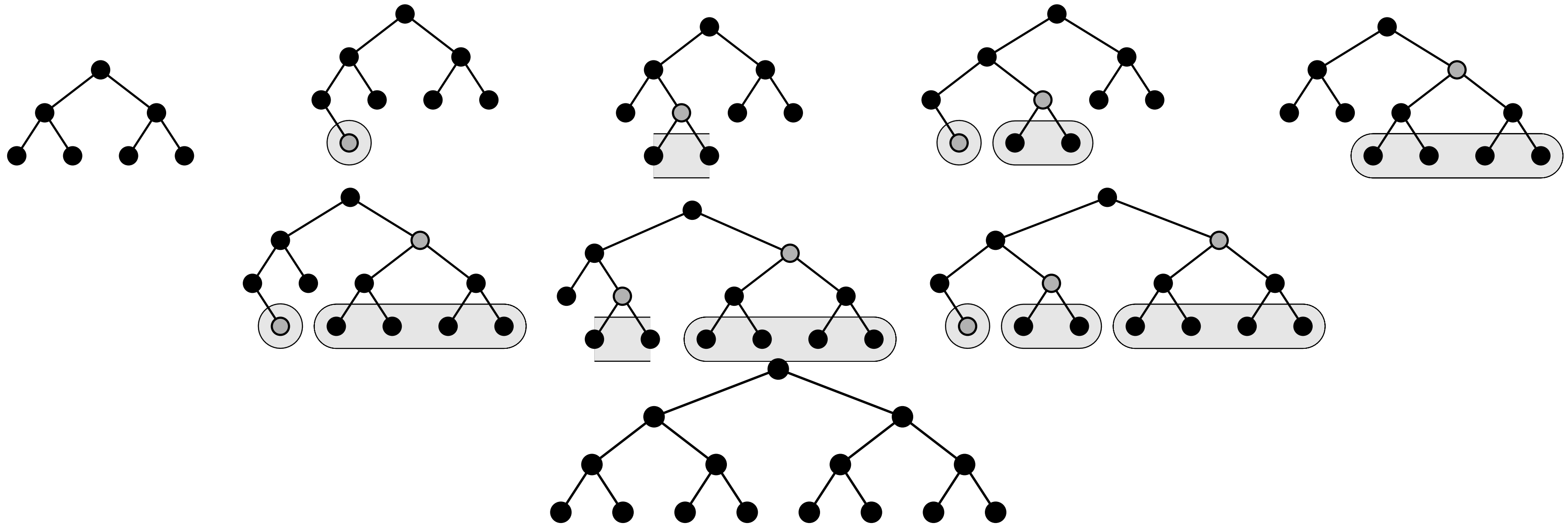
Interestingly, it is easy to modify $bottom\text{-}up$ so that it produces *optimal search-trees*. A simple *rotation to the right* suffices:

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

I call the resulting trees *quasi left-complete*.

Examples of quasi left-complete trees

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.



Digression: left-complete binary heaps

A *left-complete tree* has the leaves on the lowest level in the leftmost possible position (*heapsort*, for instance, uses an implicit representation of left-complete binary heaps).

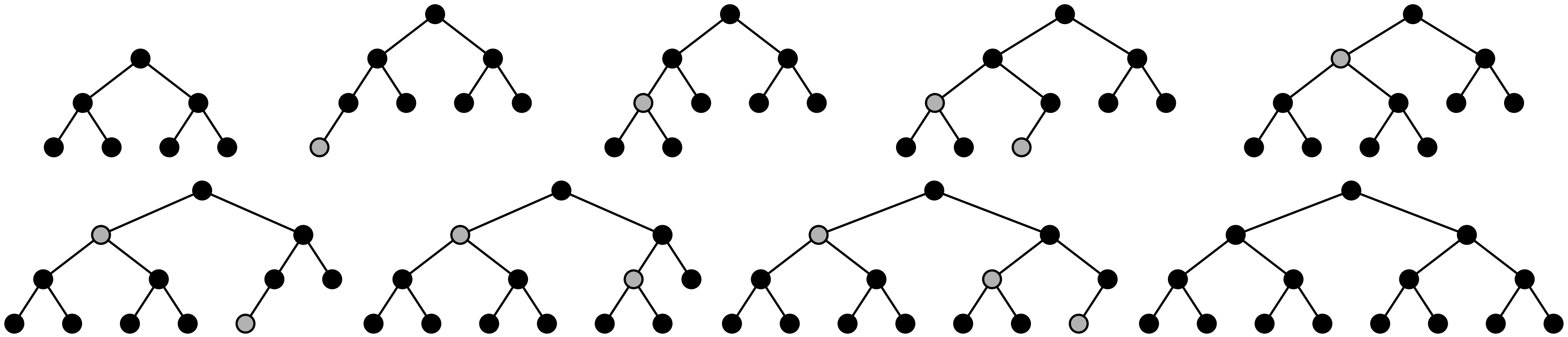
Interestingly, $link$ can be modified to produce left-complete trees.

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

☞ Since the children of the black node are swapped, the search-tree property is not preserved.

Examples of left-complete trees

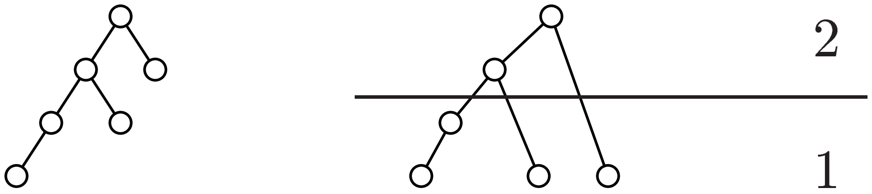
Here are the left-complete colleagues of the quasi left-complete trees.



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.

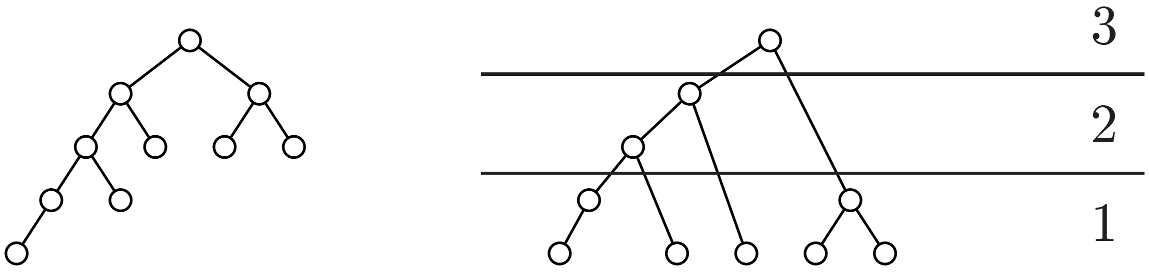


On the right-hand side the vertical position of a node corresponds to its height rather than its distance from the root. We assign a *level number* to each node: a node of height h receives the level number $\lceil h \,/\, 2 \rceil$.

☞ 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

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

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

Approach #2: coloring binary search-trees

$$
\begin{array}{lcl}
\mathit{rbtree} & :: & \mathit{Tree}\ a \to \mathit{RBTree}\ a \\
\mathit{rbtree}\ \mathit{Empty} & = & E \\
\mathit{rbtree}\ (\mathit{Node}\ h\ l\ a\ r) & = & N\ B\ (\mathit{rbtree}'\ h\ l)\ a\ (\mathit{rbtree}'\ h\ r) \\
\mathit{rbtree}' & :: & \mathit{Level} \to \mathit{Tree}\ a \to \mathit{RBTree}\ a \\
\mathit{rbtree}'\ \mathit{hp}\ \mathit{Empty} & & \\
\qquad \mid \mathit{hp} == 1 & = & E \\
\qquad \mid \mathit{otherwise} & = & \mathit{error}\ \texttt{"not a red-black tree"} \\
\mathit{rbtree}'\ \mathit{hp}\ (\mathit{Node}\ h\ l\ a\ r) & = & N\ \mathit{color}\ (\mathit{rbtree}'\ h'\ l)\ a\ (\mathit{rbtree}'\ h'\ r) \\
\qquad \mathbf{where} & & \\
\qquad h' & = & h\ \text{`}\mathit{max}\text{`}\ (\mathit{hp} - 1) \\
\qquad \mathit{color} \mid \mathit{hp} == h & = & R \\
\qquad\qquad \mid \mathit{otherwise} & = & B
\end{array}
$$

Approach #2: correctness of $rbtree$

It is relatively easy to see that $rbtree$ yields a valid red-black tree. The converse can be shown using an alternative characterization of red-black trees.

A binary tree t is said to be *half-balanced* if for every subtree u of t,

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

Every red-black tree is half-balanced because $\mathit{height}\ t \leqslant 2\,\mathit{black\text{-}height}$ and $\mathit{black\text{-}height}\ t \leqslant \mathit{min\text{-}height}\ t$. The function $rbtree$ can be viewed as a constructive proof of the reverse implication.

Approach #2: properties of $rbtree$

Historical note: $rbtree$ generalizes an algorithm for converting *AVL trees* into red-black trees.

☞ $rbtree$ produces trees of minimum black height.

☞ If 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.

Appendix: foldr

The function foldr captures a common pattern of recursion on lists (it is a so-called *catamorphism*; the greak preposition $\kappa\alpha\tau\alpha$ means "downwards").

$$
\boxed{
\begin{array}{lcl}
\mathit{foldr} & :: & (a \to b \to b) \to b \to ([\,a\,] \to b) \\
\mathit{foldr}\ (\star)\ e\ [\,] & = & e \\
\mathit{foldr}\ (\star)\ e\ (a : as) & = & a \star \mathit{foldr}\ (\star)\ e\ as
\end{array}
}
$$

For example,

$$
\mathit{foldr}\ (\star)\ e\ (a_1 : a_2 : \cdots : a_n : [\,]) \quad = \quad a_1 \star (a_2 \star (\cdots \star (a_n \star e)\ \cdots)) \ .
$$

Appendix: $foldl$

The function $foldl$ is similar to $foldr$, except that the parentheses group from the left.

$$
\begin{array}{lcl}
foldl & :: & (b \to a \to b) \to b \to ([\,a\,] \to b) \\
foldl\ (\star)\ e\ [\,] & = & e \\
foldl\ (\star)\ e\ (a : as) & = & foldl\ (\star)\ (e \star a)\ as
\end{array}
$$

For example,

$$
foldl\ (\star)\ e\ (a_1 : a_2 : \cdots : a_n : [\,]) \quad = \quad (\cdots\ ((e \star a_1) \star a_2) \star \cdots) \star a_n \quad .
$$