

# FUNCTIONAL PEARL

## *Purely Functional 1-2 Brother Trees*

RALF HINZE

*Computing Laboratory, University of Oxford, Wolfson Building, Parks Road, Oxford OX1 3QD, England*  
(e-mail: ralf.hinze@comlab.ox.ac.uk)

### 1 Prologue

Enter the computing arboretum and you will find a variety of well-studied trees: AVL trees (Adel'son-Vel'skiĭ & Landis 1962), symmetric binary B-trees (Bayer 1972), Hopcroft's 2-3 trees (Aho *et al.* 1974), the bushy finger trees (Guibas *et al.* 1977) and the colourful red-black trees (Guibas & Sedgewick 1978). In this pearl, we look at a more exotic species of balanced search trees, 1-2 brother trees (Ottmann *et al.* 1979), which deserves to be better known. Brother trees lend themselves well to a functional implementation with deletion (Section 5) as straightforward as insertion (Section 3), both running in logarithmic time. Furthermore, brother trees can be constructed from ordered lists in linear time (Section 4). With some simple optimisations in place, this implementation of search trees is one of the fastest around. So, fasten your seat belts.

### 2 Brother trees

A 1-2 brother<sup>1</sup> tree, brother tree for short, consists of nullary, unary and binary nodes.

**data**  $Tree\ a = N_0 \mid N_1\ (Tree\ a) \mid N_2\ (Tree\ a)\ a\ (Tree\ a)$

An element of type *Tree* *t* is called a *brother tree* iff (a) all nullary nodes have the same depth (*height condition*) and (b) each unary node has a binary brother (*brother condition*).

The brother condition implies that the root of a brother tree is not unary and that a unary node has not a unary child. Put positively, a unary node only occurs as the child of a binary node. We can formalise the invariants of brother trees using *subset types*.

$$\mathcal{B}_0\ a = N_0$$

$$\mathcal{B}_{h+1}\ a = N_2\ (\mathcal{U}_h\ a \cup \mathcal{B}_h\ a)\ a\ (\mathcal{B}_h\ a) \cup N_2\ (\mathcal{B}_h\ a)\ a\ (\mathcal{U}_h\ a \cup \mathcal{B}_h\ a)$$

$$\mathcal{U}_{h+1}\ a = N_1\ (\mathcal{B}_h\ a)$$

The definitions lean on the syntax of datatype declarations with  $C\ \mathcal{A}_1\ \dots\ \mathcal{A}_n$  abbreviating the set comprehension  $\{C\ a_1\ \dots\ a_n \mid a_1 \in \mathcal{A}_1, \dots, a_n \in \mathcal{A}_n\}$ .

<sup>1</sup> I decided to stick to the original terminology, even though it is not gender neutral.

Table 1. Number of brother trees of height  $0 \leq h \leq 5$  and size  $0 \leq s \leq 15$

		Size $s$															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Height $h$	0	1															
	1		1														
	2			2	1												
	3					4	6	4	1								
	4								16	32	44	60	70	56	28	8	1
	5													128	448	864	1552

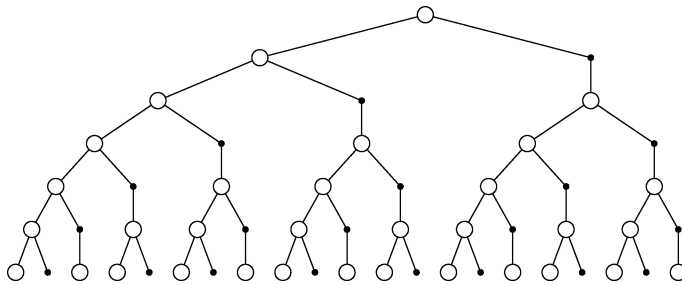


Fig. 1. Fibonacci tree of height seven, *fib-tree 7*.

A brother tree of height  $h$  with labels of type  $t$  is then an element of  $\mathcal{B}_h t \subseteq Tree t$ .

To give a feel for the restrictiveness of the conditions, Table 1 lists the number of differently shaped brother trees for a few given heights and sizes. For instance, there are 1,553 brother trees of size 15, none of which is deeper than 5. By contrast, the total number of binary trees of that size amounts to 9,694,845, with heights ranging from 4 to 15.

The sparsest brother tree of a given height is the *Fibonacci tree* defined

$$\begin{aligned}
 fib\text{-}tree &:: Integer \rightarrow Tree () \\
 fib\text{-}tree\ 0 &= N_0 \\
 fib\text{-}tree\ 1 &= N_2\ N_0\ ()\ N_0 \\
 fib\text{-}tree\ (h + 2) &= N_2\ (fib\text{-}tree\ (h + 1))\ ()\ (N_1\ (fib\text{-}tree\ h))\ .
 \end{aligned}$$

Figure 1 displays the Fibonacci tree of height seven. Since unary nodes contain no elements, they are drawn as small, filled circles. For the example tree, the ratio between binary and unary nodes is  $(F_9 - 1)/(F_8 - 1) = 33/20 = 1.65$ , where  $F_n$  is the  $n$ th Fibonacci number. As the height goes to infinity, the ratio  $(F_{h+2} - 1)/(F_{h+1} - 1)$  approaches the golden ratio,  $\phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1.618$ . Since  $F_{h+2} - 1$  is the minimum possible size of a brother tree of height  $h$ , a brother tree with  $n$  elements has height at most  $\lg(n + 1)/\lg \phi \approx 1.44 \lg(n + 1)$ .

If we remove the unary nodes from a brother tree, contracting  $N_1 t$  to  $t$ , we obtain an AVL tree of the same height! The height and the brother condition translate to the balance condition of AVL trees: for each node, the height difference of the

children is at most one. Conversely, we can transform an AVL tree to a brother tree by inserting unary nodes at the appropriate places, so that all paths from the root to a leaf are equally long.

The standard query operations on binary search trees are easy to adapt for brother trees. As an example, here is the definition of membership.

$$\begin{aligned}
 \text{member} &:: (\text{Ord } a) \Rightarrow a \rightarrow \text{Tree } a \rightarrow \text{Bool} \\
 \text{member } a \ N_0 &= \text{False} \\
 \text{member } a \ (N_1 \ t) &= \text{member } a \ t \\
 \text{member } a \ (N_2 \ l \ b \ r) \mid a \leq b &= \text{member } a \ l \\
 &\mid a > b = \text{member } a \ r
 \end{aligned}$$

### 3 Insertion

Since brother trees are in a one-to-one correspondence to AVL trees, we could adapt AVL insertion and deletion to the new setting. However, and perhaps surprisingly, if one starts afresh, two new algorithms emerge.

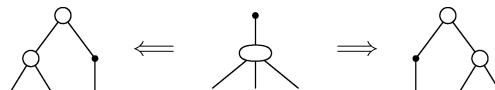
Insertion consists of two phases: a top-down search and a bottom-up construction phase. For the first phase, we use the standard algorithm for binary search trees. During the second phase, we additionally restore the invariants of brother trees using smart constructors.

$$\begin{aligned}
 \text{insert} &:: (\text{Ord } a) \Rightarrow a \rightarrow \text{Tree } a \rightarrow \text{Tree } a \\
 \text{insert } a \ t &= \text{root } (\text{ins } t) \\
 \text{where} & \\
 \text{ins } N_0 &= L_2 \ a \\
 \text{ins } (N_1 \ t) &= n_1 \ (\text{ins } t) \\
 \text{ins } (N_2 \ l \ b \ r) \mid a \leq b &= n_2 \ (\text{ins } l) \ b \ r \\
 &\mid a > b = n_2 \ l \ b \ (\text{ins } r)
 \end{aligned}$$

The helper function *ins* recurses from the root to a leaf. In the base case, the nullary constructor  $N_0$  is replaced by the leaf  $L_2 \ a$ , where  $L_2$  is a new, auxiliary data constructor. The functions  $n_1$  and  $n_2$  are smart versions of the constructors  $N_1$  and  $N_2$ , which among other things eliminate occurrences of the new constructor. This is actually quite simple. If the new element is inserted into a unary node, it is expanded to a binary node. By the same logic, a binary node is expanded to a ternary node. Like  $L_2$ , a ternary node is an auxiliary data constructor introduced solely for the purpose of insertion.

$$\text{data Tree } a = \dots \mid L_2 \ a \mid N_3 \ (\text{Tree } a) \ a \ (\text{Tree } a) \ a \ (\text{Tree } a)$$

All that is left to do is to get rid of the ternary node. If the sole son of a unary node is ternary, then we can rearrange the tree as follows.

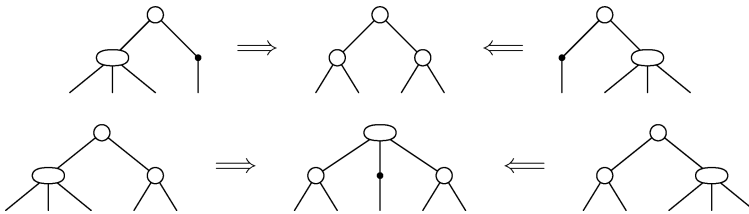


Both transformations are viable. In the code below, we arbitrarily pick the first alternative.

$$\begin{aligned}
 \text{root } (L_2 a) &= N_2 N_0 a N_0 \\
 \text{root } (N_3 t_1 a_1 t_2 a_2 t_3) &= N_2 (N_2 t_1 a_1 t_2) a_2 (N_1 t_3) \\
 \text{root } t &= t \\
 n_1 (L_2 a) &= N_2 N_0 a N_0 \\
 n_1 (N_3 t_1 a_1 t_2 a_2 t_3) &= N_2 (N_2 t_1 a_1 t_2) a_2 (N_1 t_3) \\
 n_1 t &= N_1 t
 \end{aligned}$$

The function *root* ensures that the auxiliary constructors are eliminated if they propagate to the root. If one of the first two equations matches, then we know that the tree has grown – like most height-balanced trees, brother trees grow upwards.

For a binary node, we additionally distinguish whether the brother of the ternary node is unary or binary.



A ternary and a unary node are transformed into two binary ones. If the ternary node has a binary brother, we propagate the ternary node upwards. The transformations are implemented by the code below – subtrees are re-used with the help of as-patterns.

$$\begin{aligned}
 n_2 (L_2 a_1) \quad a_2 t_1 &= N_3 N_0 a_1 N_0 a_2 t_1 \quad -- t_1 == N_0 \\
 n_2 (N_3 t_1 a_1 t_2 a_2 t_3) a_3 (N_1 t_4) &= N_2 (N_2 t_1 a_1 t_2) a_2 (N_2 t_3 a_3 t_4) \\
 n_2 (N_3 t_1 a_1 t_2 a_2 t_3) a_3 t_4@(N_2 - - -) &= N_3 (N_2 t_1 a_1 t_2) a_2 (N_1 t_3) a_3 t_4 \\
 n_2 t_1 \quad a_1 (L_2 a_2) &= N_3 t_1 a_1 N_0 a_2 N_0 \quad -- t_1 == N_0 \\
 n_2 (N_1 t_1) \quad a_1 (N_3 t_2 a_2 t_3 a_3 t_4) &= N_2 (N_2 t_1 a_1 t_2) a_2 (N_2 t_3 a_3 t_4) \\
 n_2 t_1@(N_2 - - -) a_1 (N_3 t_2 a_2 t_3 a_3 t_4) &= N_3 t_1 a_1 (N_1 t_2) a_2 (N_2 t_3 a_3 t_4) \\
 n_2 t_1 a_1 t_2 &= N_2 t_1 a_1 t_2
 \end{aligned}$$

Clearly, the smart constructors *root*, *n<sub>1</sub>* and *n<sub>2</sub>* jointly eliminate the auxiliary nodes *L<sub>2</sub>* and *N<sub>3</sub>*. But, is the result still a brother tree? It is easy to check that the transformations preserve the height – like *N<sub>0</sub>*, the height of *L<sub>2</sub> a* is by definition 0. Regarding the brother condition, note that a ternary node is either of the form *N<sub>3</sub> N<sub>0</sub> a<sub>1</sub> N<sub>0</sub> a<sub>2</sub> N<sub>0</sub>* or of the form *N<sub>3</sub> (N<sub>2</sub> t<sub>1</sub> a<sub>1</sub> t<sub>2</sub>) a<sub>2</sub> (N<sub>1</sub> t<sub>3</sub>) a<sub>3</sub> (N<sub>2</sub> t<sub>4</sub> a<sub>4</sub> t<sub>5</sub>)*. This invariant guarantees that the son of a freshly constructed unary node is never unary and that a freshly constructed binary node has at most one unary son. The invariants can be captured using subset types.

$$\begin{aligned}
 \mathcal{B}_0^+ \quad a &= \mathcal{B}_0 \quad a \cup L_2 a \\
 \mathcal{B}_1^+ \quad a &= \mathcal{B}_1 \quad a \cup N_3 N_0 a N_0 a N_0 \\
 \mathcal{B}_{h+2}^+ \quad a &= \mathcal{B}_{h+2} a \cup N_3 (\mathcal{B}_{h+1} a) a (\mathcal{U}_{h+1} a) a (\mathcal{B}_{h+1} a)
 \end{aligned}$$

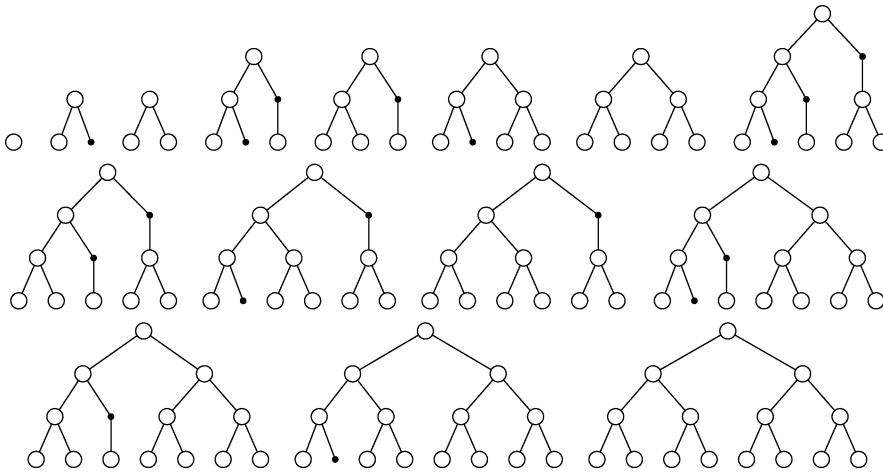


Fig. 2. Brother trees generated by *from-list* [1..n] for  $n = 1, \dots, 15$ .

The set  $\mathcal{B}_h^+ t$  comprises *grown trees*, which possibly have an auxiliary node as their root. It is important to note that the auxiliary nodes only appear on the top level, never below a root node. The functions involved in inserting an element then satisfy the following invariants, where  $f \in P \rightarrow Q$  means that  $\forall x . x \in P \implies f x \in Q$ .

$$\begin{aligned}
 \text{ins} &\in \mathcal{B}_h a \rightarrow \mathcal{B}_h^+ a \\
 \text{ins} &\in \mathcal{U}_h a \rightarrow (\mathcal{U}_h a \cup \mathcal{B}_h a) \\
 n_1 &\in \mathcal{B}_h^+ a \rightarrow (\mathcal{U}_{h+1} a \cup \mathcal{B}_{h+1} a) \\
 n_2 &\in \mathcal{B}_h^+ a \rightarrow a \rightarrow (\mathcal{U}_h a \cup \mathcal{B}_h a) \rightarrow \mathcal{B}_{h+1}^+ a \\
 n_2 &\in (\mathcal{U}_h a \cup \mathcal{B}_h a) \rightarrow a \rightarrow \mathcal{B}_h^+ a \rightarrow \mathcal{B}_{h+1}^+ a \\
 \text{root} &\in \mathcal{B}_h^+ a \rightarrow (\mathcal{B}_h a \cup \mathcal{B}_{h+1} a)
 \end{aligned}$$

Note that *ins* preserves the height,  $n_1$  and  $n_2$  increase it and *root* possibly increases it. The smart constructor  $n_2$  is really two-in-one, as it takes care of growth in either the left or the right subtree.

Finally, all the transformations preserve the search-tree property: the relative order and multiplicity of elements and subtrees are unchanged.

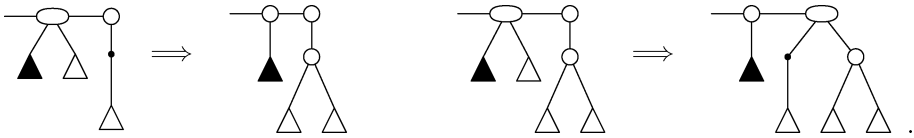
### 4 Construction

Using *insert* we can easily construct a brother tree from an *unordered list*.

$$\begin{aligned}
 \text{from-list} &:: (\text{Ord } a) \Rightarrow [a] \rightarrow \text{Tree } a \\
 \text{from-list} &= \text{foldr insert } N_0
 \end{aligned}$$

Figure 2 displays the trees generated by *from-list* [1..n] for  $n = 1, \dots, 15$ . Note that we do not label the nodes as the keys are uniquely determined by the search-tree property. Rather amazingly, if  $n$  is  $2^i - 1$  for some  $i$ , we obtain a perfectly balanced binary tree, perfect tree for short. Moreover, unary nodes are quite rare. In fact, they only appear immediately below the left spine of a tree. This is not a coincidence. Since the list is processed from right to left, the elements are actually inserted in

descending order. Consequently, *ins* always traverses the left spine of the tree to the leftmost leaf. Since furthermore the left spine contains only binary nodes, only the first three equations of the smart constructor  $n_2$  can possibly match. Drawing the left spine horizontally, the relevant transformations are



The transformations preserve the following invariant: the right son of a binary node is either a perfect tree or a unary node applied to a perfect tree ( $\blacktriangle$ ); the same holds for the middle son of a ternary node, whereas its right son is always a perfect tree ( $\triangle$ ).

The transformations along the left spine are reminiscent of the binary increment with the ternary node corresponding to a cascading carry. In fact, the construction of a brother tree from an *ordered list* can be modelled after a funny variant of the binary number system that uses the digits  $\frac{1}{2}$  and 1. Why these two digits? Well, the  $i$ th tree on the left spine has either  $\frac{1}{2} \cdot 2^i$  or  $1 \cdot 2^i$  elements, including the element on the spine. Recall that the value of the binary number  $b_0 \dots b_{n-1}$  is  $\sum_{i=0}^{n-1} b_i 2^i$ . For our number system, we constrain the digits to  $b_0 = 1$  and  $b_{i+1} \in \{\frac{1}{2}, 1\}$ . The binary increment is then given by  $1 + \varepsilon = 1, 1 + 1s = 1(\frac{1}{2} + s)$  and  $\frac{1}{2} + \varepsilon = \frac{1}{2}, \frac{1}{2} + \frac{1}{2}s = 1s, \frac{1}{2} + 1s = \frac{1}{2}(\frac{1}{2} + s)$ . Thus, the first eight positive numbers are

$$1, 1\frac{1}{2}, 11, 1\frac{1}{2}\frac{1}{2}, 11\frac{1}{2}, 1\frac{1}{2}1, 111, 1\frac{1}{2}\frac{1}{2}\frac{1}{2} .$$

These numbers correspond to the trees in the first row of Figure 2. We can use this correspondence to improve the running time of *from-list* from  $\Theta(n \log n)$  to  $\Theta(n)$  for the special case that the input list is ordered.

First, we define a suitable representation for the left spine of a brother tree.

**data** *Spine*  $a = Nil \mid Half\ a\ (Tree\ a)\ (Spine\ a) \mid Full\ a\ (Tree\ a)\ (Spine\ a)$

The constructor *Half* corresponds to the digit  $\frac{1}{2}$ , the constructor *Full* to 1. Consing an element to the spine is modelled after the binary increment: *cons a s* implements  $1 + s$  and *half a t s* implements  $\frac{1}{2} + s$ .

```

cons :: a -> Spine a -> Spine a
cons a1 Nil           = Full a1 N0 Nil
cons a1 (Full a2 t2 s) = Full a1 N0 (half a2 t2 s)
half :: a -> Tree a -> Spine a -> Spine a
half a1 t1 Nil       = Half a1 t1 Nil
half a1 t1 (Half a2 t2 s) = Full a1 (N2 t1 a2 t2) s
half a1 t1 (Full a2 t2 s) = Half a1 t1 (half a2 t2 s)
    
```

The new construction function *from-ord-list* first transforms the input list to a spine and then converts the spine to a brother tree.

$$\begin{aligned} \text{from-ord-list} &:: [a] \rightarrow \text{Tree } a \\ \text{from-ord-list} &= \text{from-spine } N_0 \cdot \text{foldr } \text{cons } \text{Nil} \\ \text{from-spine} &:: \text{Tree } a \rightarrow \text{Spine } a \rightarrow \text{Tree } a \\ \text{from-spine } t_1 \text{ Nil} &= t_1 \\ \text{from-spine } t_1 (\text{Half } a_1 t_2 s) &= \text{from-spine } (N_2 t_1 a_1 (N_1 t_2)) s \\ \text{from-spine } t_1 (\text{Full } a_1 t_2 s) &= \text{from-spine } (N_2 t_1 a_1 t_2 \quad ) s \end{aligned}$$

Since *cons* has a constant amortised running time, *from-ord-list* works in linear time. As an aside, note that the functions above are truly polymorphic. In particular, *from-ord-list* does not require an *Ord a* context since we assume that the input is given in ascending order.

## 5 Deletion

Deletion is typically more involved than insertion. One reason is that insertion adds the new element to the fringe of the tree, whereas deletion removes the element from an arbitrary node, not necessarily a leaf. Second, with the notable exception of AVL trees, re-balancing seems to be more intricate for deletion. In the case of red-black trees, for instance, there is an elegant functional insertion algorithm (Okasaki 1999) that simplifies the complex imperative original (Guibas & Sedgewick 1978). However, for deletion no such improvement is known. In the case of brother trees, the situation is almost reversed. For a start, we do not need any auxiliary data constructors: if an element is deleted from a binary node, it is contracted to a unary node. Like insertion, deletion is a two-phase algorithm.

$$\begin{aligned} \text{delete} &:: (\text{Ord } a) \Rightarrow a \rightarrow \text{Tree } a \rightarrow \text{Tree } a \\ \text{delete } a \ t &= \text{root } (\text{del } t) \\ \text{where} & \\ \text{del } N_0 &= N_0 \\ \text{del } (N_1 \ t) &= N_1 (\text{del } t) \\ \text{del } (N_2 \ l \ b \ r) & \mid a < b = n_2 (\text{del } l) \ b \ r \\ & \mid a == b = \text{case } \text{split-min } r \ \text{of } \text{Nothing} \rightarrow N_1 \ l \\ & \quad \text{Just } (a', r') \rightarrow n_2 \ l \ a' \ r' \\ & \mid a > b = n_2 \ l \ b \ (\text{del } r) \end{aligned}$$

If the to-be-deleted element is found, it is replaced by its inorder successor, if any.

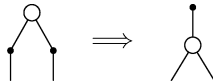
$$\begin{aligned} \text{split-min } N_0 &= \text{Nothing} \\ \text{split-min } (N_1 \ t) &= \text{case } \text{split-min } t \ \text{of } \text{Nothing} \rightarrow \text{Nothing} \\ & \quad \text{Just } (a, t') \rightarrow \text{Just } (a, N_1 \ t') \\ \text{split-min } (N_2 \ t_1 \ a_1 \ t_2) &= \text{case } \text{split-min } t_1 \ \text{of } \text{Nothing} \rightarrow \text{Just } (a_1, N_1 \ t_2) \\ & \quad \text{Just } (a, t'_1) \rightarrow \text{Just } (a, n_2 \ t'_1 \ a_1 \ t_2) \end{aligned}$$

As before,  $n_2$  is a smart constructor that locally detects and repairs violations of the invariants with  $root$  finalising the process.

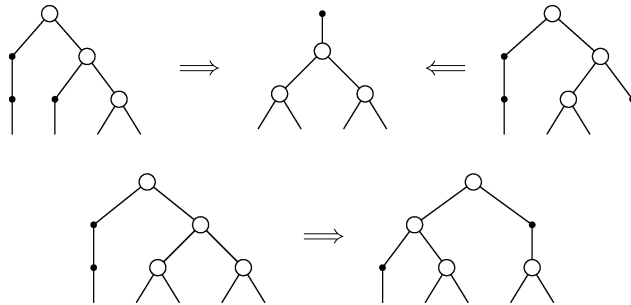
$$\begin{aligned} root (N_1 t) &= t \\ root t &= t \end{aligned}$$

If the first equation matches, the tree has shrunk.

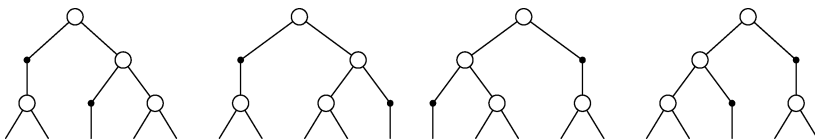
Now, since  $del$  or  $split-min$  replaces a binary node by a unary one, the brother condition is possibly violated: a unary node may have a unary brother or it may not have a brother at all. The first defect is easy to remedy.



If a unary node has a unary son, we have to include its binary father in our considerations. Let us assume that it is the left subtree of the father that violates the brother condition – the symmetric case is handled, well, symmetrically. Since the right subtree must be binary, there are three sub-cases to consider.



For each sub-case, the resulting tree is inevitable; there is no other choice. If the right subtree contains a unary node, the height condition completely determines the shape of the tree: no other tree of height 3 has three binary nodes. If the right subtree consists of three binary nodes, the height condition leaves us with four choices.



There are no other trees of height 3 with four binary nodes. However, all choices, with the notable exception of the third, possibly violate the brother condition since the unary node on the lowest level possibly has a unary son. The third alternative, on the other hand, is a valid brother tree because it re-uses the subtrees from the original tree. Only the two upper levels are changed using a ‘left rotation’. Again, it is easy to see that the transformations preserve the height. In the code below,



subtrees are re-used with the help of as-patterns.

$$\begin{aligned}
 n_2 (N_1 t_1) a_1 (N_1 t_2) &= N_1 (N_2 t_1 a_1 t_2) \\
 n_2 (N_1 (N_1 t_1)) \quad a_1 (N_2 (N_1 t_2) \quad a_2 t_3 @ (N_2 \_ \_ \_)) &= N_1 (N_2 (N_2 t_1 a_1 t_2) a_2 t_3) \\
 n_2 (N_1 (N_1 t_1)) \quad a_1 (N_2 (N_2 t_2 a_2 t_3) \quad a_3 (N_1 t_4)) &= N_1 (N_2 (N_2 t_1 a_1 t_2) a_2 \\
 &\quad (N_2 t_3 a_3 t_4)) \\
 n_2 (N_1 t_1 @ (N_1 \_)) \quad a_1 (N_2 t_2 @ (N_2 \_ \_ \_)) a_2 t_3 @ (N_2 \_ \_ \_) &= N_2 (N_2 t_1 a_1 t_2) a_2 (N_1 t_3) \\
 n_2 (N_2 (N_1 t_1) \quad a_1 (N_2 t_2 a_2 t_3)) \quad a_3 (N_1 (N_1 t_4)) &= N_1 (N_2 (N_2 t_1 a_1 t_2) a_2 \\
 &\quad (N_2 t_3 a_3 t_4)) \\
 n_2 (N_2 t_1 @ (N_2 \_ \_ \_)) a_1 (N_1 t_2)) \quad a_2 (N_1 (N_1 t_3)) &= N_1 (N_2 t_1 a_1 (N_2 t_2 a_2 t_3)) \\
 n_2 (N_2 t_1 @ (N_2 \_ \_ \_)) a_1 t_2 @ (N_2 \_ \_ \_) \quad a_2 (N_1 t_3 @ (N_1 \_)) &= N_2 (N_1 t_1) a_1 (N_2 t_2 a_2 t_3) \\
 n_2 t_1 a_1 t_2 &= N_2 t_1 a_1 t_2
 \end{aligned}$$

Turning to formal treatment, we first introduce subset types that capture the notion of a *shrunk tree*.

$$\begin{aligned}
 \mathcal{B}_h^- \quad a &= \mathcal{B}_h a \cup \mathcal{U}_h a \\
 \mathcal{U}_{h+1}^- \quad a &= N_1 (\mathcal{B}_h^- a)
 \end{aligned}$$

The functions involved in deleting an element then satisfy the following invariants.

$$\begin{aligned}
 del &\in \mathcal{B}_h a \rightarrow \mathcal{B}_h^- a \\
 del &\in \mathcal{U}_h a \rightarrow \mathcal{U}_h^- a \\
 split-min &\in \mathcal{B}_h a \rightarrow Maybe (a, \mathcal{B}_h^- a) \\
 split-min &\in \mathcal{U}_h a \rightarrow Maybe (a, \mathcal{U}_h^- a) \\
 n_2 &\in \mathcal{U}_h^- a \rightarrow a \rightarrow \mathcal{B}_h a \rightarrow \mathcal{B}_{h+1}^- a \\
 n_2 &\in \mathcal{B}_h a \rightarrow a \rightarrow \mathcal{U}_h^- a \rightarrow \mathcal{B}_{h+1}^- a \\
 n_2 &\in \mathcal{B}_h^- a \rightarrow a \rightarrow \mathcal{B}_h^- a \rightarrow \mathcal{B}_{h+1}^- a \\
 root &\in \mathcal{B}_0^- a \rightarrow \mathcal{B}_0 a \\
 root &\in \mathcal{B}_{h+1}^- a \rightarrow (\mathcal{B}_{h+1} a \cup \mathcal{B}_h a)
 \end{aligned}$$

Note that *del* and *split-min* preserve the height, *n<sub>2</sub>* increases it and *root* possibly decreases it.

While the definition of re-balancing is inevitable, *delete* can alternatively be defined in terms of an operation that appends, or rather, zips two brother trees of height *h* forming a brother tree of height *h + 1*. The details are left as an exercise to the reader.

## 6 Epilogue

Brother trees lend themselves well to a functional implementation. In particular, the re-balancing operations are nicely captured by equational rewrite rules. While insertion and deletion are adaptations of imperative algorithms, the construction of brother trees appears to be original. A similar approach also works for red-black trees (Hinze 1999).

Some simple, but effective optimisations suggest themselves. Since all leaves have the same depth, we can eliminate nullary nodes by specialising the nodes on the penultimate level: *N<sub>1</sub> N<sub>0</sub>* becomes *L<sub>1</sub>* and *N<sub>1</sub> N<sub>0</sub> a<sub>1</sub> N<sub>0</sub>* becomes *L<sub>2</sub> a<sub>1</sub>*. Alternatively

Table 2. Comparison of red-black trees and 1-2 brother trees. The programs were compiled using `ghc-6.8.2 -O2`. The running time is given in seconds, minimum of three runs, as reported by `ghc`'s run-time system. All measurements were taken on an unloaded machine, AMD Athlon 64 X2 Dual Core Processor 5000+ with 8GB of main memory. Problem descriptions: (a) sorting; (b) first build using repeated inserts, then look-up (each element 100 times); (c) first build using repeated inserts, then destruct using repeated deletes. Data structures: (i) Okasaki's purely functional implementation of red-black trees with the smart-constructor optimisation in place, see Ex.3.10(a) in (1998); (ii) refinement of (i), so that only subtrees on the search path are checked for red-red violations, see Ex.3.10(b); (iii) 1-2 brother trees with leaf nodes eliminated and the smart-constructor optimisation incorporated; (iv) like (iii), but additionally doing away with unary nodes

Random input						
(a) Sorting	10,000	50,000	100,000	500,000	1,000,000	
Red-black trees	0.01	0.22	0.69	13.96	53.91	
Red-black trees'	0.02	0.32	0.98	17.53	68.21	
1-2 brother trees	0.01	0.21	0.67	13.08	50.73	
1-2 brother trees'	0.01	0.20	0.63	12.06	46.79	
(b) Searching	10,000	50,000	100,000	500,000	1,000,000	
Red-black trees	0.62	5.06	12.42	99.14	257.83	
Red-black trees'	0.63	5.18	12.83	103.41	274.18	
1-2 brother trees	0.56	4.75	11.68	93.41	241.99	
1-2 brother trees'	0.62	5.01	12.28	98.09	250.30	
(c) Deletion	10,000	50,000	100,000	500,000	1,000,000	
Red-black trees	0.05	0.79	2.79	61.88	246.94	
Red-black trees'	0.06	0.96	3.29	72.50	293.14	
1-2 brother trees	0.07	0.96	3.41	76.41	309.98	
1-2 brother trees'	0.05	0.69	2.39	52.12	208.49	
Strictly ascending input						
(a) Sorting	10,000	50,000	100,000	500,000	1,000,000	
Red-black trees	0.01	0.17	0.60	15.79	66.70	
Red-black trees'	0.01	0.18	0.63	17.26	73.33	
1-2 brother trees	0.00	0.10	0.37	9.74	40.81	
1-2 brother trees'	0.01	0.11	0.38	9.64	40.26	
(b) Searching	10,000	50,000	100,000	500,000	1,000,000	
Red-black trees	0.41	2.51	5.80	44.79	127.28	
Red-black trees'	0.41	2.52	5.80	46.28	134.20	
1-2 brother trees	0.42	2.46	5.51	38.19	100.35	
1-2 brother trees'	0.42	2.51	5.66	39.37	102.35	
(c) Deletion	10,000	50,000	100,000	500,000	1,000,000	
Red-black trees	0.05	0.81	3.02	77.29	321.07	
Red-black trees'	0.05	0.84	3.21	83.07	344.60	
1-2 brother trees	0.05	0.75	2.80	69.25	284.55	
1-2 brother trees'	0.03	0.50	1.85	45.54	187.10	

or additionally, unary nodes can be eliminated by introducing skewed binary nodes:  $N_2 (N_1 t_1) a_1 t_2$  becomes  $N_{12} t_1 a_1 t_2$  and  $N_2 t_1 a_1 (N_1 t_2)$  becomes  $N_{21} t_1 a_1 t_2$ . Furthermore, to avoid unnecessary tests, the smart binary constructor  $n_2$  should be split into two functions that only check for violations of the invariants involving either the left or the right son (*smart-constructor optimisation*). Again, the details are left as an exercise to the reader.

Several implementations of search trees have appeared in the functional programming literature, including AVL trees (Myers 1984; Bird 1998), 2-3 trees (Reade 1992), red-black trees (Okasaki 1998; Okasaki 1999; Kahrs 2001) and finger trees (Hinze & Paterson 2006). But which to choose? Like AVL trees, but unlike 2-3 trees and red-black trees, brother trees support a simple implementation of deletion. Like 2-3 trees, but unlike AVL trees, there is no need for an additional field that contains the height or a balance factor. (The colour field of red-black trees can be eliminated at the cost of an additional constructor.) Finger trees are much more general and consequently more involved. When it comes to raw speed, initial measurements (see Table 2) are very encouraging. With the above optimisations in place, brother trees consistently outperform red-black trees, whose optimised implementation is reported to fly (Okasaki 1999).

### Acknowledgment

I am grateful to Richard Bird for suggesting the use of representation invariants.

### References

- Adel'son-Vel'skiĭ, G. & Landis, Y. (1962) An algorithm for the organization of information, *Dokl. Akad. Nauk SSSR*, **146**: 263–266. English translation in *Soviet Math. Dokl.* **3**: pp. 1259–1263.
- Aho, A. V., Hopcroft, J. E. & Ullman, J. D. (1974) *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company.
- Bayer, R. (1972) Symmetric binary B-trees: Data structure and maintenance algorithms, *Acta Inform.*, **1**: 290–306.
- Bird, R. (1998) *Introduction to Functional Programming using Haskell*. 2nd ed. Prentice Hall.
- Guibas, L. J., McCreight, E. M., Plass, M. F. & Roberts, J. R. (1977) A new representation for linear lists. In *Conference Record of the 9th Annual ACM Symposium on Theory of Computing*, Boulder, Colorado, United States, pp. 49–60.
- Guibas, L. J. & Sedgewick, R. (1978) A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, pp. 8–21.
- Hinze, R. (1999) Constructing red-black trees. In *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99, Paris, France*, Okasaki, C. (ed). Columbia University, pp. 89–99 (Technical Report, CUCS-023-99).
- Hinze, R. & Paterson, R. (2006) Finger trees: A simple general-purpose data structure, *J. Funct. Program.*, **16** (2): 197–217.
- Kahrs, S. (2001) Functional Pearl: Red-black trees with types, *J. Funct. Program.*, **11** (4): 425–432.

- Myers, E. W. (1984) Efficient applicative data types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Salt Lake City, UT*, Ken Kennedy, Mary S. van Deusen, and Larry Landweber (eds.). ACM, pp. 66–75.
- Okasaki, C. (1998) *Purely Functional Data Structures*. Cambridge University Press.
- Okasaki, C. (1999) Functional pearl: Red-black trees in a functional setting, *J. Funct. Program.*, **9** (4): 471–477.
- Ottmann, T., Six, H.-W. & Wood, D. (1979) On the correspondence between AVL trees and brother trees, *Computing*, **23**, 43–54.
- Reade, C. (1992) Balanced trees with removals: An exercise in rewriting and proof, *Sci. Comput. Program.*, **18** (2): 181–204.