JFP **30**, e3, 13 pages, 2020. © The Author(s) 2020. Published by Cambridge University Press 2020
doi:10.1017/S0956796819000194

FUNCTIONAL PEARLS

An optimal, purely functional implementation of the Garsia–Wachs algorithm

RICHARD S. BIRD
Department of Computer Science, Oxford University,
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
(*e-mail:* bird@cs.ox.ac.uk)

1 Introduction

The Garsia–Wachs algorithm is an algorithm for building a binary leaf tree whose cost is as small as possible. The problem and the algorithm are described in more detail below, but the task is essentially the same as that of building a Huffman coding tree with the added constraint that the fringe of the tree has to be exactly the given list of inputs (in Huffman coding, the fringe of the tree can be any permutation of the input). As we will show below, the Garsia–Wachs algorithm can be implemented with a *linearithmic* running time—a running time of $O(n \log n)$ steps for an input of length n, the same time bound as for Huffman coding.

The problem has an interesting history. It was first discussed in terms of restricted Huffman coding in Gilbert & Moore (1959), where a cubic-time algorithm was proposed. Knuth gave a quadratic-time algorithm in Knuth (1971) for a version of the problem in terms of binary search trees and the expected frequencies of occurrence of the elements in a search. A different method of attack was proposed by Hu & Tucker (1971); see also Hu (1982) and the first edition of Knuth (1998). However, a rigorous proof of the correctness of the algorithm was obtained only later. Then along came a simplification of the Hu–Tucker algorithm in Garsia & Wachs (1977). This algorithm was adopted in place of the Hu–Tucker algorithm in the second edition of Knuth (1998). The best current proofs of its correctness, while not exactly simple, are given in Kingston (1988) and Karpinski et al. (1997); Kingston's proof also appears in Knuth (1998). All these references describe only a simpler, quadratic-time algorithm. There is a fairly short appendix to Garsia & Wachs (1977), written by Robert E. Tarjan, that outlines how to implement the linearithmic version, but some details are missing. Knuth also asks for the linearithmic algorithm in an exercise (6.2.2, Exercise 45) in Knuth (1998), which is answered fairly briefly on page 713 with the suggestion that a doubly linked list should be used.

In 2008, Filliâtre (2008) wrote a very nice pearl on the implementation of the Garsia–Wachs algorithm in ML. His algorithm depended on mutable references and local side effects, so it was "mostly functional" in his words. He described only the quadratic version of the problem, mainly so that he could compare it to Knuth's implementation in C. Our aim in this pearl is to give a purely functional description of the Garsia–Wachs algorithm in Haskell that does achieve the linearithmic bound.

2 The problem

The problem concerns building a binary tree of the following kind:

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

Such a tree will be referred to as a *leaf tree* to distinguish it from another kind of tree needed later on. The fringe of a leaf tree is defined by

$$\begin{aligned}
&\mathit{fringe} :: \mathit{Tree}\ a \to [a]\\
&\mathit{fringe}\ (\mathit{Leaf}\ x) \qquad = [x]\\
&\mathit{fringe}\ (\mathit{Fork}\ t_1\ t_2) = \mathit{fringe}\ t_1 +\!\!+\ \mathit{fringe}\ t_2
\end{aligned}$$

By definition, the depth of a leaf is the length of the path from the root of the tree to the leaf. The depths of the leaves in fringe order are defined by

$$\begin{aligned}
&\mathit{depths} :: \mathit{Tree}\ a \to [\mathit{Int}]\\
&\mathit{depths} = \mathit{from}\ 0\ \textbf{where}\\
&\quad \mathit{from}\ d\ (\mathit{Leaf}\ x) \qquad = [d]\\
&\quad \mathit{from}\ d\ (\mathit{Fork}\ t_1\ t_2) = \mathit{from}\ (d+1)\ t_1 +\!\!+\ \mathit{from}\ (d+1)\ t_2
\end{aligned}$$

Both *fringe* and *depths* can be improved to take linear time; the optimisation, which uses an accumulating parameter, is a standard one and we won't go into details.

In Huffman coding, the input consists of a list of distinct values, usually characters, paired with a measure of their likelihood of occurring in a text, a measure usually called the *weight* of the value. To keep things simple, we will omit the values and suppose the input consists only of a list of positive integer weights. The aim is to construct a leaf tree whose fringe is exactly the given list of weights and whose cost is as small as possible, where

$$\begin{aligned}
&\textbf{type}\ \mathit{Weight} = \mathit{Int}\\
&\mathit{cost} :: \mathit{Tree}\ \mathit{Weight} \to \mathit{Int}\\
&\mathit{cost}\ t = \mathit{sum}\ (\mathit{zipWith}\ (\times)\ (\mathit{fringe}\ t)\ (\mathit{depths}\ t))
\end{aligned}$$

In words, the cost of a tree is the sum of the weighted path lengths from the root to the leaves. For example, the tree of Figure 1 has cost

$$32\times 3 + 12\times 4 + 20\times 4 + 51\times 2 + 57\times 2 + 18\times 3 + 37\times 3 = 605$$

which happens to be the cheapest tree for the weights $[32, 12, 20, 51, 57, 18, 37]$.

3 The algorithm

The Garsia–Wachs algorithm is a two-stage process. In the first stage, we build a leaf tree from the given list of weights and in the second stage we rebuild it:

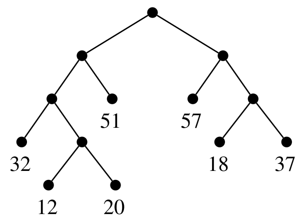


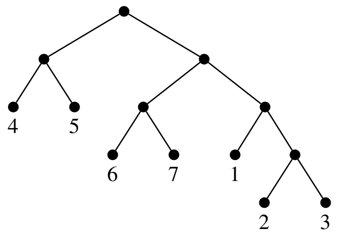
Fig. 1. A minimum cost tree.

$$\begin{aligned}
&\mathit{gwa} :: [\mathit{Weight}] \rightarrow \mathit{Tree}\ \mathit{Weight}\\
&\mathit{gwa}\ \mathit{ws} = \mathit{rebuild}\ \mathit{ws}\ (\mathit{build}\ \mathit{ws})
\end{aligned}$$

With *Label* as another synonym for *Int*, the types of *build* and *rebuild* are

$$\begin{aligned}
&\mathit{build} \quad\ :: [\mathit{Weight}] \rightarrow \mathit{Tree}\ \mathit{Label}\\
&\mathit{rebuild} :: [\mathit{Weight}] \rightarrow \mathit{Tree}\ \mathit{Label} \rightarrow \mathit{Tree}\ \mathit{Weight}
\end{aligned}$$

The result of *build ws* is a tree whose fringe is not *ws* but some permutation of the labels $[1\,..\,n]$, where n is the length of *ws*. The critical property of this tree concerns the depths of its leaves. Suppose the depths are $d_1, d_2, \ldots, d_n$, where d_j is the depth of *Leaf j*. Then there is a tree with minimum cost and fringe *ws* in which the depth of the leaf labelled with w_j is d_j. As an example, suppose *build* applied to $[32, 12, 20, 51, 57, 18, 37]$ produces the tree



The list of depths in numerical order of leaf value is $[3, 4, 4, 2, 2, 3, 3]$. The claim, which we will not prove, is that there is a minimum cost tree for the given input whose depths in fringe order are exactly this list, and that tree is just the tree in Figure 1.

The tree in Figure 1 can be obtained from the one above by starting with a list of pairs; the first component of each pair being a leaf containing the label w_j and the second component being the depth d_j. For our example, this is the list

$$(32, 3), (12, 4), (20, 4), (51, 2), (57, 2), (18, 3), (37, 3)$$

in which a pair (w, d) represents $(\mathit{Leaf}\ w, d)$. This list of pairs is reduced to a single pair by repeatedly combining the first two adjacent trees on the list with the same depth until only a single pair remains. When two trees with a common depth are combined, the depth is reduced by 1. Thus, for our example, we get the sequence of steps pictured in Figure 2, ending with a single tree and a final depth of 0. The trees in Figure 2 are displayed using parentheses, so the tree of Figure 1 is displayed as

(((32 (12 20)) 51) (57 (18 37)))

```
(32,3), (12,4), (20,4), (51,2), (57,2), (18,3), (37,3)
(32,3), ((12 20),3),    (51,2), (57,2), (18,3), (37,3)
((32 (12 20)),2),       (51,2), (57,2), (18,3), (37,3)
(((32 (12 20)) 51),1),          (57,2), (18,3), (37,3)
(((32 (12 20)) 51),1),          (57,2)  ((18 37),2)
(((32 (12 20)) 51),1),          ((57 (18 37)),1)
((((32 (12 20)) 51) (57 (18 37))),0)
```

Fig. 2. Combining trees.

This process is not guaranteed to work for any list of depths, but only those that result from the first stage of the Garsia–Wachs algorithm. For example, the tree ((1 3) 2) has depths $[2, 1, 2]$ in numerical order of leaf label, but no adjacent pair has the same depth so no pair can be combined and the reduction process fails to make progress.

The obvious way to implement this reduction process is by a function *reduce*, defined by

```
reduce :: [(Tree Label, Depth)] → Tree Label
reduce = extract · until singleton step where
  step (x : y : xs) = if depth x == depth y
                      then join x y : xs
                      else  x : step (y : xs)
```

where *Depth* is a synonym for *Int*, *depth* = *snd* and *singleton* is a test for whether a list is a singleton. The functions *extract* and *join* are defined by

```
extract :: [(Tree Label, Depth)] → Tree Label
extract [(t, _)] = t

join :: (Tree Label, Depth) → (Tree Label, Depth) → (Tree Label, Depth)
join (t₁, d) (t₂, _) = (Fork t₁ t₂, d − 1)
```

The function *step* is applied repeatedly until it produces a singleton list. However, this definition of *reduce* can take quadratic time because *step* can take linear time. The inefficiency arises because if *step* finds the first pair to be joined at positions k and $k + 1$, then the next call of *step* will repeat the unsuccessful search on the first $k - 2$ elements when it could begin a new search at position $k - 1$, the earliest position at which two depths could be the same. One way to avoid the inefficiency is to use a *foldl* and a recursive definition of *step*, redefining *reduce* to read

```
reduce :: [(Tree Label, Depth)] → Tree Label
reduce = extract · foldl step [ ] where
  step [ ] y      = [y]
  step (x : xs) y = if depth x == depth y
                    then step xs (join x y)
                    else y : x : xs
```

The first argument to *step* maintains the invariant that no two adjacent pairs on the list have the same depth; this list is kept in reverse order for efficiency. To maintain the invariant, *step* is called recursively whenever two pairs are joined. Each call of *step* takes time proportional to the number of *join* operations, and there are exactly $n - 1$ of these operations in total, so *reduce* now takes linear time.

Having dealt with *reduce*, we can now define *rebuild*:

$$\begin{aligned}&\mathit{rebuild} :: [\mathit{Weight}] \to \mathit{Tree\ Label} \to \mathit{Tree\ Weight}\\&\mathit{rebuild\ ws\ t} = \mathit{reduce}\,(\mathit{zip}\,(\mathit{map\ Leaf\ ws})\,(\mathit{sortDepths\ t}))\end{aligned}$$

The function *sortDepths* sorts the depths of a tree into increasing order of label value. Since labels take the form $[1\,..\,n]$, where n is the number of nodes in the tree, sorting can be accomplished in linear time using an array:

$$\begin{aligned}&\mathit{sortDepths} :: \mathit{Tree\ Label} \to [\mathit{Depth}]\\&\mathit{sortDepths\ t} = \mathit{elems}\,(\mathit{array}\,(1, \mathit{size\ t})\,(\mathit{zip}\,(\mathit{fringe\ t})\,(\mathit{depths\ t})))\end{aligned}$$

The functions *size*, which counts the number of nodes in a tree, *fringe* and *depths* can all be computed in linear time, so *sortDepths* and *rebuild* each take linear time.

It remains to deal with the first stage, the function *build*. This is where the intricacy of the Garsia–Wachs algorithm resides. The plan of attack is to develop a quadratic-time solution first and then improve it to a linearithmic one by a suitable choice of data structure.

For input $[w_1, w_2, \ldots, w_n]$ the starting point is a list

$$(0, w_0), (1, w_1), (2, w_2), \ldots, (n, w_n)$$

of pairs of leaves and weights, so (j, w) abbreviates $(\mathit{Leaf}\ j, w)$. The first pair $(0, w_0)$ is a sentinel pair in which $w_0 = \infty$. Use of a sentinel simplifies the description of the algorithm but is not essential. The following two steps are now repeated until just two pairs remain, the sentinel pair and one other:

1. Given the current list $(0, w_0), \ldots, (t_p, w_p)$, where $p > 1$, find the largest j with $1 \leqslant j < p$ such that $w_{j-1} + w_j \geqslant w_j + w_{j+1}$, equivalently $w_{j-1} \geqslant w_{j+1}$. Such a j is guaranteed to exist since $w_0 = \infty$. Replace the pairs (t_j, w_j) and (t_{j+1}, w_{j+1}) by a single pair

   $$(t_*, w_*) = (\mathit{Fork}\ t_j\ t_{j+1}, w_j + w_{j+1})$$

   giving a new list $(0, w_0), (t_1, w_1), \ldots, (t_{j-1}, w_{j-1}), (t_*, w_*), (t_{j+2}, w_{j+2}), \ldots, (t_p, w_p)$.
2. Now move (t_*, w_*) to the right over all pairs (t, w) for which $w < w_*$.

At the end of this process there are just two pairs left, the sentinel and a second pair whose first component is the required tree. It is difficult to give a convincing intuition as to why this process works, except to say that it is similar in outline to the way Huffman's algorithm proceeds. In Huffman's algorithm the trees are kept in order of weight, and at each step the two lightest trees are combined. The resulting tree is inserted into the remaining list in such a way as to maintain weight order. With the Garsia–Wachs algorithm the process is more complicated.

Here is an example. Suppose we begin with the list

$$(0, \infty), (1, 32), (2, 12), (3, 20), (4, 51), (5, 57), (6, 18), (7, 37)$$

The first pair to be combined is $(6, 18)$ and $(7, 37)$ (because $57 \geqslant 37$). The result is shifted zero places to the right, giving

$$(0, \infty), (1, 32), (2, 12), (3, 20), (4, 51), (5, 57), ((6\ 7), 55)$$

The next pair to be combined is $(2, 12)$ and $(3, 20)$ (because $32 \geqslant 30$). The result is shifted zero places to the right, giving

$$(0, \infty), (1, 32), ((2\ 3), 32), (4, 51), (5, 57), ((6\ 7), 55)$$

The next pair to be combined is $(1, 32)$ and $((2\ 3), 32)$ (because $\infty \geqslant 32$). The result is shifted three places to the right, giving

$$(0, \infty), (4, 51), (5, 57), ((6\ 7), 55), ((1\ (2\ 3)), 64)$$

The remaining three steps are similar in that they all involve combining the second two pairs:

$$\begin{array}{l}(0, \infty), ((6\ 7), 55), ((1\ (2\ 3)), 64), ((4\ 5), 108)\\ (0, \infty), ((4\ 5), 108), (((6\ 7)\ (1\ (2\ 3))), 119)\\ (0, \infty), (((4\ 5)\ ((6\ 7)\ (1\ (2\ 3)))), 227)\end{array}$$

The first component of the second pair is the final tree. Note that the sentinel plays a passive role and is never combined with another pair.

The obvious way to implement this algorithm is repeatedly to scan the whole list from right to left at each step, looking for the largest j such that $w_{j-1} \geqslant w_{j+1}$. However, a better way of organising the search stems from the following observation. Say that a sequence $w_1, w_2, \ldots$ is *two-sorted*, if $w_1 < w_3 < w_5 < \ldots$ and $w_2 < w_4 < w_6 < \ldots$ It follows from the definition of j in step 1 that the sequence $w_j, \ldots, w_p$ is two-sorted. Suppose that the following sequence of w-values is produced by step 2:

$$w_0, w_1, w_2, \ldots, w_{j-1}, w_{j+2}, \ldots, w_{k-1}, w_*, w_k, \ldots, w_p$$

Again, both $w_k, \ldots, w_p$ and $w_{j+2}, \ldots, w_{k-1}, w_*$ are two-sorted because $w_{k-2} < w_*$. Furthermore, we know that $w_{j+r} < w_* \leqslant w_k$ for $2 \leqslant r < k - j$. That means the next pair to be combined is the first one on the following list of three possibilities:

1. w_k and w_{k+1}, provided $w_* \geqslant w_{k+1}$;
2. w_{j+2} and w_{j+3}, provided $w_{j-1} \geqslant w_{j+3}$;
3. w_i and w_{i+1}, provided $1 \leqslant i < j - 1$ and $w_{i-1} \geqslant w_{i+1}$.

These three cases can be captured by expressing *build* in terms of *foldr*:

```
build :: [Weight] → Tree Label
build ws = extract (foldr combine [ ] (zip (map Leaf [0 . .]) (infinity : ws)))
           where extract [_, (t, _)] = t
                 infinity = sum ws
```

No weight arising during the algorithm can be greater than the sum of the input weights, so this definition of *infinity* is adequate. The function *foldr combine* [ ] scans the input from right to left, looking for the next pair to be combined. To define *combine*, we first introduce

```
type Pair = (Tree Label, Weight)

weight :: Pair → Weight
weight (t, w) = w
```

Then *combine* is defined by

```
$combine :: Pair \to [Pair] \to [Pair]$
$combine\ x\ (y : z : xs) = \mathbf{if}\ weight\ x \geqslant weight\ z$
                            $\mathbf{then}\ combine\ x\ (insert\ (fork\ y\ z)\ xs)$
                            $\mathbf{else}\ x : y : z : xs$
$combine\ x\ xs$             $= x : xs$

$fork :: Pair \to Pair \to Pair$
$fork\ (t_1, w_1)\ (t_2, w_2) = (Fork\ t_1\ t_2, w_1 + w_2)$

$insert :: Pair \to [Pair] \to [Pair]$
$insert\ x\ xs = ys \mathbin{+\!\!+} combine\ x\ zs\ \mathbf{where}\ (ys, zs) = split\ x\ xs$

$split :: Pair \to [Pair] \to ([Pair], [Pair])$
$split\ x\ xs = span\ (\lambda y.\ weight\ y < weight\ x)\ xs$
```

The function *insert* makes use of an instance *split* of the general utility function *span* to find the right place for a combined pair to be inserted and calls *combine* again to deal with Case 1. The recursive call to *combine* in the definition of *combine* deals with Case 2, and Case 3 is handled by the right-to-left search in *foldr combine* [ ]. Note that the second argument of *combine* and *insert* is always a two-sorted list, a fact we will exploit later on.

In the worst case, each pass through the data leads to a single pair being combined, so the running time of *build* is quadratic in the length of the input. One worst case is when the list of weights takes the form

$$[k, k, k+1, k+1, \ldots, 2 \times k - 1, 2 \times k - 1]$$

In such a case, the first two trees are combined at each step and inserted at the end of the list. The culprit is the function *insert*, which takes linear time at each step. If we could arrange that *insert* took logarithmic time, then the total running time of the Garsia–Wachs algorithm would be reduced to a linearithmic number of steps. Such an implementation is indeed possible because the second argument to *insert* is not an arbitrary list of pairs but one that is two-sorted on second components.

4 An improved algorithm

The revised implementation is carried out in two stages. The first stage is to rewrite *build* in terms of a new data type *List Pair*, designed for representing lists of pairs of leaf trees and weights that are two-sorted on weights. The following six operations are to be provided:

```
$emptyL\ \ :: List\ a$
$nullL\ \ \ \ :: List\ a \to Bool$
$consL\ \ \ \ :: a \to List\ a \to List\ a$
$deconsL :: List\ a \to (a, List\ a)$
$concatL\ :: List\ a \to List\ a \to List\ a$
$splitL\ \ \ \ :: Pair \to List\ Pair \to (List\ Pair, List\ Pair)$
```

Most of these operations are self-explanatory. Only the last function, *splitL* is specific to lists of pairs; the others work for lists of any type. The function *splitL* is the analogue of *split* used in the definition of *insert*. The function *build* is replaced by a new version

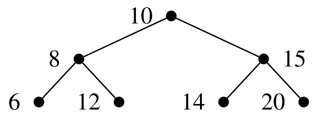


Fig. 3. A two-sorted binary search tree.

buildL, basically the same as before except that [*Pair*] operations are replaced by *List Pair* operations:

```
gwaL :: [Weight] → Tree Weight
gwaL ws = rebuild ws (buildL ws)
buildL :: [Weight] → Tree Label
buildL ws = extractL (foldr combineL emptyL (zip (map Leaf [0 . .]) (infinity : ws)))
            where infinity = sum ws
```

The new functions *extractL* and *combineL* are defined by

```
extractL :: List Pair → Tree Label
extractL xs = t where ((t, _), _) = deconsL (snd (deconsL xs))
combineL :: Pair → List Pair → List Pair
combineL x xs = if nullL xs ∨ nullL ys ∨ weight x < weight z
                then consL x xs else combineL x (insertL (fork y z) zs)
                where (y, ys) = deconsL xs
                      (z, zs) = deconsL ys
insertL :: Pair → List Pair → List Pair
insertL x xs = concatL ys (combineL x zs) where (ys, zs) = splitL x xs
```

The second stage is to implement *List Pair* so that the six operations above take at most logarithmic time. Then *buildL* will take linearithmic time. There are various options and we choose an implementation based on a modification of balanced binary search trees, also called AVL trees after their inventors, Georgy Adelson-Velsky and Evgenii Landis, see Knuth (1998). To motivate the modification, consider the AVL tree in Figure 3 whose nodes are labelled with pairs of leaf trees and their weights, although only the weights are shown. Flattening this tree produces a list of weights $[6, 8, 12, 10, 14, 15, 20]$ which is two-sorted but not sorted. Now suppose we want to insert a new pair with weight w into this AVL tree. We cannot use straightforward binary search because the labels of the tree are not in increasing order of weight. Instead, as well as comparing w to the weight of the leaf tree at the root, we have also to compare it with the weight of the preceding leaf tree in the list. Only if w is greater than both these weights, we can continue by searching the right subtree; otherwise, we have to search the left subtree. In order to avoid repeatedly having to discover the weight of the preceding leaf tree, we can install the preceding leaf tree at the root of an AVL tree, rather than just its weight. If there is no preceding leaf tree, then we can artificially install a copy of the leaf tree. That leads to the tree of Figure 4.

The data type *List Pair* is now introduced as an instance of

```
data List a = Null | Node Height (List a) (a, a) (List a)
type Height = Int
```

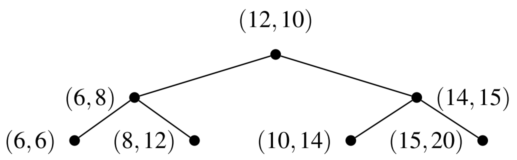


Fig. 4. An augmented two-sorted binary search tree.

The first label of a *Node* records the height of the AVL tree, needed to maintain balance. The second label is a pair of values. The data type invariant on *List a* is that flattening a list produces a list of the form $(x_1,x_1),(x_1,x_2),(x_2,x_3),\ldots,(x_{n-1},x_n)$; when each x is a *Pair*, the list $x_1,x_2,\ldots,x_n$ is two-sorted on weight components.

The implementations of *emptyL* and *nullL* are immediate:

```
emptyL :: List a
emptyL = Null

nullL :: List a → Bool
nullL Null = True
nullL _    = False
```

The operation *consL* adds a new pair as a leftmost element of a binary tree:

```
consL :: a → List a → List a
consL x Null                = node Null (x, x) Null
consL x (Node _ t1 (y, z) t2) = if nullL t1
                               then balance (consL x t1) (x, z) t2
                               else  balance (consL x t1) (y, z) t2
```

For an empty tree t, the operation *consL* $x\,t$ creates a new node with label (x,x). For a non-empty tree t whose left subtree is empty, *consL* $x\,t$ creates a new node with label (x,x) and, since x is now the predecessor of t, assigns x as the new predecessor of t. Otherwise *consL* x is applied to the left subtree of t. The code makes use of two smart constructors, *node* and *balance*. The function *node* is defined by

```
node :: (List a) → (a, a) → List a → List a
node t1 xy t2 = Node h t1 xy t2 where h = 1 + max (height t1) (height t2)

height Null          = 0
height (Node h _ _ _) = h
```

The function *balance* has the same type as *node* and is needed for restoring balance in an AVL tree. The value *balance* $t_1\,x\,t_2$ is defined only for two trees t_1 and t_2 whose heights differ by at most 2. We will not give the code for *balance* since it is completely standard. However, in a moment we will define a second, more general balancing function *gbalance* that can be applied to two trees of arbitrarily different heights.

Next, the function *deconsL* is defined by

```
deconsL :: List a → (a, List a)
deconsL (Node _ t1 xy t2) = if nullL t1 then (snd xy, t2) else (z, balance t3 xy t2)
                          where (z, t3) = deconsL t1
```

This function searches along the left spine of a tree to find the first element.

Next, the function *concatL* is defined by

$$\begin{aligned}
&\mathit{concatL} :: \mathit{List}\ a \to \mathit{List}\ a \to \mathit{List}\ a\\
&\mathit{concatL}\ t_1\ \mathit{Null} = t_1\\
&\mathit{concatL}\ \mathit{Null}\ t_2 = t_2\\
&\mathit{concatL}\ t_1\ t_2 \quad = \mathit{gbalance}\ t_1\ (\mathit{lastL}\ t_1, y)\ t_3\ \textbf{where}\ (y, t_3) = \mathit{deconsL}\ t_2
\end{aligned}$$

The subsidiary function *lastL* returns the last value in a non-empty tree:

$$\begin{aligned}
&\mathit{lastL} :: \mathit{List}\ a \to a\\
&\mathit{lastL}\ (\mathit{Node}\ \_\ t_1\ \mathit{xy}\ t_2) = \textbf{if}\ \mathit{nullL}\ t_2\ \textbf{then}\ \mathit{snd}\ \mathit{xy}\ \textbf{else}\ \mathit{lastL}\ t_2
\end{aligned}$$

In the third clause of $\mathit{concat}\ t_1\ t_2$, the last value in t_1 and the first value in t_2 are combined as a new root. The new function is *gbalance* which is defined for two trees of arbitrarily different heights. To compute $\mathit{gbalance}\ t_1\ x\ t_2$, suppose firstly that $\mathit{abs}\ (h_1 - h_2) \leqslant 2$. Then *gbalance* can be defined as an instance of *balance*. Now suppose $h_1 > h_2 + 2$. In this case, the subtrees $r_1, r_2, \ldots$ along the right spine of t_1 can be traversed to find a subtree $r = r_k$ satisfying

$$0 \leqslant \mathit{height}\ r - \mathit{height}\ t_2 \leqslant 1$$

Such a tree is guaranteed to exist because the subtrees $r_1, r_2, \ldots$ decrease in height by at least 1 and at most 2 at each step. Furthermore, if l is the left sibling of r, then

$$\mathit{abs}\ (\mathit{height}\ l - \mathit{height}\ (\mathit{node}\ r\ x\ t_2)) \leqslant 2$$

because t_1 is a balanced tree and $\mathit{abs}\ (\mathit{height}\ l - \mathit{height}\ r) \leqslant 1$. That means l and $\mathit{node}\ r\ x\ t_2$ can be combined with *balance*. Rebalancing can increase the height of a tree by at most 1, so further rebalancing up the tree maintains the precondition on *balance*. The case $h_2 > h_1 + 2$ is dual, except that the left spine of t_2 is traversed. With that explanation, the complete definition of *gbalance* is as follows:

$$\begin{aligned}
&\mathit{gbalance} :: \mathit{List}\ a \to (a, a) \to \mathit{List}\ a \to \mathit{List}\ a\\
&\mathit{gbalance}\ t_1\ x\ t_2\\
&\quad \mid \mathit{abs}\ (h_1 - h_2) \leqslant 2 = \mathit{balance}\ \ \ t_1\ x\ t_2\\
&\quad \mid h_1 > h_2 + 2 \qquad\ \ = \mathit{balanceR}\ t_1\ x\ t_2\\
&\quad \mid h_1 + 2 < h_2 \qquad\ \ = \mathit{balanceL}\ t_1\ x\ t_2\\
&\quad \textbf{where}\ h_1 = \mathit{height}\ t_1;\quad h_2 = \mathit{height}\ t_2
\end{aligned}$$

The subsidiary functions *balanceR* and *balanceL* are defined by

$$\begin{aligned}
&\mathit{balanceR}, \mathit{balanceL} :: \mathit{List}\ a \to (a, a) \to \mathit{List}\ a \to \mathit{List}\ a\\
&\mathit{balanceR}\ (\mathit{Node}\ \_\ l\ y\ r)\ x\ t_2 = \textbf{if}\ \mathit{height}\ r \geqslant \mathit{height}\ t_2 + 2\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{balance}\ l\ y\ (\mathit{balanceR}\ r\ x\ t_2)\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \mathit{balance}\ l\ y\ (\mathit{node}\ r\ x\ t_2)\\
&\mathit{balanceL}\ t_1\ x\ (\mathit{Node}\ \_\ l\ y\ r) = \textbf{if}\ \mathit{height}\ l \geqslant \mathit{height}\ t_1 + 2\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{balance}\ (\mathit{balanceL}\ t_1\ x\ l)\ y\ r\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \mathit{balance}\ (\mathit{node}\ t_1\ x\ l)\ y\ r
\end{aligned}$$

It is clear that $\mathit{balanceR}\ t_1\ x\ t_2$ takes $O(h_1 - h_2)$ steps, where h_1 and h_2 are the heights of t_1 and t_2. Dually, $\mathit{balanceL}\ t_1\ x\ t_2$ takes $O(h_2 - h_1)$ steps. It follows that $\mathit{gbalance}\ t_1\ x\ t_2$ takes

$O(abs\,(h_1 - h_2))$ steps and that $concatL\ t_1\ t_2$ applied to two balanced trees of sizes m and n takes $O(\log\ n + \log\ m)$ steps.

The final function we have to implement is *splitL*. The function $splitL\ x$ has to split a tree t into a pair of trees (t_1, t_2) in which t_1 consists of the initial segment of t whose weight components are all less than $weight\ x$, and t_2 is the remaining final segment of t. Thus, $concatL\ t_1\ t_2 = t$. To carry out this process, we split a tree into pieces and then sew the pieces together again to make the final pair of trees. Thus,

```
splitL :: Pair → List Pair → (List Pair, List Pair)
splitL x t = sew (pieces x t)
```

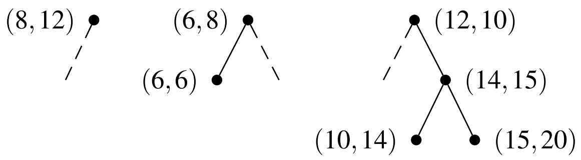
A piece consists of a tree minus one of its subtrees, so it consists of a label and either a left or right subtree:

```
data Piece a = LP (List a) (a, a) | RP (a, a) (List a)
```

A left piece $LP\ t_1\ xy$ is missing its right subtree, and a right piece $RP\ xy\ t_2$ is missing its left subtree. To see how to define $pieces\ x$, consider the tree of Figure 4 and suppose the weight of pair x is 11. The root of the tree has value $(12, 10)$ and 11 is not greater than both these weights, so we construct a right piece and continue by searching the left subtree. If the weight of x were, say 13, then we construct a left piece and continue searching the right subtree. The idea of breaking a tree into pieces is similar to Huet's zipper function, see Huet (1997). The function *pieces* is defined by

```
pieces :: Pair → List Pair → [Piece Pair]
pieces x t = addPieces t [ ] where
  addPieces Null ps = ps
  addPieces (Node _ t₁ (y, z) t₂) ps =
    if w > max (weight y) (weight z)
    then addPieces t₂ (LP t₁ (y, z) : ps)
    else addPieces t₁ (RP (y, z) t₂ : ps)
  w = weight x
```

For example, with $weight\ x = 11$, the tree of Figure 4 produces the three pieces

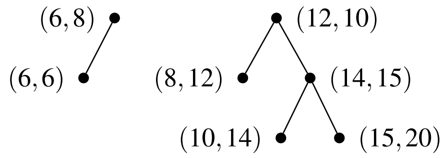


in which the missing tree is indicated by a dashed line.

Finally, we can sew a list of pieces into a pair of trees by

```
sew :: [Piece Pair] → (List Pair, List Pair)
sew = foldl step (Null, Null)
        where step (t₁, t₂) (LP t x) = (gbalance t x t₁, t₂)
              step (t₁, t₂) (RP x t) = (t₁, gbalance t₂ x t)
```

For example, sewing the three pieces above produces the two trees



We claim that *split* x t takes $O(h)$ steps, where $h = height\ t$. Certainly, *pieces* x t takes this time, so we have to show that *sew* does too. If we define the height of a piece to be the height of the tree associated with the piece, then *pieces* x t produces a list of pieces whose heights $h_1, h_2, ..., h_k$ are strictly increasing and bounded above by h. For example, the pieces pictured above have heights 0, 1, 2. The total cost of *sew* is proportional to

$$(h_1 - 0) + (h_2 - h_1) + ... + (h_k - h_{k-1}) \leqslant h$$

because each call of *gbalance* t_1 x t_2 takes time proportional to the difference between the heights of t_1 and t_2. Thus, both *piece* and *sew* take logarithmic time in the size of the sets, so *split* does too.

5 Comparisons and conclusions

In his pearl, Filliâtre compared his OCaml implementation with a C implementation written by Knuth as he was preparing for the second edition of Knuth (1998). For example, with $n = 500$ and random weights in the range $1 \leqslant w \leqslant 5000$, the OCaml code took 1.82 s, while the C code took 1.56 s. Both implementations had the same quadratic-time worst-case complexity. But that was in 2008 and computers have got faster in the past dozen years. One of the referees of this paper translated our Haskell quadratic-time algorithm for *gwa* into OCaml and compared it with Filliâtre's original benchmark code and inputs, showing that *gwa* was about three times slower than Filliâtre's version (0.64 ms versus 0.2 ms). This statistic was confirmed by running the code on the author's machine. The linearithmic version *gwaL* was translated into OCaml and the results compared. With $n = 5000$, both Filliâtre's version and *gwaL* took about the same time (14 ms), but for greater values of n the linearithmic version pulled ahead, being about three times faster for $n = 15000$.[1]

In summary, it does seem that the internal use of references and purely local side effects present in the OCaml implementation are not necessary for reasonable efficiency. Wadler's famous question "Shall I be pure or impure?" can still be answered, at least for problems in algorithm design, with "Stay pure as long as possible". Indeed in our new book (Bird & Gibbons, 2020), from which this pearl was extracted, we adopt a purely functional approach to many other algorithms in computer science.

Acknowledgments

I am indebted to the referees of this paper for a careful reading of the original submission. One referee pointed out that my original code of *gwaL* was a factor of log n worse than

[1] The code for *gwa* and *gwaL* in both Haskell and Ocaml is available at `http://www.cs.ox.ac.uk/people/richard.bird/GW.lhs` (and similarly for `GW.ml`).

linearithmic and suggested the idea of keeping both the current value and its predecessor in each node to avoid having to search for predecessors at each step. Another referee, as mentioned above, carried out the timing comparisons for Filliâtre's version and ours. For these and many other suggestions for improvement, I am very grateful.

Conflict of Interest

None

References

Bird, R. & Gibbons, J. (2020) *Algorithm Design with Haskell*. Cambridge University Press. To appear.

Filliâtre, J.-C. (2008) A functional implementation of the Garsia Wachs algorithm. In ACM SIGPLAN Workshop on ML, Victoria, British Columbia, Canada.

Garsia, A. M. & Wachs, M. L. (1977) A new algorithm for minimum cost binary trees. *SIAM J. Comput.* **6**(4), 622–642.

Gilbert, E. N. & Moore, E. F. (1959) Variable length binary encodings. *Bell Syst. Tech. J.* **38**, 933–968.

Hu, T. C. (1982) *Combinatorial Algorithms*. Reading, MA: Addison-Wesley.

Hu, T. C. & Tucker, A. C. (1971) Optimal computer-search trees and variable-length alphabetic codes. *SIAM J. Appl. Math.* **21**, 514–532.

Huet, G. (1997) The Zipper. *J. Funct. Program.* **7**(5), 549–554.

Karpinski, M., Larmore, L. L. & Rytter, W. (1997) Correctness of constructing optimal alphabetic tree revisited. *Theor. Comput. Sci.* **180**(1–2), 309–324.

Kingston, J. H. (1988) A new proof of the Garsia-Wachs Algorithm. *J. Algorithms* **9**, 129–136.

Knuth, D. E. (1971) Optimum binary search trees. *Acta Inf.* **1**(1), 14–25.

Knuth, D. E. (1998) *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA: Addison-Wesley.