## Journal of Functional Programming <br> http://journals.cambridge.org/JFP

Additional services for Journal of Functional Programming:

Email alerts: Click here
Subscriptions: Click here
Commercial reprints: Click here
Terms of use : Click here

# FUNCTIONAL PEARL On building trees with minimum height 

RICHARD S. BIRD

Journal of Functional Programming / Volume 7 / Issue 04 / July 1997, pp 441-445
DOI: null, Published online: 08 September 2000
Link to this article: http://journals.cambridge.org/abstract S0956796897002803
How to cite this article:
RICHARD S. BIRD (1997). FUNCTIONAL PEARL On building trees with minimum height. Journal of Functional Programming, 7, pp 441-445

Request Permissions : Click here

# FUNCTIONALPEARL On building trees with minimum height 

RICHARD S. BIRD<br>Programming Research Group, University of Oxford, Wolfson Building, Parks Rd, Oxford OX1 3QD, UK

## 1 Introduction

A common solution to the problem of handling list indexing efficiently in a functional program is to build a binary tree. The tree has the given list as frontier and is of minimum height. Each internal node of the tree stores size information (actually, the size of its left subtree) to direct the search for an element at a given position in the frontier. One application was considered in my previous pearl (Bird, 1997). There are two complementary methods for building such a tree, both of which can be implemented in linear time. One method is 'recursive', or top down, and works by splitting the list into two equal halves, recursively building a tree for each half, and then combining the two results. The other method is 'iterative', or bottom up, and works by first creating a list of singleton trees, and then repeatedly combining the trees in pairs until just one tree remains. The two methods lead to different trees, but in each case the result is a tree with smallest possible height.

The form of the bottom-up algorithm suggests the following intriguing generalisation: given an arbitrary sequence of $N$ trees together with their heights, is there an $O(N)$ time algorithm to combine them into a single tree of minimum height? The restriction, of course, is that the given trees should appear as subtrees of the final tree in the order they appear in the sequence. An alternative but equivalent version of the problem is to ask: given a sequence $h s=\left[h_{1}, h_{2}, \ldots, h_{N}\right]$ of natural numbers, can one find an $O(N)$ algorithm to build a tree $t$ with frontier $h s$ that minimises

$$
\operatorname{cost} t=\left(\max i: 1 \leq i \leq N: \operatorname{depth}_{i}+h_{i}\right) ?
$$

The depth, depth $_{i}$, of the $i$ th tip is the length of the path in $t$ from the root to tip number $i$. The height of a tree is the maximum of the depths of its tips.

Since cost is a regular cost function in the sense of (Hu, 1982), the Hu-Tucker algorithm (Knuth, 1973) is applicable to the problem, but the best implementation of that algorithm has a running time of $O(N \log N)$. Our aim in this pearl is to give a direct construction of a linear-time algorithm.

## 2 A greedy algorithm

Given a sequence $t_{i}(1 \leq i \leq N)$ of trees with heights $h_{i}(1 \leq i \leq N)$, say that the pair ( $t_{i}, t_{i+1}$ ) for $1 \leq i<N$ is a local minimum pair (abbreviated: lmp) if

$$
\max \left(h_{i-1}, h_{i}\right) \geq \max \left(h_{i}, h_{i+1}\right)<\max \left(h_{i+1}, h_{i+2}\right),
$$

where, by convention, $h_{0}=h_{N+1}=\infty$. Thus an lmp is a point in the sequence $m_{i}=\max \left(h_{i}, h_{i+1}\right)(0 \leq i \leq N)$ where the sequence stops descending and starts increasing. Equivalently, it is easy to show that $\left(t_{i}, t_{i+1}\right)$ is an $\operatorname{lmp}$ if and only if either

1. $h_{i+1} \leq h_{i}<h_{i+2}$, or
2. $h_{i}<h_{i+1}<h_{i+2}$ and $h_{i-1} \geq h_{i+1}$.

This alternative characterisation is used in a case analysis in the final program.
There is at least one lmp, namely, the rightmost pair $\left(t_{i}, t_{i+1}\right)$ for which $m_{i}$ is a minimum, but there may be others. In outline, the greedy algorithm is to combine the rightmost $\operatorname{lmp}$ at each stage, repeating until just one tree remains. It is worth mentioning, for the greater comfort of imperative programmers, that there is a dual variant in which the notion of an $\operatorname{lmp}$ is modified by replacing $\geq$ by $>$ and $<$ by $\leq$. Then the greedy algorithm combines the leftmost lmp at each stage. But this pearl is designed for functional programmers who, other things being equal, like to process from right to left.

To illustrate the greedy algorithm, consider the following computation in which the numbers denote the heights of the trees, and the braces denote the lmps at each stage (recall that the height of a tree is one more than the greater of the heights of its two subtrees):

$$
\begin{aligned}
& 4, \underbrace{2,3}, 5, \underbrace{1,4}, 6 \\
\Rightarrow & 4, \underbrace{2,3}, \underbrace{5,5}, 6 \\
\Rightarrow & 4, \underbrace{2,3}, \underbrace{6,6} \\
\Rightarrow & 4, \underbrace{2,3}, 7 \\
\Rightarrow & \underbrace{4,4,7} \\
\Rightarrow & \underbrace{5,7} \\
\Rightarrow & 8
\end{aligned}
$$

The correctness of the greedy algorithm rests on the following definition and lemma. Say that two trees are siblings in a tree $T$ if they are the immediate subtrees of some node of $T$.

## Lemma 1

Suppose $\left(t_{i}, t_{i+1}\right)$ is an $\operatorname{lmp}$ in a given sequence of trees $t_{j}(1 \leq j \leq N)$. Then the sequence can be combined into a tree $T$ of minimum height in which $\left(t_{i}, t_{i+1}\right)$ are siblings.


Fig. 1. $t_{i}$ and $t_{i+1}$ not siblings.


Fig. 2. Case $m \leq n$.

## Proof

Suppose by way of contradiction that there is no optimum tree (i.e. a tree of minimum height) in which ( $t_{i}, t_{i+1}$ ) are siblings. Let $T$ be an optimum tree for the sequence. Since ( $t_{i}, t_{i+1}$ ) are not siblings in $T$ it follows that $T$ contains some subtree of the form depicted in Figure 1 in which not both $m$ and $n$ can be zero. In the figure, $X_{1}, \ldots, X_{m}$ are subtrees erected on some final segment of $t_{1}, \ldots t_{i-1}$, and $Y_{n}, \ldots, Y_{1}$ are subtrees erected on some initial segment of $t_{i+2}, \ldots, t_{N}$.

Say that a subtree of $T$ is critical if increasing its depth increases the height of $T$. There are three cases to consider:
(i) Neither $t_{i}$ nor $t_{i+1}$ are critical.

In this case, if $m<=n$ (so $n \neq 0$ ), then we can move $t_{i+1}$ to be a sibling of $t_{i}$, as in figure 2 . The resulting tree is still optimal since the depth of $t_{i}$ is increased by one, and the depth of $t_{i+1}$ by at most one. This contradicts the assumption. Dually, if $n \leq m$ we can move $t_{i}$ to be a sibling of $t_{i+1}$.
(ii) $t_{i}$ is critical, $t_{i+1}$ is not critical.

In this case the height of the tree of figure 1 is $m+1+h_{i}$. If $m>n$, then we can move $t_{i}$ to be a sibling of $t_{i+1}$ without increasing the height of the tree. This contradicts our assumption. If $m \leq n$ (so $n \neq 0$ ), then the tree $Y_{n}$ exists and is either $t_{i+2}$ or contains $t_{i+2}$ as its leftmost subtree. In either case, we have $h_{i} \geq \max \left(h_{i+1}, h_{i+2}\right)$. Hence

$$
\max \left(h_{i}, h_{i+1}\right) \geq h_{i} \geq \max \left(h_{i+1}, h_{i+2}\right)
$$

contradicting the assumption that $\left(t_{i}, t_{i+1}\right)$ is an 1 mp .
(iii) $t_{i+1}$ is critical.

This time we have $m+1+h_{i} \leq n+1+h_{i+1}$. If $n<m$ (so $m \neq 0$ ), then $h_{i}<h_{i+1}$. Moreover, since $m \neq 0$, the tree $X_{m}$ exists and is either $t_{i-1}$ or has $t_{i-1}$ as its rightmost
subtree. In either case $h_{i-1}<h_{i+1}$, so

$$
\max \left(h_{i-1}, h_{i}\right)<h_{i+1} \leq \max \left(h_{i}, h_{i+1}\right)
$$

This contradicts the assumption that $\left(t_{i}, t_{i+1}\right)$ is an 1 mp .
Finally, if $n \geq m$ (so $n \neq 0$ ), then $Y_{n}$ contains $t_{i+2}$ and so $h_{i+1} \geq h_{i+2}$. Hence

$$
\max \left(h_{i}, h_{i+1}\right) \geq h_{i+1} \geq \max \left(h_{i+1}, h_{i+2}\right),
$$

again contradicting our assumption.

## 3 Implementation

There are a number of ways the algorithm can be implemented. Since lmps cannot overlap, i.e. it is not possible for both $\left(t_{i}, t_{i+1}\right)$ and $\left(t_{i+1}, t_{i+2}\right)$ to be 1 mps , one possibility is to scan the list of trees repeatedly from right to left, combining all 1 mps found during each scan. However, it is possible that only one 1 mp will be found during each scan, so this method may take $\Omega\left(n^{2}\right)$ steps on a list of length $n$.

Instead we will implement a stack-based algorithm. For simplicity let us ignore tip values and suppose that trees are given as elements of the datatype

$$
\text { data Tree }=\text { Tip } \mid \text { Bin Int Tree Tree, }
$$

in which height $(\operatorname{Bin} n x y)=n$. Below we will use the two functions

$$
\begin{array}{ll}
\text { join } & :: \text { Tree } \rightarrow \text { Tree } \rightarrow \text { Tree } \\
\text { join } x y & = \\
\text { Bin }(\max (h t x)(\text { ht } y)+1) \times y \\
\text { ht } & :: \text { Tree } \rightarrow \text { Int } \\
\text { ht Tip } & = \\
\text { ht }(\text { Bin } n x y) & =
\end{array}
$$

The algorithm for building a tree, build say, is given as the composition of two loops:

$$
\text { build }=\text { foldl } 1 \text { join } \cdot \text { foldr step }[]
$$

The main processing loop foldr step [] produces a list of trees in strictly increasing order of height. This constraint is the invariant of the loop. Thus, the expression foldr step stack rest represents a partially processed list of trees rest ++ stack in which the trees in stack appear in strictly increasing order of height. In particular, if rest is empty, then the first two trees in stack are the unique lmp of the sequence. After joining them, the first two remaining trees are again the unique 1 mp of the sequence; and so on. The loop foldl1 join therefore combines these unique 1 mps into the final tree.

Suppose now that $t$ is the next tree to be processed, i.e. $t$ is the last element of rest. For simplicity, we consider first the case when stack contains at least two trees; thus stack $=u: v: t s$. If ht $t<h t u$, then $t$ is added to stack, maintaining the invariant. If, on the other hand, ht $t \geq h t u$, then either $(t, u)$ or $(u, v)$ is the rightmost 1 mp . If ht $t \geq h t v$, then $(u, v)$ is an 1 mp because

$$
\max (\text { ht } t, \text { ht } u)=h t t \geq h t v=\max (\text { ht } u, \text { ht } v)<\max (\text { ht } v, \text { ht } w),
$$

where $w$ is the next (possibly fictitious) tree on the stack. The height of $w$ is greater than that of $v$ by the invariant. If, on the other hand, ht $t<h t v$, then $(t, u)$ is the rightmost 1 mp because, whatever tree $s$ is next in the remaining input, we have

$$
\max (\text { ht } s, \text { ht } t) \geq \text { ht } t \geq \max (\text { ht } t, \text { ht } u)<\max (\text { ht } u, \text { ht } v) \text {. }
$$

Combining either of these lmps may create new lmps, so the list has to be processed again.

The full definition of step is

$$
\begin{array}{lll}
\text { step } t] & & {[t]} \\
& & \\
\text { step } t[u] & & \text { if } h t t<h t u \\
& =[j o i n t u], & \\
\text { otherwise } \\
\text { step } t(u: v: t s) & =t: u: v: t s, & \\
& & \text { if } h t t<h t u \\
& & \text { step }(\text { join } t u)(v: t s), \\
& \text { if } h t t<h t v \\
& =\text { step } t(\text { step }(\text { join } u v) \text { ts }), & \\
\text { otherwise }
\end{array}
$$

A standard amortisation argument shows that the program for build takes linear time: each input adds at most one tree to the stack, and the time to evaluate step is proportional to the number of trees removed from the stack. All in all, a neat solution to a nice problem.

## Acknowledgement

I would like to thank Sharon Curtis and a referee for help in improving the presentation.

## References

Bird, R. S. (1997) On merging and selection. J. Functional Programming.
Hu, T. C. (1982) Combinatorial Algorithms. Addison-Wesley.
Knuth, D. E. (1973) The Art of Computer Programming, Vol 4: Searching and Sorting. AddisonWesley.

