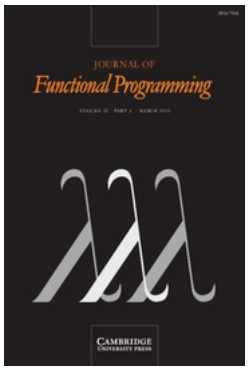
# Journal of Functional Programming

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


# Three algorithms on Braun trees

CHRIS OKASAKI

Journal of Functional Programming / Volume 7 / Issue 06 / November 1997, pp 661 - 666
DOI: null, Published online: 08 September 2000

**Link to this article:** http://journals.cambridge.org/abstract_S0956796897002876

**How to cite this article:**
CHRIS OKASAKI (1997). Three algorithms on Braun trees . Journal of Functional Programming, 7, pp 661-666

**Request Permissions :** Click here

CAMBRIDGE JOURNALS

**Downloaded from http://journals.cambridge.org/JFP, IP address: 132.248.181.1 on 04 Sep 2013**

*J. Functional Programming* **7** (6): 661–666, November 1997. Printed in the United Kingdom 
© 1997 Cambridge University Press

# FUNCTIONAL PEARL

## *Three algorithms on Braun trees*

CHRIS OKASAKI*

*School of Computer Science, Carnegie Mellon University,*
*5000 Forbes Avenue, Pittsburgh, PA 15213, USA*
(*e-mail:* cokasaki@cs.cmu.edu)

### 1 Introduction

Among the many flavours of balanced binary trees, Braun trees (Braun and Rem, 1983) are perhaps the most circumscribed. For any given node of a Braun tree, the left subtree is either exactly the same size as the right subtree, or one element larger. Braun trees always have minimum height, and the shape of each Braun tree is completely determined by its size. In return for this rigor, algorithms that manipulate Braun trees are often exceptionally simple and elegant, and need not maintain any explicit balance information.

Braun trees have been used to implement both flexible arrays (Braun and Rem, 1983; Hoogerwoord, 1992; Paulson, 1996) and priority queues (Paulson, 1996; Bird, 1996). Most operations involving a single element (e.g. adding, removing, inspecting or updating an element) take $O(\log n)$ time, since the trees are balanced. We consider three algorithmically interesting operations that manipulate entire trees. First, we give an $O(\log^2 n)$ algorithm for calculating the size of a tree. Second, we show how to create a tree containing $n$ copies of some element $x$ in $O(\log n)$ time. Finally, we describe an order-preserving algorithm for converting a list to a tree in $O(n)$ time. This last operation is not nearly as straightforward as it sounds!

#### *Notation*

A tree is either empty, written $\langle\rangle$, or a triple $\langle x, s, t\rangle$, where $x$ is an element and $s$ and $t$ are trees. The subtrees $s$ and $t$ must satisfy the balance condition

$$|t| + 1 \geq |s| \geq |t|$$

We abbreviate the leaf $\langle x, \langle\rangle, \langle\rangle\rangle$ as $\langle x\rangle$.

* This research was sponsored by the Advanced Research Projects Agency CSTO under the title 'The Fox Project: Advanced Languages for Systems Software', ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050.

## 2 Calculating the size of a tree

It is trivial to calculate the size of a tree in $O(n)$ time by counting every node individually.

$$\begin{array}{lcl} \mathit{size}\ \langle\rangle & = & 0 \\ \mathit{size}\ \langle x,s,t\rangle & = & 1+\mathit{size}\ s+\mathit{size}\ t \end{array}$$

However, this fails to take advantage of the fact that, once we know the size of one subtree, there are only two possibilities for the size of the other subtree. If $|t|=m$ then either $|s|=m$ or $|s|=m+1$. Let us define a function $\mathit{diff}\ s\ m$ that returns 0 if $|s|=m$ and 1 if $|s|=m+1$. Then, *size* can be rewritten

$$\begin{array}{lcl} \mathit{size}\ \langle\rangle & = & 0 \\ \mathit{size}\ \langle x,s,t\rangle & = & \textbf{let}\ m=\mathit{size}\ t\ \textbf{in}\ 1+2*m+\mathit{diff}\ s\ m \end{array}$$

The base cases for *diff* are trivial:

$$\begin{array}{lcl} \mathit{diff}\ \langle\rangle\ 0 & = & 0 \\ \mathit{diff}\ \langle x\rangle\ 0 & = & 1 \end{array}$$

The remaining cases use the easily verified fact that, if $|\langle x,s,t\rangle|=m$, then $|s|=\lceil(m-1)/2\rceil$ and $|t|=\lfloor(m-1)/2\rfloor$. Now, suppose that $|\langle x,s,t\rangle|$ is either $m$ or $m+1$. If $m$ is odd, then the size of the right subtree is fixed, since $\lfloor(m-1)/2\rfloor=(m-1)/2=\lfloor(m+1-1)/2\rfloor$. On the other hand, the size of the left subtree might be either $\lceil(m-1)/2\rceil=(m-1)/2$ or $\lceil(m+1-1)/2\rceil=(m+1)/2$. We can determine which by recursing on the left subtree.

$$\mathit{diff}\ \langle x,s,t\rangle\ (2*k+1)=\mathit{diff}\ s\ k$$

If $m$ is even, the situation is reversed – the size of the left subtree is fixed and we recurse on the right subtree.

$$\mathit{diff}\ \langle x,s,t\rangle\ (2*k+2)=\mathit{diff}\ t\ k$$

The complete algorithm is

$$\begin{array}{lcl} \mathit{size}\ \langle\rangle & = & 0 \\ \mathit{size}\ \langle x,s,t\rangle & = & \textbf{let}\ m=\mathit{size}\ t\ \textbf{in}\ 1+2*m+\mathit{diff}\ s\ m \\ \\ \mathit{diff}\ \langle\rangle\ 0 & = & 0 \\ \mathit{diff}\ \langle x\rangle\ 0 & = & 1 \\ \mathit{diff}\ \langle x,s,t\rangle\ (2*k+1) & = & \mathit{diff}\ s\ k \\ \mathit{diff}\ \langle x,s,t\rangle\ (2*k+2) & = & \mathit{diff}\ t\ k \end{array}$$

The running time of *size* is dominated by the calls to *diff*, one for each left subtree along the right spine. Each call to *diff* runs in $O(\log n)$ time, for a total of $O(\log^2 n)$.

## 3 Creating a tree by copying

Suppose we want a function $\mathit{copy}\ x\ n$ that creates a tree containing $n$ copies of $x$. Of course, we can easily do this in $O(n)$ time with

$$\begin{array}{lcl} \mathit{copy}\ x\ 0 & = & \langle\rangle \\ \mathit{copy}\ x\ n & = & \langle x,\mathit{copy}\ x\ \lceil(n-1)/2\rceil,\mathit{copy}\ x\ \lfloor(n-1)/2\rfloor\rangle \end{array}$$

However, this function will frequently call *copy* multiple times on the same arguments. In particular, whenever $n$ is odd, the two recursive calls will be identical. Our next version of *copy* takes advantage of this fact.

$$\begin{array}{lcl} copy\ x\ 0 & = & \langle\rangle \\ copy\ x\ (2 * m + 1) & = & \textbf{let}\ t = copy\ x\ m\ \textbf{in}\ \langle x, t, t\rangle \\ copy\ x\ (2 * m + 2) & = & \langle x, copy\ x\ (m + 1), copy\ x\ m\rangle \end{array}$$

**Exercise.** Show that this version of *copy* runs in

$$O(\text{fib}\ (\log_2 n)) = O(\phi^{\log_2 n}) = O(n^{\log_2 \phi}) = O(n^{0.69\ldots})$$

time, where $\phi$ is the golden mean, $(1 + \sqrt{5})/2$. □

We can do still better by realizing that *copy x* $(m + 1)$ and *copy x m* produce very similar results. The former is the result of adding a single $x$ to the latter. Writing the cons function on trees $x \oplus t$, we get

$$\begin{array}{lcl} copy\ x\ 0 & = & \langle\rangle \\ copy\ x\ (2 * m + 1) & = & \langle x, t, t\rangle \\ copy\ x\ (2 * m + 2) & = & \langle x, x \oplus t, t\rangle \\ \multicolumn{3}{c}{\textbf{where}\ \ t = copy\ x\ m} \end{array}$$

where

$$\begin{array}{lcl} x \oplus \langle\rangle & = & \langle x\rangle \\ x \oplus \langle y, s, t\rangle & = & \langle x, y \oplus t, s\rangle \end{array}$$

is the standard algorithm for adding an element to a Braun tree. Note that this function swaps the subtrees $s$ and $t$. This behaviour is a distinguishing feature of Braun trees. It is used to maintain the balance condition, since

$$\begin{array}{lcl} & & |t| + 1 \geq |s| \geq |t| \\ \Rightarrow & & |s| + 1 \geq |t| + 1 \geq |s| \end{array}$$

This version of *copy* runs in $O(\log^2 n)$ time. The analysis is identical to that of *size*.

For our final version of *copy*, we delve deeper into the structure of Braun trees. Note that if $|\langle x_1, s_1, t_1\rangle| = |\langle x_2, s_2, t_2\rangle| + 1$, then either $|s_1| = |t_1| = |s_2| = |t_2| + 1$ or $|s_1| - 1 = |t_1| = |s_2| = |t_2|$. In either case, we can create trees of both size $n$ and size $n + 1$ given only trees of sizes $\lfloor (n - 1)/2 \rfloor$ and $\lfloor (n - 1)/2 \rfloor + 1$. Applying this idea recursively yields

$$\begin{array}{lcl} copy\ x\ n & = & snd\ (copy2\ x\ n) \\ \\ copy2\ x\ 0 & = & (\langle x\rangle, \langle\rangle) \\ copy2\ x\ (2 * m + 1) & = & (\langle x, s, t\rangle, \langle x, t, t\rangle) \\ copy2\ x\ (2 * m + 2) & = & (\langle x, s, s\rangle, \langle x, s, t\rangle) \\ \multicolumn{3}{c}{\textbf{where}\ \ (s, t) = copy2\ x\ m} \end{array}$$

where *copy2 x n* returns a pair of trees of sizes $n + 1$ and $n$, respectively. This runs in only $O(\log n)$ time.

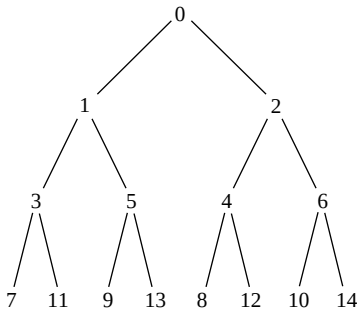



Fig. 1. A Braun tree of size 15, with each node labelled by its index.

## 4 Converting a list to a tree

The previous algorithms have applied to Braun trees representing either flexible arrays or priority queues. This last algorithm applies only to flexible arrays. See Bird (1996) for a similar treatment of priority queues.

Given a list, we want to create a flexible array containing the same elements in the same order. Figure 1 illustrates the order of elements in a Braun tree representing an array. This order is defined recursively. Element 0 of $\langle x, s, t\rangle$ is $x$. The left subtree $s$ contains the odd elements, while the right subtree $t$ contains the (positive) even elements. Thus, for example, the indexing function $s\,!\,i$ can be written

$$\begin{array}{lcl} \langle x, s, t\rangle\,!\,0 & = & x \\ \langle x, s, t\rangle\,!\,(2 * i + 1) & = & s\,!\,i \\ \langle x, s, t\rangle\,!\,(2 * i + 2) & = & t\,!\,i \end{array}$$

Now, a simple but inefficient way to convert a list to an array is to insert the elements one at a time into an initially empty array.

$$makeArray\ xs\ =\ foldr\ (\oplus)\ \langle\rangle\ xs$$

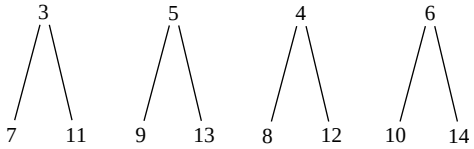
Unfortunately, this takes $O(n \log n)$ time.

A second approach exploits the fact that the left subtree contains the odd elements and the right subtree contains the even elements.

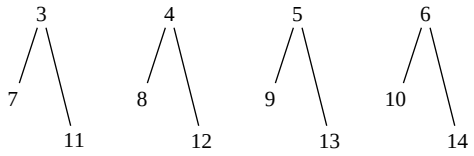
$$\begin{array}{lcl} makeArray\ [\,] & = & \langle\rangle \\ makeArray\ (x : xs) & = & \langle x, makeArray\ odds, makeArray\ evens\rangle \\ & & \textbf{where}\ (odds, evens) = unravel\ xs \\ \\ unravel\ [\,] & = & ([\,], [\,]) \\ unravel\ (x : xs) & = & (x : evens, odds) \\ & & \textbf{where}\ (odds, evens) = unravel\ xs \end{array}$$

But this also takes $O(n \log n)$ time.

This last approach works top down. Let us instead try to work bottom up. First, consider the relationship between adjacent rows. For example, here are the third and fourth rows from Figure 1.




A pattern emerges as we rearrange the nodes from the third row in numerical order. We draw the subtrees slightly askew to emphasize our point.




From this picture, we see that the first half of each row become the left children of the previous row, and the second half of each row become the right children of the previous row. We begin to code this idea as an algorithm by partitioning the input list into rows.

$$\begin{array}{lcl} \mathit{rows}\ k\ [\,] & = & [\,] \\ \mathit{rows}\ k\ \mathit{xs} & = & (k, \mathit{take}\ k\ \mathit{xs}) : \mathit{rows}\ (2 * k)\ (\mathit{drop}\ k\ \mathit{xs}) \end{array}$$

For example,

$$\mathit{rows}\ 1\ [0..14] = [(1,[0]),(2,[1,2]),(4,[3,4,5,6]),(8,[7,8,9,10,11,12,13,14])]$$

Note that we explicitly store the size of each row. This size may be inaccurate for the last row if it is not full.

Next, we process the rows bottom up. At each step, we combine a row with a list of its subtrees.

$$\begin{array}{l} \mathit{build}\ (k, \mathit{xs})\ \mathit{ts} = \mathit{zipWith3}\ \mathit{makeNode}\ \mathit{xs}\ \mathit{ts}_1\ \mathit{ts}_2 \\ \quad \textbf{where}\ (\mathit{ts}_1, \mathit{ts}_2) = \mathit{split}\ k\ (\mathit{ts} +\!\!+ \mathit{repeat}\ \langle\rangle) \\ \quad\quad \mathit{makeNode}\ x\ s\ t = \langle x, s, t\rangle \end{array}$$

We first split the list of subtrees into left children and right children, and then zip these lists with *xs* to make a list of trees. We use the infinite list *repeat* $\langle\rangle$ to fill in $\langle\rangle$ for any missing children. Note that we are not committing to lazy evaluation by using an infinite list – we could easily replace it with a finite list of length $2k$.

Finally, we fold *build* across the list of rows, and extract the head of the result.

$$\mathit{makeArray} = \mathit{head} \circ \mathit{foldr}\ \mathit{build}\ [\langle\rangle] \circ \mathit{rows}\ 1$$

The singleton list $[\langle\rangle]$ guarantees that *head* will find a tree even if *xs* is empty. The complete algorithm is

$$\begin{array}{lcl} \mathit{rows}\ k\ [\,] & = & [\,] \\ \mathit{rows}\ k\ \mathit{xs} & = & (k, \mathit{take}\ k\ \mathit{xs}) : \mathit{rows}\ (2 * k)\ (\mathit{drop}\ k\ \mathit{xs}) \end{array}$$

$$\begin{array}{l} \mathit{build}\ (k, \mathit{xs})\ \mathit{ts} = \mathit{zipWith3}\ \mathit{makeNode}\ \mathit{xs}\ \mathit{ts}_1\ \mathit{ts}_2 \\ \quad \textbf{where}\ (\mathit{ts}_1, \mathit{ts}_2) = \mathit{split}\ k\ (\mathit{ts} +\!\!+ \mathit{repeat}\ \langle\rangle) \\ \quad\quad \mathit{makeNode}\ x\ s\ t = \langle x, s, t\rangle \end{array}$$

$$\mathit{makeArray} = \mathit{head} \circ \mathit{foldr}\ \mathit{build}\ [\langle\rangle] \circ \mathit{rows}\ 1$$

Each call to *rows* or *build* takes $O(k)$ time, so the entire program runs in $O(n)$ time.

**Exercise.** Invert this program to obtain a function that lists the elements of a Braun tree in $O(n)$ time. □

## References

Bird, R. S. (1996) Functional algorithm design. *Science of Computer Programming* **26**(1–3): 15–31.

Braun, W. and Rem, M. (1983) *A logarithmic implementation of flexible arrays.* Memorandum MR83/4. Eindhoven University of Technology.

Hoogerwoord, R. R. (1992) A logarithmic implementation of flexible arrays. *Conference on Mathematics of Program Construction* pp. 191–207.

Paulson, L. C. (1996) *ML for the Working Programmer*, 2nd edition. Cambridge University Press.