FUNCTIONAL PEARL
A fresh look at binary search trees

RALF HINZE
Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
(*e-mail:* ralf@cs.uu.nl)

Alle Abstraktion ist anthropomorphes Zerdenken.
— Oswald Spengler, *Urfragen*

1 Introduction

Binary search trees are old hat, aren't they? Search trees are routinely covered in introductory computer science classes and they are widely used in functional programming courses to illustrate the benefits of algebraic data types and pattern matching. And indeed, the operation of insertion enjoys a succinct and elegant functional formulation. Fig. 1 contains the six-liner given in the language Haskell 98.

Alas, both succinctness and elegance are lost when it comes to implementing the dual operation of deletion, also shown in Fig. 1. Two additional helper functions are required causing the code size to double in comparison with insertion.

Why this discrepancy? The algorithmic explanation is that insertion always takes place at an external node, that is, at a leaf whereas deletion always takes place at an internal node and that manipulating internal nodes is notoriously more difficult than manipulating external nodes.

Our own stab at explaining this phenomenon is algebraic or, if you like, linguistic. Arguably, the data type *Tree* with its two constructors, *Leaf* and *Node*, does not constitute a particularly elegant algebra. If we use binary search trees for representing sets, then *Leaf* denotes the empty set $\emptyset$ and *Node* $l\ a\ r$ denotes the set $s_l \uplus \{a\} \uplus s_r$ where s_l and s_r are the denotations of l and r, respectively. One might reasonably advance that *Node* mingles two abstract operations, namely, forming a singleton set '$\{\cdot\}$' and taking the disjoint union '$\uplus$' of two sets, and that it is preferable to consider these two operations separately.

Of course, there is a good reason for using a ternary constructor: the second argument of *Node*, the split key, is vital for steering the binary search. Thus, as a replacement for the tree constructors the algebra $\emptyset$, $\{\cdot\}$, '$\uplus$' is inadequate; we additionally need a substitute for the split key. Now, a search tree satisfies the invariant that for each node the split key is greater than the elements in the left subtree (and smaller than the ones in the right subtree). This suggests to augment the algebra with an observer function *max* (or *min*, equivalently) that determines the maximum (or the minimum) element of a set. We will see that all standard operations

$$
\begin{array}{ll}
\mathbf{data}\ \mathit{Tree}\ a & = \mathit{Leaf} \mid \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\
\mathit{insert} & :: (\mathit{Ord}\ a) \Rightarrow a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \\
\mathit{insert}\ x\ \mathit{Leaf} & = \mathit{Node}\ \mathit{Leaf}\ x\ \mathit{Leaf} \\
\mathit{insert}\ x\ (\mathit{Node}\ l\ a\ r) & \\
\quad \mid x < a & = \mathit{Node}\ (\mathit{insert}\ x\ l)\ a\ r \\
\quad \mid x == a & = \mathit{Node}\ l\ x\ r \\
\quad \mid x > a & = \mathit{Node}\ l\ a\ (\mathit{insert}\ x\ r) \\
\mathit{delete} & :: (\mathit{Ord}\ a) \Rightarrow a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \\
\mathit{delete}\ x\ \mathit{Leaf} & = \mathit{Leaf} \\
\mathit{delete}\ x\ (\mathit{Node}\ l\ a\ r) & \\
\quad \mid x < a & = \mathit{Node}\ (\mathit{delete}\ x\ l)\ a\ r \\
\quad \mid x == a & = \mathit{join}\ l\ r \\
\quad \mid x > a & = \mathit{Node}\ l\ a\ (\mathit{delete}\ x\ r) \\
\mathit{join} & :: \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \\
\mathit{join}\ \mathit{Leaf}\ r & = r \\
\mathit{join}\ (\mathit{Node}\ \mathit{ll}\ \mathit{la}\ \mathit{lr})\ r & = \mathit{Node}\ l\ m\ r \\
\quad \mathbf{where}\ (l, m) & = \mathit{split\text{-}max}\ \mathit{ll}\ \mathit{la}\ \mathit{lr} \\
\mathit{split\text{-}max} & :: \mathit{Tree}\ a \rightarrow a \rightarrow \mathit{Tree}\ a \rightarrow (\mathit{Tree}\ a, a) \\
\mathit{split\text{-}max}\ l\ a\ \mathit{Leaf} & = (l, a) \\
\mathit{split\text{-}max}\ l\ a\ (\mathit{Node}\ \mathit{rl}\ \mathit{ra}\ \mathit{rr}) & = (\mathit{Node}\ l\ a\ r, m) \\
\quad \mathbf{where}\ (r, m) & = \mathit{split\text{-}max}\ \mathit{rl}\ \mathit{ra}\ \mathit{rr} \\
\mathit{member} & :: (\mathit{Ord}\ a) \Rightarrow a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Bool} \\
\mathit{member}\ x\ \mathit{Leaf} & = \mathit{False} \\
\mathit{member}\ x\ (\mathit{Node}\ l\ a\ r) & \\
\quad \mid x < a & = \mathit{member}\ x\ l \\
\quad \mid x == a & = \mathit{True} \\
\quad \mid x > a & = \mathit{member}\ x\ r
\end{array}
$$

Fig. 1. The standard implementation of binary search trees.

on search trees can be conveniently expressed using this extended algebra. This does not mean, however, that we abandon binary search trees altogether. Rather, we shall use the algebra as an interface to the concrete representation of this data structure. This is the point of the pearl: even concrete data types may benefit from data structural abstraction.

2 An interface to binary search trees

The following signature provides the aforementioned interface to binary search trees. In fact, it can be seen as a declaration of an abstract data type—the choice of names and symbols reflects our intention to use trees for representing sets.

$$
\begin{array}{ll}
\mathbf{data}\ \mathit{Set}\ a & \\
\emptyset & :: (\mathit{Ord}\ a) \Rightarrow \mathit{Set}\ a \\
\{\cdot\} & :: (\mathit{Ord}\ a) \Rightarrow a \rightarrow \mathit{Set}\ a \\
(\uplus) & :: (\mathit{Ord}\ a) \Rightarrow \mathit{Set}\ a \rightarrow \mathit{Set}\ a \rightarrow \mathit{Set}\ a \\
\mathit{max} & :: (\mathit{Ord}\ a) \Rightarrow \mathit{Set}\ a \rightarrow a
\end{array}
$$

The constructor $\emptyset$ denotes the empty set, $\{\cdot\}$ forms a singleton set, and $s_l \uplus s_r$ takes the disjoint union of s_l and s_r under the proviso that the elements in s_l precede the elements in s_r. For each constructor there is a corresponding destructor (typeset with a bar) that can be used in patterns: $\bar{\emptyset}$ matches the empty set, $\bar{\{}\cdot\bar{\}}$ matches singleton sets, and $s_l \bar{\uplus} s_r$ matches sets with at least two elements. In the latter case, we may assume that $\mathit{max}\ s_l < \mathit{min}\ s_r$ and furthermore that both s_l and s_r are non-empty. Thus, the patterns $\bar{\emptyset}$, $\bar{\{}a\bar{\}}$, and $s_l \bar{\uplus} s_r$ are exhaustive and exclusive. The operation *max* is used to determine the maximum element of a non-empty set. We guarantee that all operations, constructors as well as destructors, have a running time that is bounded by a constant.

The signature is asymmetrical in that we provide constant access to the maximum element but not to the minimum element. This will be rectified in Sec. 6. For the moment, we simply note that *min* can be defined as a derived operation—albeit with a running time proportional to the height of a tree.

$$
\begin{array}{lcl}
\mathit{min} & :: & (\mathit{Ord}\ a) \Rightarrow \mathit{Set}\ a \to a \\
\mathit{min}\ \bar{\{}a\bar{\}} & = & a \\
\mathit{min}\ (s_l \bar{\uplus} s_r) & = & \mathit{min}\ s_l
\end{array}
$$

In the second equation we employ the invariants that $\mathit{max}\ s_l < \mathit{min}\ s_r$ and that s_l is non-empty.

At this point the reader may wonder why it is necessary to distinguish between constructors and destructors? First, the constructors will be implemented by Haskell functions and Haskell does not allow functions to appear in patterns. Second, the expression $\emptyset \uplus \{a\}$ must not be equal to the pattern $\bar{\emptyset} \bar{\uplus} \bar{\{}a\bar{\}}$ since we wish to guarantee that both arguments of the destructor '$\bar{\uplus}$' are non-empty. In fact, we have $s_l \uplus s_r = s_l \bar{\uplus} s_r$ if and only if both s_l and s_r are non-empty. On the other hand, $\emptyset = \bar{\emptyset}$ and $\{a\} = \bar{\{}a\bar{\}}$ hold unconditionally.

3 Set functions

Given the above interface we can easily define the standard operations on sets.

Here is how we implement set membership.

$$
\begin{array}{lcl}
\mathit{member} & :: & (\mathit{Ord}\ a) \Rightarrow a \to \mathit{Set}\ a \to \mathit{Bool} \\
\mathit{member}\ x\ \bar{\emptyset} & = & \mathit{False} \\
\mathit{member}\ x\ \bar{\{}a\bar{\}} & = & x == a \\
\mathit{member}\ x\ (s_l \bar{\uplus} s_r) & & \\
\quad \mid x \leqslant \mathit{max}\ s_l & = & \mathit{member}\ x\ s_l \\
\quad \mid \mathit{otherwise} & = & \mathit{member}\ x\ s_r
\end{array}
$$

The recursive structure of the definition is archetypical and nicely illustrates a separation of concerns. Elements of a set are accessed solely through the pattern $\bar{\{}a\bar{\}}$ whereas the pattern $s_l \bar{\uplus} s_r$ in conjunction with the observer function *max* is used for implementing the divide-and-conquer step. In other words, the operations on elements always take place at the fringe of the tree.

Insertion and deletion are now equally simple to implement.

$$
\begin{array}{lcl}
\mathit{insert} & :: & (\mathit{Ord}\ a) \Rightarrow a \to \mathit{Set}\ a \to \mathit{Set}\ a \\
\mathit{insert}\ x\ \overline{\emptyset} & = & \{x\} \\
\mathit{insert}\ x\ \overline{\{}a\overline{\}} & & \\
\quad \mid x < a & = & \{x\} \uplus \{a\} \\
\quad \mid x == a & = & \{x\} \\
\quad \mid x > a & = & \{a\} \uplus \{x\} \\
\mathit{insert}\ x\ (s_l \mathrel{\bar{\uplus}} s_r) & & \\
\quad \mid x \leqslant \mathit{max}\ s_l & = & \mathit{insert}\ x\ s_l \uplus s_r \\
\quad \mid \mathit{otherwise} & = & s_l \uplus \mathit{insert}\ x\ s_r
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{delete} & :: & (\mathit{Ord}\ a) \Rightarrow a \to \mathit{Set}\ a \to \mathit{Set}\ a \\
\mathit{delete}\ x\ \overline{\emptyset} & = & \emptyset \\
\mathit{delete}\ x\ \overline{\{}a\overline{\}} & & \\
\quad \mid x == a & = & \emptyset \\
\quad \mid \mathit{otherwise} & = & \{a\} \\
\mathit{delete}\ x\ (s_l \mathrel{\bar{\uplus}} s_r) & & \\
\quad \mid x \leqslant \mathit{max}\ s_l & = & \mathit{delete}\ x\ s_l \uplus s_r \\
\quad \mid \mathit{otherwise} & = & s_l \uplus \mathit{delete}\ x\ s_r
\end{array}
$$

Note that the two functions differ in the treatment of the base cases only.

The definition of *delete* can be slightly simplified for the special case that we remove the maximum element.

$$
\begin{array}{lcl}
\mathit{delete\text{-}max} & :: & (\mathit{Ord}\ a) \Rightarrow \mathit{Set}\ a \to \mathit{Set}\ a \\
\mathit{delete\text{-}max}\ \overline{\emptyset} & = & \emptyset \\
\mathit{delete\text{-}max}\ \overline{\{}a\overline{\}} & = & \emptyset \\
\mathit{delete\text{-}max}\ (s_l \mathrel{\bar{\uplus}} s_r) & = & s_l \uplus \mathit{delete\text{-}max}\ s_r
\end{array}
$$

The functions *max* and *delete-max* provide priority queue functionality—except that priority queues are usually bags rather than sets.

4 Implementing the interface

Recall from Sec. 2 that we have to guarantee that none of the operations takes more than a constant number of steps. Clearly, this condition rules out ‘standard’ binary search trees as the underlying data structure: determining the maximum element in a search tree takes time proportional to the length of the right spine. However, this observation suggests that we might meet the desired time bound if we constrain the length of the right spine. We take the simplest approach and restrict ourselves to search trees where the right subtree of the root is empty. The following data declaration makes this restriction explicit.

$$\textbf{data}\ \mathit{Set}\ a\ =\ \mathit{Leaf} \mid \mathit{Root}\ (\mathit{Set}\ a)\ a \mid \mathit{Node}\ (\mathit{Set}\ a)\ a\ (\mathit{Set}\ a)$$

The term $\mathit{Root}\ t_l\ a$ serves as a replacement for the top-level term $\mathit{Node}\ t_l\ a\ \mathit{Leaf}$. Thus, a search tree is either empty or of the form $\mathit{Root}\ t_l\ a$—we insist that *Root* is used only at the top level and that *Node* only appears below a *Root* constructor.

Given this representation it is straightforward to implement the constructors $\emptyset$, $\{\cdot\}$, '$\uplus$' and the observer function *max*.

$$
\begin{aligned}
&\emptyset &&= \mathit{Leaf} \\
&\{a\} &&= \mathit{Root}\ \mathit{Leaf}\ a \\
&\mathit{Leaf} \uplus t_r &&= t_r \\
&t_l \uplus \mathit{Leaf} &&= t_l \\
&\mathit{Root}\ t_l\ a_l \uplus \mathit{Root}\ t_r\ a_r &&= \mathit{Root}\ (\mathit{Node}\ t_l\ a_l\ t_r)\ a_r \\
&\mathit{max}\ (\mathit{Root}\ t\ a) &&= a
\end{aligned}
$$

To implement the destructors $\bar{\emptyset}$, $\bar{\{}\cdot\bar{\}}$, and '$\bar{\uplus}$' we make use of an extension to Haskell 98 called views (Burton *et al.*, 1996; Okasaki, 1998). Briefly, a view allows any type to be viewed as a free data type. A view declaration for a type T consists of an anonymous data type, the view type, and an anonymous function, the view transformation, that shows how to map elements of T to the view type.

$$
\begin{aligned}
&\mathbf{view}\ \mathit{Set}\ a &&= \bar{\emptyset} \mid \bar{\{}a\bar{\}} \mid \mathit{Set}\ a \mathbin{\bar{\uplus}} \mathit{Set}\ a\ \mathbf{where} \\
&\quad \mathit{Leaf} &&\to \bar{\emptyset} \\
&\quad \mathit{Root}\ \mathit{Leaf}\ a &&\to \bar{\{}a\bar{\}} \\
&\quad \mathit{Root}\ (\mathit{Node}\ t_l\ a_l\ t_r)\ a_r &&\to \mathit{Root}\ t_l\ a_l \mathbin{\bar{\uplus}} \mathit{Root}\ t_r\ a_r
\end{aligned}
$$

The view transformation essentially undoes the work of the constructors—it is not the inverse since, for instance, $\emptyset \uplus \{a\}$ matches $\bar{\{}a\bar{\}}$ rather than $s_l \mathbin{\bar{\uplus}} s_r$.

5 Eliminating the abstraction layer

Worried about efficiency? It is a simple exercise in program fusion to eliminate the anonymous view type from the definitions given in Sec. 3—a good optimizing compiler should be able to perform this transformation automatically. However, since the resulting code is instructive from an algorithmic point of view, let us briefly discuss one example.

In general, each of the set functions can be written as a composition of the view transformation and the 'original' function that works on the view type. Since the view type is non-recursive, we can easily fuse the view transformation and the original function. In the case of set membership we obtain the following definition.

$$
\begin{aligned}
&\mathit{member}\ x\ \mathit{Leaf} &&= \mathit{False} \\
&\mathit{member}\ x\ (\mathit{Root}\ \mathit{Leaf}\ a) &&= x == a \\
&\mathit{member}\ x\ (\mathit{Root}\ (\mathit{Node}\ t_l\ a_l\ t_r)\ a_r) && \\
&\quad \mid x \leqslant a_l &&= \mathit{member}\ x\ (\mathit{Root}\ t_l\ a_l) \\
&\quad \mid \mathit{otherwise} &&= \mathit{member}\ x\ (\mathit{Root}\ t_r\ a_r)
\end{aligned}
$$

Note that in both recursive calls *member* is passed a *Root* node that is constructed on the fly. As a simple optimization we avoid building this intermediate term by specializing *member* x (*Root* t a) to *member'* x t a.

$$\begin{array}{lcl}
\mathit{member}\ x\ \mathit{Leaf} & = & \mathit{False} \\
\mathit{member}\ x\ (\mathit{Root}\ t\ a) & = & \mathit{member}'\ x\ t\ a \\
\mathit{member}'\ x\ \mathit{Leaf}\ a & = & x == a \\
\mathit{member}'\ x\ (\mathit{Node}\ t_l\ a_l\ t_r)\ a_r & & \\
\quad \mid x \leqslant a_l & = & \mathit{member}'\ x\ t_l\ a_l \\
\quad \mid \mathit{otherwise} & = & \mathit{member}'\ x\ t_r\ a_r
\end{array}$$

Interestingly, this implementation of *member* closely resembles an algorithm proposed by Andersson (1991). Recall that the standard implementation of set membership shown in Fig. 1 uses one three-way comparison per visited node. The variant of Andersson manages with one two-way comparison by keeping track of a candidate element that might be equal to the query element. The third argument of *member'* exactly corresponds to this candidate element, which is only checked for equality when a leaf is hit.

6 A more symmetric design

The implementation in Sec. 4 supports a constant time *max* operation but not a constant time *min* operation. In this section we show how to symmetrize the implementation so that both operations can be supported in constant time. This time we deviate slightly from binary search trees.

Currently, the *Root* constructor only contains the maximum element of the represented set. An obvious idea is to add a third field to the constructor which contains the minimum. This, however, implies that we can no longer represent singleton sets—unless we are willing to allow both fields to contain the same element. Instead, we introduce a new unary constructor that forms a singleton.

Of course, we have to make sure that we can still take the disjoint union of two sets in constant time. This is easily done if one of the sets is empty or both are singletons. In the latter case, we construct a new *Root* node with an empty subtree. Now, assume that both arguments are of the form *Root* a_l t a_r. In this case, we have to form an internal tree using two subtrees and *two* split keys. Similarly, if one of the arguments is a singleton and the other one has *Root* as the topmost constructor, then we have to build a tree using *one* subtree and one split key. For each of the three cases, we invent a tailor-made constructor:

$$\begin{array}{lcl}
\mathbf{data}\ \mathit{Set}\ a & = & \mathit{Leaf} \\
 & \mid & \mathit{Single}\ a \\
 & \mid & \mathit{Root}\ a\ (\mathit{Set}\ a)\ a \\
 & \mid & \mathit{Cons}\ a\ (\mathit{Set}\ a) \\
 & \mid & \mathit{Snoc}\ (\mathit{Set}\ a)\ a \\
 & \mid & \mathit{Node}\ (\mathit{Set}\ a)\ a\ a\ (\mathit{Set}\ a).
\end{array}$$

We insist that *Single* and *Root* are only used at the top level and that *Cons*, *Snoc*, and *Node* only appear below a *Root* node. Given this data structure the implementation of the interface is straightforward (Fig. 2 lists the code).

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Set}\ a & = & \mathit{Leaf} \\
 & \mid & \mathit{Single}\ a \\
 & \mid & \mathit{Root}\ a\ (\mathit{Set}\ a)\ a \\
 & \mid & \mathit{Cons}\ a\ (\mathit{Set}\ a) \\
 & \mid & \mathit{Snoc}\ (\mathit{Set}\ a)\ a \\
 & \mid & \mathit{Node}\ (\mathit{Set}\ a)\ a\ a\ (\mathit{Set}\ a)
\end{array}
$$

$$
\begin{array}{lcl}
\emptyset & = & \mathit{Leaf} \\
\{a\} & = & \mathit{Single}\ a \\
\mathit{Leaf} \uplus t' & = & t' \\
t \uplus \mathit{Leaf} & = & t \\
\mathit{Single}\ a \uplus \mathit{Single}\ a' & = & \mathit{Root}\ a\ \mathit{Leaf}\ a' \\
\mathit{Single}\ a \uplus \mathit{Root}\ a'_l\ t'\ a'_r & = & \mathit{Root}\ a\ (\mathit{Cons}\ a'_l\ t')\ a'_r \\
\mathit{Root}\ a_l\ t\ a_r \uplus \mathit{Single}\ a' & = & \mathit{Root}\ a_l\ (\mathit{Snoc}\ t\ a_r)\ a' \\
\mathit{Root}\ a_l\ t\ a_r \uplus \mathit{Root}\ a'_l\ t'\ a'_r & = & \mathit{Root}\ a_l\ (\mathit{Node}\ t\ a_r\ a'_l\ t')\ a'_r \\
\mathit{max}\ (\mathit{Single}\ a) & = & a \\
\mathit{max}\ (\mathit{Root}\ a_l\ t\ a_r) & = & a_r \\
\mathit{min}\ (\mathit{Single}\ a) & = & a \\
\mathit{min}\ (\mathit{Root}\ a_l\ t\ a_r) & = & a_l \\
\mathbf{view}\ \mathit{Set}\ a & = & \bar{\emptyset} \mid \bar{\{}a\bar{\}} \mid \mathit{Set}\ a \mathbin{\bar{\uplus}} \mathit{Set}\ a\ \mathbf{where} \\
\quad \mathit{Leaf} & \rightarrow & \bar{\emptyset} \\
\quad \mathit{Single}\ a & \rightarrow & \bar{\{}a\bar{\}} \\
\quad \mathit{Root}\ a\ \mathit{Leaf}\ a' & \rightarrow & \mathit{Single}\ a \mathbin{\bar{\uplus}} \mathit{Single}\ a' \\
\quad \mathit{Root}\ a\ (\mathit{Cons}\ a'_l\ t')\ a'_r & \rightarrow & \mathit{Single}\ a \mathbin{\bar{\uplus}} \mathit{Root}\ a'_l\ t'\ a'_r \\
\quad \mathit{Root}\ a_l\ (\mathit{Snoc}\ t\ a_r)\ a' & \rightarrow & \mathit{Root}\ a_l\ t\ a_r \mathbin{\bar{\uplus}} \mathit{Single}\ a' \\
\quad \mathit{Root}\ a_l\ (\mathit{Node}\ t\ a_r\ a'_l\ t')\ a'_r & \rightarrow & \mathit{Root}\ a_l\ t\ a_r \mathbin{\bar{\uplus}} \mathit{Root}\ a'_l\ t'\ a'_r
\end{array}
$$

Fig. 2. An implementation supporting constant time *min*- and *max*-operations.

This variation nicely illustrates the merits of abstraction. Since the set functions of Sec. 3 only rely on the abstract interface, they happily work with the new implementation. Another interesting variation is to augment the implementation of Sec. 4 by a balancing scheme. An extension along this line is described in Hinze (2001) albeit for the more elaborate data structure of priority search queues. All in all, a refreshing view on an old data structure.

References

Andersson, A. (1991) A note on searching in a binary search tree. *Software — Practice and Experience* **21**(10):1125–1128.

Burton, W., Meijer, E., Sansom, P., Thompson, S. and Wadler, P. (1996) *Views: An Extension to Haskell Pattern Matching.* Available from `http://www.haskell.org/development/views.html`.

Hinze, R. (2001) A simple implementation technique for priority search queues. Leroy, X. (ed), *Proceedings of the 2001 International Conference on Functional Programming, Firenze, Italy, September 3-5, 2001* pp. 110–121.

Okasaki, C. (1998) Views for Standard ML. *The 1998 ACM SIGPLAN Workshop on ML, Baltimore, Maryland* pp. 14–23.