FUNCTIONAL PEARLS

On Constructing 2-3 Trees

RALF HINZE
Institute for Computing and Information Sciences
Radboud University, 6525EC Nijmegen, The Netherlands
(*e-mail:* ralf@cs.ru.nl)

1 Introduction

There are only 10 types of people in the world:
those who understand binary, and those who don't.

In computer science, we are so accustomed to thinking about binary numbers, that we sometimes forget that other bases are possible.

Purely Functional Data Structures—Chris Okasaki

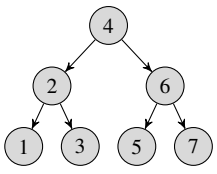
This pearl considers the task of constructing 2-3 trees.

Hopcroft's 2-3 trees (1974) were among the first data structures to be given a proper functional treatment. Hoffmann and O'Donnell's adaption (1982) was strikingly elegant, not the least because it was far removed from the intricate pointer manipulations of the original: the balancing operations were captured by simple equational rewrite rules.

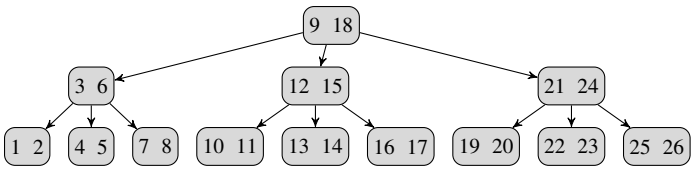
2-3 trees are an interesting blend of binary and ternary trees:

```
data Tree23 a
  = L                                        — leaf
  | N2 (Tree23 a) a (Tree23 a)               — 2-node
  | N3 (Tree23 a) a (Tree23 a) a (Tree23 a)  — 3-node
```

An element of *Tree23 t* is called a 2-3 tree if all leaves appear on the same level (*height condition*). Two extreme instances of 2-3 trees are full binary trees of size $2^h - 1$ and full ternary trees of size $3^h - 1$, where h is the height of the tree in each case. For example:



$7 = 2^3 - 1$



$26 = 3^3 - 1$

Given a sequence of elements we seek to build a 2-3 tree—in linear time—that contains the elements in symmetric order:

$$inorder \cdot build = id$$

where the inorder traversal of a 2-3 tree is defined:

$$\begin{aligned}
&\mathit{inorder} :: \mathit{Tree23}\ a \to [a] \\
&\mathit{inorder}\ L &&= [\,] \\
&\mathit{inorder}\ (N_2\ t\ a\ u) &&= \mathit{inorder}\ t +\!\!+ [a] +\!\!+ \mathit{inorder}\ u \\
&\mathit{inorder}\ (N_3\ t\ a\ u\ b\ v) &&= \mathit{inorder}\ t +\!\!+ [a] +\!\!+ \mathit{inorder}\ u +\!\!+ [b] +\!\!+ \mathit{inorder}\ v
\end{aligned}$$

The function *inorder* is surjective—there is at least one 2-3 tree of a given size—but not injective. In other words, the specification is ambiguous. For example, there are two trees of size 4 and two trees of size 5:

(1)

The larger the size, the more numerous the choices: for example, there are 727 trees of size 20. But which one to pick? 2-3 trees with a greater number of 3-nodes are usually preferred as they are of lower height and have a more favourable memory footprint. We consider both extremes: *slim* trees, which contain the minimal number of 3-nodes for a given size, and *fat* trees, which feature a maximal number of 3-nodes.

We discuss three approaches for defining *build*: top-down, bottom-up, and incremental. The incremental approach is more flexible than the other two in that it allows us to interleave the construction work with other operations, for example, queries.

2 Top-down Construction

A straightforward approach to constructing a 2-3 tree is to add one element after the other, starting with an empty tree: *foldr add L*. Alas, since *add* has to traverse the left spine of the 2-3 tree from the root to the leftmost leaf, the overall running time is $\Theta(n \log n)$. Perhaps a divide-and-conquer scheme is more efficient?

If the input list is empty, we return an empty 2-3 tree. Otherwise, we put one element aside for the root node, divide the remaining list into two halves (of roughly equal size), recursively construct 2-3 trees, and finally combine the resulting trees and the put-aside element. To avoid dealing with the tedious details of this binary sub-division scheme, let us assume that the input is given by a *Braun tree* (Braun & Rem, 1983), rather than a list. (A functional idiom at work: a control structure is turned into a data structure, whose fold corresponds to the control structure.) Recall that a Braun tree is binary tree:

$$\textbf{data}\ \mathit{Braun}\ a = \mathit{Empty} \mid \mathit{Node}\ (\mathit{Braun}\ a)\ a\ (\mathit{Braun}\ a)$$

whose shape is uniquely determined by its size. For any given node *Node l a r*, the left subtree l is either exactly the same size as the right subtree r, or one element smaller: $\mathit{size}\ l \leqslant \mathit{size}\ r \leqslant \mathit{size}\ l + 1$ (making "of roughly equal size" precise).

The challenge in constructing a 2-3 tree is to ensure that the height condition is met. To this end, we annotate each intermediate 2-3 tree with its height, in the hope that the builder can be implemented using a simple tree fold.

$$\begin{aligned}
&\textbf{type}\ \mathit{Height} = \mathit{Int} \\
&\mathit{create} :: \mathit{Braun}\ a \to (\mathit{Height}, \mathit{Tree23}\ a)
\end{aligned}$$

$$
\begin{aligned}
&\mathit{create}\ \mathit{Empty} &&= \mathit{empty}\\
&\mathit{create}\ (\mathit{Node}\ l\ a\ r) &&= \mathit{node}\ (\mathit{create}\ l)\ a\ (\mathit{create}\ r)\\
&\mathit{empty} :: (\mathit{Height}, \mathit{Tree23}\ a)\\
&\mathit{empty} &&= (0, L)
\end{aligned}
$$

The smart constructor *node* creates a 2-node if both argument trees have the same height. Otherwise, it makes the shorter tree the child of the taller one, creating a 3-node:

$$
\begin{aligned}
&\mathit{node} :: (\mathit{Height}, \mathit{Tree23}\ a) \to a \to (\mathit{Height}, \mathit{Tree23}\ a) \to (\mathit{Height}, \mathit{Tree23}\ a)\\
&\mathit{node}\ (h,t)\ a\ (k,u) &&\mid h == k &&= (h+1, N_2\ t\ a\ u)\\
&\mathit{node}\ (h,t)\ a\ (k, N_2\ u\ b\ v) &&\mid h+1 == k &&= (h+1, N_3\ t\ a\ u\ b\ v)
\end{aligned}
$$

While the definition of *node* is simple—deceptively simple—it is not immediately clear that it works. It seems to rely on three assumptions each of which requires justification: (i) the height difference is at most one; (ii) the left tree, first argument of *node*, is at most as tall as the right tree, third argument, or, more generally, smaller trees are shorter; (iii) in case of a height difference, the root of the right tree is necessarily a 2-node.

To address the first two points, let us express the height of the 2-3 tree in terms of the input size n. If $n = 0$, *create* returns *empty*, so the height is 0. Otherwise, *create* calls *node*. Inspecting its definition we observe that the height of the resulting tree is only determined by the height of the first argument, the *left* tree t: if $n = 2 \times m + 1 = (m) + 1 + (m)$ or $n = 2 \times m + 2 = (m) + 1 + (m+1)$, the height is one plus the height of t of size m.

$$
\begin{aligned}
&\mathit{height}\ 0 &&= 0\\
&\mathit{height}\ (2 \times m + 1) &&= 1 + \mathit{height}\ m\\
&\mathit{height}\ (2 \times m + 2) &&= 1 + \mathit{height}\ m
\end{aligned}
$$

The definition of *height* shows that the height h is given by the length of the binary representation of the input size n, *composed of the digits* 1 *and* 2:

$$n = (d_0 \ldots d_{h-1})_2 = \sum_{i=0}^{h-1} d_i \times 2^i \qquad \text{with } d_i \in \{1,2\}$$

Quite reassuringly, this observation confirms (i) and (ii). For (iii) note that constructed trees of size n and $n+1$ have different height if and only if the binary representations of n and $n+1$ have different lengths, if the binary increment of n features a cascading carry. Perhaps, it is useful to remind ourselves how to count in the $\{1,2\}$-binary number system:

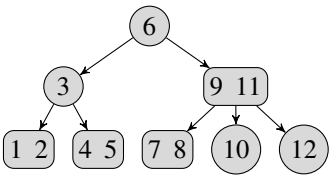
$$()_2,\ (1)_2,\ (2)_2,\ (11)_2,\ (21)_2,\ (12)_2,\ (22)_2,\ (111)_2,\ (211)_2,\ (121)_2,\ (221)_2,\ \ldots$$

The definition of the binary increment, $()_2 + 1 = (1)_2$, $(1\alpha)_2 + 1 = (2\alpha)_2$, and $(2\alpha)_2 + 1 = (1(\alpha+1))_2$, shows that a cascading carry takes a sequence of 2s to a sequence of 1s. Now, a tree of size $(1^{\{h\}})_2 = 2^h - 1$ is a full *binary* tree as always the first equation of *node* matches—the notation $d^{\{n\}}$ is shorthand for the digit d repeated n times. Consequently, the root of the right tree in the second equation of *node* is indeed necessarily a 2-node.

The top-down construction of 2-3 trees combines unfolding and folding Braun trees:

$$
\begin{aligned}
&\mathit{top\text{-}down} :: [a] \to \mathit{Tree23}\ a\\
&\mathit{top\text{-}down} = \mathit{snd} \cdot \mathit{create} \cdot \mathit{braun}
\end{aligned}
$$

where *braun* constructs a Braun tree in linear time, see Appendix A. To illustrate, for the input list $[1..12]$ of size $12 = (212)_2$ the builder creates:



$$\begin{array}{c} 12 \\ = \\ (5)+1+(6) \\ = \\ ((2)+1+(2))+1+((2)+1+(3)) \end{array}$$

The running time of *top-down* is clearly linear. Sometimes, we can do even better than this:

Exercise 1 *Show how to construct a 2-3 tree that contains n copies of the* same *element in time* $\Theta(\log n)$. Hint: *adapt the algorithm of Okasaki (1997) to 2-3 trees.*

Exercise 2 *The divide-and-conquer scheme above puts an element aside for the root node. A simpler scheme divides the input sequence into two halves of roughly the same size and then joins the recursively constructed 2-3 trees. Fill in the details.*

3 Bottom-up Construction of 2-3 Trees

3.1 Slim Trees

The height condition of 2-3 trees dictates that all leaves appear on the same level, suggesting, in fact, a bottom-up approach: a 2-3 tree is created level by level starting at the bottom. We represent a level by an alternating sequence of 2-3 trees and elements,

$$t_0 \mathbin{:\!\diagup} a_1 \mathbin{:\!\diagdown} t_1 \mathbin{:\!\diagup} a_2 \mathbin{:\!\diagdown} t_2 \cdots t_{n-1} \mathbin{:\!\diagup} a_n \mathbin{:\!\diagdown} t_n \square$$

and require that all trees have the same height (*height condition*). The symbols have been chosen to resemble the graphical rendering of trees, with $\square$ marking the end of the alternating sequence. The builder repeatedly passes over this sequence combining adjacent 2-3 trees and elements until the sequence consists of a single 2-3 tree.

To represent alternating sequences we introduce two general-purpose datatypes:

$$\begin{array}{l} \textbf{data}\ a \curlyvee b = a\square \mid a \mathbin{:\!\diagup} (a \curlywedge b) \\ \textbf{data}\ a \curlywedge b = b \mathbin{:\!\diagdown} (a \curlyvee b) \end{array}$$

The datatype $a \curlyvee b$ captures sequences constrained by the regular expression $a(ba)^* = a\epsilon \mid aba(ba)^*$ with the helper type $a \curlywedge b$ modelling $ba(ba)^*$.[1]

The function *bottom* creates the bottom layer from the given list of elements:

$$\begin{array}{l} \mathit{bottom} :: [a] \to \mathit{Tree23}\ a \curlyvee a \\ \mathit{bottom}\ [\,] \quad\ = L\square \\ \mathit{bottom}\ (a : x) = L \mathbin{:\!\diagup} a \mathbin{:\!\diagdown} \mathit{bottom}\ x \end{array}$$

The implementation of a single pass is mostly straightforward:

[1] The helper type is only needed since Haskell does not support distfix notation. However, we have allowed ourselves the liberty of using postfix notation for $\square$.

$$
\begin{aligned}
&\mathit{pass\text{-}slim} :: \mathit{Tree23}\ a \curlyvee a \to \mathit{Tree23}\ a \curlyvee a \\
&\mathit{pass\text{-}slim}\ (t :\!\!\swarrow a :\!\!\searrow u\,\Box) &&= N_2\ t\ a\ u\,\Box \\
&\mathit{pass\text{-}slim}\ (t :\!\!\swarrow a :\!\!\searrow u :\!\!\swarrow b :\!\!\searrow v\,\Box) &&= N_3\ t\ a\ u\ b\ v\,\Box \\
&\mathit{pass\text{-}slim}\ (t :\!\!\swarrow a :\!\!\searrow u :\!\!\swarrow b :\!\!\searrow v :\!\!\swarrow c :\!\!\searrow q) &&= N_2\ t\ a\ u :\!\!\swarrow b :\!\!\searrow \mathit{pass\text{-}slim}\ (v :\!\!\swarrow c :\!\!\searrow q)
\end{aligned}
$$

We only have to be careful that the sequence passed to the recursive call contains at least two trees. In particular, we cannot turn $t :\!\!\swarrow a :\!\!\searrow u :\!\!\swarrow b :\!\!\searrow v\,\Box$ into $N_2\ t\ a\ u :\!\!\swarrow b :\!\!\searrow v\,\Box$ as this would violate the height condition.

The function *pass-slim* favours 2-nodes over 3-nodes so that each level of the resulting 2-3 tree contains at most one 3-node. We shall see later (Section 4.1) that the trees constructed are actually slim in that they contain the minimal number of 3-nodes. Ternary nodes are, of course, unavoidable as a full binary tree necessarily contains $2^h - 1$ elements.

The function *loop-slim* repeatedly invokes *pass-slim* until the sequence consists of a single 2-3 tree. Each pass roughly halves the length of the alternating sequence, resulting in a linear running time overall.

$$
\begin{aligned}
&\mathit{loop\text{-}slim} :: \mathit{Tree23}\ a \curlyvee a \to \mathit{Tree23}\ a \\
&\mathit{loop\text{-}slim}\ (t\,\Box) &&= t \\
&\mathit{loop\text{-}slim}\ (t :\!\!\swarrow a :\!\!\searrow q) &&= \mathit{loop\text{-}slim}\ (\mathit{pass\text{-}slim}\ (t :\!\!\swarrow a :\!\!\searrow q)) \\
&\mathit{bottom\text{-}up\text{-}slim} :: [a] \to \mathit{Tree23}\ a \\
&\mathit{bottom\text{-}up\text{-}slim} = \mathit{loop\text{-}slim} \cdot \mathit{bottom}
\end{aligned}
$$

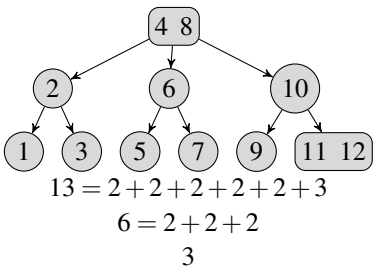
3.2 Fat Trees

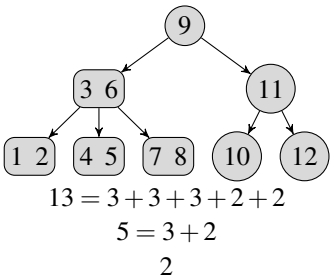
An attractive feature of the bottom-up approach is that it can be easily modified to build fat 2-3 trees featuring a maximal number of 3-nodes. It suffices to adapt the function *pass-slim*:

$$
\begin{aligned}
&\mathit{pass\text{-}fat}\ (t :\!\!\swarrow a :\!\!\searrow u\,\Box) &&= N_2\ t\ a\ u\,\Box \\
&\mathit{pass\text{-}fat}\ (t :\!\!\swarrow a :\!\!\searrow u :\!\!\swarrow b :\!\!\searrow v\,\Box) &&= N_3\ t\ a\ u\ b\ v\,\Box \\
&\mathit{pass\text{-}fat}\ (t :\!\!\swarrow a :\!\!\searrow u :\!\!\swarrow b :\!\!\searrow v :\!\!\swarrow c :\!\!\searrow w\,\Box) &&= N_2\ t\ a\ u :\!\!\swarrow b :\!\!\searrow N_2\ v\ c\ w\,\Box \\
&\mathit{pass\text{-}fat}\ (t :\!\!\swarrow a :\!\!\searrow u :\!\!\swarrow b :\!\!\searrow v :\!\!\swarrow c :\!\!\searrow w :\!\!\swarrow d :\!\!\searrow q) &&= N_3\ t\ a\ u\ b\ v :\!\!\swarrow c :\!\!\searrow \mathit{pass\text{-}fat}\ (w :\!\!\swarrow d :\!\!\searrow q)
\end{aligned}
$$

Again, we are are careful to pass at least two trees to the recursive call. Observe that each level now contains at most two 2-nodes. Of course, it is not at all clear that this ensures that the resulting 2-3 tree is fat. We defer the proof of this fact until Section 4.2.

For example, a slim 2-3 tree of size twelve contains eight 2-nodes and two 3-nodes, while a fat tree consists of four 2-nodes and four 3-nodes:



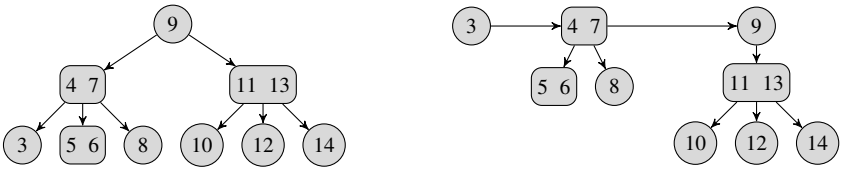


Exercise 3 *Does the top-down approach create slim or fat 2-3 trees?*

4 Incremental Construction of 2-3 Trees

4.1 Slim Trees

We have discarded the implementation of *build* as a fold, *foldr add L*, because *add* necessarily has to traverse the left spine of the 2-3 tree from the root to the leftmost leaf, resulting in a overall super-linear running-time. We can, however, improve the performance if we reverse the left spine exposing the "hot spot", the leftmost leaf. The diagrams below illustrate the transformation:



(2)

The horizontal rendering of the *left-spine view* on the right emphasizes that the spine amounts to a sequence of *pennants*: a 1-pennant consists of a 2-3 tree with an additional element on top; likewise, a 2-pennant consists of two 2-3 trees topped with two elements.

The datatype for the left-spine view is similar to the type of 2-3 trees except that for each node the "downward pointer" to the leftmost child has been replaced by an "upward pointer" to the parent.

```
data Spine23 a
  = Nil                                        — empty spine
  | D1 a (Tree23 a)              (Spine23 a)   — 1-pennant plus tail of spine
  | D2 a (Tree23 a) a (Tree23 a) (Spine23 a)   — 2-pennant plus tail of spine
```

An element of *Spine23 t* is called a 2-3 spine if the height of the *i*-th pennant is *i*, assuming indexing starts from 1 (*height condition*).

To convert a tree to the left-spine view, we additionally pass in the "pointer" to the parent:

```
to-spine :: Tree23 a → Spine23 a → Spine23 a
to-spine L             z = z
to-spine (N2 t a u)    z = to-spine t (D1 a u     z)
to-spine (N3 t a u b v) z = to-spine t (D2 a u b v z)
```

Conversely, to turn a spine into a tree we supply a "pointer" to the leftmost child:

```
from-spine :: Tree23 a → Spine23 a → Tree23 a
from-spine t Nil             = t
from-spine t (D1 a u     z) = from-spine (N2 t a u)     z
from-spine t (D2 a u b v z) = from-spine (N3 t a u b v) z
```

Notice that the second and third equation are obtained from the corresponding equations of *to-spine* by swapping left- and right-hand sides, renaming *to-spine* to *from-spine*. Our 2-3 trees and 2-3 spines are in one-to-one correspondence, with $\lambda\, t \to$ *to-spine t Nil* and $\lambda\, z \to$ *from-spine L z* witnessing the isomorphism. In particular, the conversions "translate" the height conditions.

Exercise 4 *Establish the isomorphism.* Hint: *prove the following generalizations:*

$$\mathit{from\text{-}spine}\ L\ (\mathit{to\text{-}spine}\ t\ z) = \mathit{from\text{-}spine}\ t\ z$$
$$\mathit{to\text{-}spine}\ (\mathit{from\text{-}spine}\ t\ z)\ \mathit{Nil} = \mathit{to\text{-}spine}\ t\ z$$

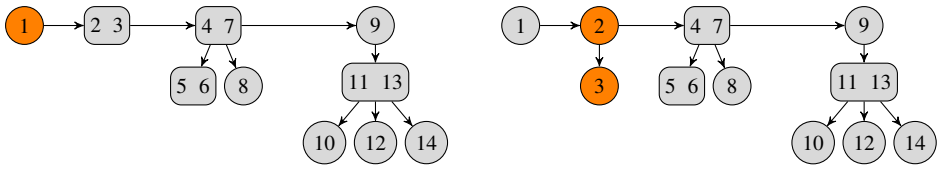
Remark 1 *The one-to-one correspondence allows us to easily port algorithms on 2-3 trees to the left-spine view. For example, assuming search structures, membership and insertion on 2-3 spines are defined:*

$$\begin{aligned}&\mathit{member}' :: (\mathit{Ord}\ a) \Rightarrow a \to \mathit{Spine23}\ a \to \mathit{Bool}\\ &\mathit{member}'\ a\ z = \mathit{member}\ a\ (\mathit{from\text{-}spine}\ L\ z)\end{aligned}$$

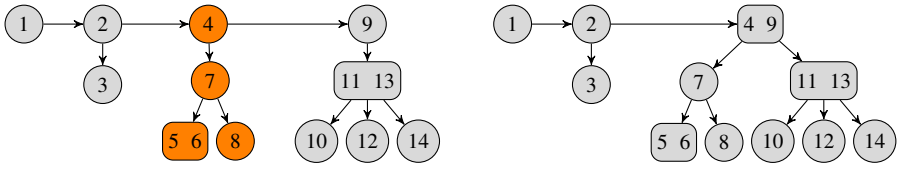
$$\begin{aligned}&\mathit{insert}' :: (\mathit{Ord}\ a) \Rightarrow a \to \mathit{Spine23}\ a \to \mathit{Spine23}\ a\\ &\mathit{insert}'\ a\ z = \mathit{to\text{-}spine}\ (\mathit{insert}\ a\ (\mathit{from\text{-}spine}\ L\ z))\ \mathit{Nil}\end{aligned}$$

where $\mathit{member} :: (\mathit{Ord}\ a) \Rightarrow a \to \mathit{Tree23}\ a \to \mathit{Bool}$ *and* $\mathit{insert} :: (\mathit{Ord}\ a) \Rightarrow a \to \mathit{Tree23}\ a \to \mathit{Tree23}\ a$ *are the corresponding operations on 2-3 trees. Since the conversion functions run in* $\Theta(\log n)$ *time, the logarithmic running-time of membership and insertion is preserved.*

Now, our goal is to construct a spine by adding one element after the other, starting with the empty spine: *foldr cons Nil*. The name *cons* has been chosen to reflect the fact that each element is added to the front. To see how we might implement *cons*, let us go through an example: consider adding first 2 and then 1 to the 12-element spine drawn above (2). Adding 2 is easy: we simply turn the initial 2-node into a 3-node. Things become interesting then, as we cannot further grow the 3-node. To make progress, we generalize the task from adding an element to adding a pennant, highlighted in orange (colour printing) or dark grey (grey-scale printing) below.



We keep the 1-pennant created for 1 and turn the 2-pennant 2 3 into a 1-pennant, to be added to the next "level". Since the next tree on the spine is again a 2-pennant, we repeat the manoeuvre: we keep the to-be-inserted 1-pennant and turn the 2-pennant with root 4 7 into a 1-pennant.



The next tree is finally a 1-pennant which can be grown into a 2-pennant.

The code is a direct rendering of the description above:

$$\begin{aligned}&\mathit{cons} :: a \to \mathit{Spine23}\ a \to \mathit{Spine23}\ a\\ &\mathit{cons}\ a\ z = c_1\ a\ L\ z\end{aligned}$$

where the "carry" function c_1 adds a 1-pennant (highlighted above):

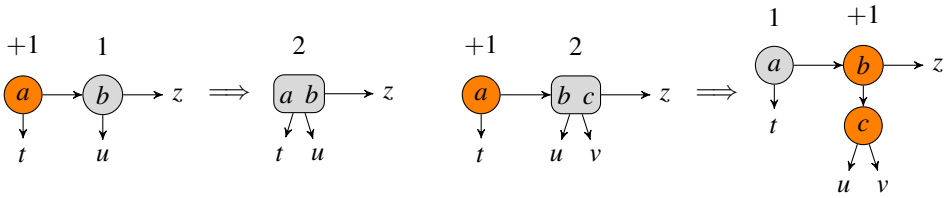
$$
\begin{aligned}
&c_1 :: a \to \mathit{Tree23}\ a \to \mathit{Spine23}\ a \to \mathit{Spine23}\ a\\
&c_1\ a\ t\ \mathit{Nil} &&= D_1\ a\ t\ \mathit{Nil}\\
&c_1\ a\ t\ (D_1\ b\ u\quad z) &&= D_2\ a\ t\ b\ u\ z\\
&c_1\ a\ t\ (D_2\ b\ u\ c\ v\ z) &&= D_1\ a\ t\ (c_1\ b\ (N_2\ u\ c\ v)\ z) \quad \text{— new 2-node}
\end{aligned}
$$

The function maintains the invariant that the height of the 2-3 tree (second argument) is the same as the height of the first tree hanging off the 2-3 spine (third argument).

To construct a 2-3 tree, we first build a spine and then convert it to a tree:

$$
\begin{aligned}
&\mathit{incremental\text{-}slim} :: [a] \to \mathit{Tree23}\ a\\
&\mathit{incremental\text{-}slim} = \mathit{from\text{-}spine\ L} \cdot \mathit{foldr\ cons\ Nil}
\end{aligned}
$$

Turning to the analysis, observe that c_1 only creates 2-nodes below the spine:



Thus, if we grow a spine using calls to *cons* only, then the trees hanging off the spine are full binary trees. Consequently, a 1-pennant $D_1\ a\ t$ has size 1×2^h and a 2-pennant $D_2\ a\ t\ b\ u$ has size 2×2^h, where h is the height of t and u. In other words, the shape of the 2-3 spine is determined by the binary representation of its size. The $\{1,2\}$-binary number system makes a second appearance: the 1-pennant $D_1\ a\ t$ corresponds to the digit 1, with a and t witnessing the weight of the digit; likewise, the 2-pennant $D_2\ a\ t\ b\ u$ corresponds to the digit 2. Moreover, consing an element to the front corresponds to incrementing a binary number (see Section 2). Since the binary increment runs in amortized constant time, the overall running time of *incremental-slim* is linear.

We can use the correspondence to the $\{1,2\}$-number system, to show that the builder *incremental-slim*, like *bottom-up-slim*, actually produces a slim 2-3 tree. Since 3-nodes only occur *on* the left spine, not below, the tree constructed has at most one 3-node on each level. (A tree with more than one 3-node on a level cannot be slim, as it can be transformed into one with strictly fewer 3-nodes, using the identity $2 \times 3 = 3 \times 2$, see eg (1).) The distribution of 3-nodes on the different levels is, however, fixed as each natural number has a unique representation in the $\{1,2\}$-number system. Thus, the fattest slim trees, the ones that contain exactly one 3-node on each level, are of size $(2^{\{h\}})_2 = 2 \times (2^h - 1) = 2^{h+1} - 2$. (The slimmest slim trees, those with no 3-nodes, are full binary trees of size $(1^{\{h\}})_2 = 2^h - 1$.)

Remark 2 *The builder is called* incremental *as it is easy to interleave the construction of the 2-3 spine with other operations such as, for example, queries:*

$$
\begin{aligned}
&\textbf{let}\ s_1 = \mathit{foldr\ cons}\ s_0\ x_1\ \textbf{in} \ldots \mathit{member'}\ \texttt{"Lisa"}\ s_1 \ldots\\
&\textbf{let}\ s_2 = \mathit{foldr\ cons}\ s_1\ x_2\ \textbf{in} \ldots \mathit{member'}\ \texttt{"Florian"}\ s_2 \ldots
\end{aligned}
$$

where member′ is defined in Remark 1.

4.2 Fat Trees

Can we use the same approach to create fat 2-3 trees? The first idea that springs to mind is to replace the binary number system by a ternary one, featuring the digits 1, 2, and 3. We extend the spine datatype, adding the digit 3,

```
data Spine234 a
  = Nil
  | D1 a (Tree23 a)                                  (Spine234 a)
  | D2 a (Tree23 a) a (Tree23 a)                     (Spine234 a)
  | D3 a (Tree23 a) a (Tree23 a) a (Tree23 a)        (Spine234 a)
```

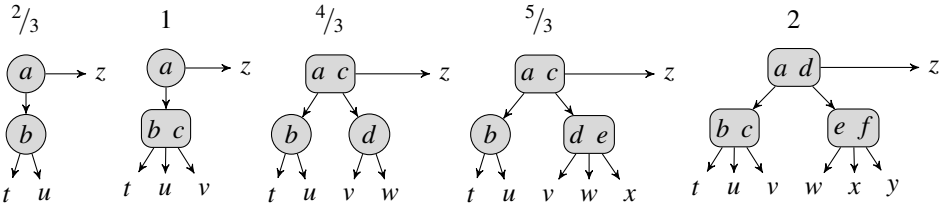
and adapt the carry function c_1:

```
c1 :: a → Tree23 a → Spine234 a → Spine234 a
c1 a t Nil                 = D1 a t Nil
c1 a t (D1 b u         z)  = D2 a t b u z
c1 a t (D2 b u c v     z)  = D3 a t b u c v z
c1 a t (D3 b u c v d w z)  = D1 a t (c1 b (N3 u c v d w) z)    — new 3-node
```

Since the new version of c_1 only creates 3-nodes, the trees hanging off the spine are now full ternary trees. Thus, an i-pennant D_i has size $i \times 3^h$ and consing an element corresponds to the ternary increment.

That was easy—except that it does not work. We cannot convert the 2-3-4 spine to a 2-3 tree: the new digit D_3 has no counterpart as 2-3 trees do not feature 4-nodes. (Of course, the approach works beautifully for 2-3-4 trees.) How to proceed?

Perhaps, we can draw some inspiration from the bottom-up approach? Recall that fat trees contain at most *two* 2-nodes per level, which suggests to consider pennants whose children are 2-nodes. Meet *fractional digits*:



In addition to the whole digits 1 and 2, our revised ternary number system features the fractional digits $^2/_3$, $^4/_3$, and $^5/_3$. Recall that pennants witness *weighted digits*: the size of an r-pennant is $r \times 3^h$ where h is the height of the underlying 2-3 tree. Let us briefly check that the arithmetic works out for the first pennant. The subtrees t and u are full ternary trees of height $h-1$. Thus, the size of the $^2/_3$-pennant is $2 \times (3^{h-1}-1)+2 = {}^2/_3 \times 3^h$ as desired. (As an aside, it turns out that the digit $^4/_3$ is not needed. The *right* child of the pennant $^4/_3$ is a 2-node, which is the *third* node on the level below. But fat trees do not contain more than two 2-nodes per level, see below. For the same reason, we do not consider $^5/_3$-pennants whose *right* child is a 2-node.) We revise the datatype of 2-3 spines accordingly (overloading the constructor names):

```
data Spine23 a
  = Nil
```

$$
\begin{array}{l}
\mid D_{2/3}\ a\ (\mathit{Tree23}\ a)\ (\mathit{Spine23}\ a) \\
\mid D_1\ \ \ a\ (\mathit{Tree23}\ a)\ (\mathit{Spine23}\ a) \\
\mid D_{5/3}\ a\ (\mathit{Tree23}\ a)\ a\ (\mathit{Tree23}\ a)\ (\mathit{Spine23}\ a) \\
\mid D_2\ \ \ a\ (\mathit{Tree23}\ a)\ a\ (\mathit{Tree23}\ a)\ (\mathit{Spine23}\ a)
\end{array}
$$

To eliminate the digit 3 we convert a ternary number whose digits are drawn from $\{1,2,3\}$ to a ternary number with digits in $\{{}^2\!/_3, 1, {}^5\!/_3, 2\}$, using the identity $3 = 1 + {}^2\!/_3 \times 3$:

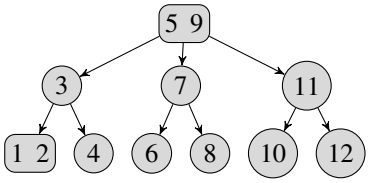
$$
\begin{array}{l}
\mathit{convert} :: \mathit{Spine234}\ a \to \mathit{Spine23}\ a \\
\mathit{convert}\ \mathit{Nil} \qquad\qquad\qquad\quad = \mathit{Nil} \\
\mathit{convert}\ (D_1\ a\ u \qquad\quad z) = D_1\ a\ u \qquad (\mathit{convert} \qquad\quad z) \\
\mathit{convert}\ (D_2\ a\ u\ b\ v \qquad z) = D_2\ a\ u\ b\ v\ (\mathit{convert} \qquad\quad z) \\
\mathit{convert}\ (D_3\ a\ u\ b\ v\ c\ w\ z) = D_1\ a\ u \qquad (c_{2/3}\ b\ (N_2\ v\ c\ w)\ z) \qquad \text{— new 2-node}
\end{array}
$$

$$
\begin{array}{l}
c_{2/3} :: a \to \mathit{Tree23}\ a \to \mathit{Spine234}\ a \to \mathit{Spine23}\ a \\
c_{2/3}\ a\ t\ \mathit{Nil} \qquad\qquad\qquad\quad = D_{2/3}\ a\ t\ \mathit{Nil} \\
c_{2/3}\ a\ t\ (D_1\ b\ u \qquad\quad z) = D_{5/3}\ a\ t\ b\ u\ (\mathit{convert} \qquad\quad z) \\
c_{2/3}\ a\ t\ (D_2\ b\ u\ c\ v \qquad z) = D_{2/3}\ a\ t \qquad (c_{2/3}\ b\ (N_2\ u\ c\ v)\ \ z) \qquad \text{— new 2-node} \\
c_{2/3}\ a\ t\ (D_3\ b\ u\ c\ v\ d\ w \quad z) = D_{5/3}\ a\ t\ b\ u\ (c_{2/3}\ c\ (N_2\ v\ d\ w)\ z) \qquad \text{— new 2-node}
\end{array}
$$

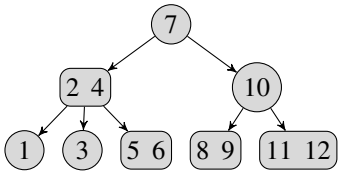
The conversion creates some 2-nodes below the spine. We shall see that the total number of these is actually minimal. Before we tackle the proof, let us put the pieces together:

$$
\begin{array}{l}
\mathit{incremental\text{-}fat} :: [a] \to \mathit{Tree23}\ a \\
\mathit{incremental\text{-}fat} = \mathit{from\text{-}spine}\ L \cdot \mathit{convert} \cdot \mathit{foldr}\ (\lambda a\ z \to c_1\ a\ L\ z)\ \mathit{Nil}
\end{array}
$$

where *from-spine* maps the digits $D_{2/3}$ and D_1 to 2-nodes, and $D_{5/3}$ and D_2 to 3-nodes. The diagrams below show a slim and a fat tree generated by the incremental approach:



$12 = (2\ 1\ 2)_2$



$12 = (3\ 3)_3 = (1\ {}^5\!/_3\ {}^2\!/_3)_3$

We have conjectured that *bottom-up-fat* and *incremental-fat* create fat trees. It is high time to substantiate this claim. Clearly, fat trees contain at most two 2-nodes per level: a tree with more than two 2-nodes on one level cannot be fat, as it can be transformed into one with strictly fewer 2-nodes, using the identity $3 \times 2 = 2 \times 3$, see eg (1). But is the distribution of 2-nodes across the levels also cast in stone? For slim trees we have argued that this is indeed the case as each natural number has a unique representation in the $\{1,2\}$-binary number system. An analogous argument does not work here as the $\{{}^2\!/_3, 1, {}^5\!/_3, 2\}$-ternary number system is redundant. Most naturals have multiple representations, for example, $7 = (12)_3 = (2\,{}^5\!/_3)_3$. In general, the value of a ternary numeral is unchanged if we decrease a digit by 1 and increase the following digit by $\frac{1}{3}$:

$$(\alpha\, a\, b\, \beta)_3 = (\alpha\, (a-1)\, (b+\tfrac{1}{3})\, \beta)_3$$

Interestingly, the corresponding tree rewrites do not change the number of 2-nodes, *unless* the digit $^4/_3$ is involved. For example: $15 = (1\,^5/_3\,1)_3$ has three 2-nodes, while $(1\,^2/_3\,^4/_3)_3$ has five. By contrast, the only legal rewrites in the $\{^2/_3, 1, ^5/_3, 2\}$-ternary system are benign:

$$(\alpha\,^5/_3\,^2/_3\,\beta)_3 = (\alpha\,^2/_3\,1\,\beta)_3 \qquad (\alpha\,^5/_3\,^5/_3\,\beta)_3 = (\alpha\,^2/_3\,2\,\beta)_3 \tag{3a}$$

$$(\alpha\,2\,^2/_3\,\beta)_3 = (\alpha\,1\,1\,\beta)_3 \qquad (\alpha\,2\,^5/_3\,\beta)_3 = (\alpha\,1\,2\,\beta)_3 \tag{3b}$$

We may conclude that the distribution of 2-nodes across the levels is indeed fixed. Of course, within a single level 2-nodes can be freely moved around. Incidentally, applying the rewrites from left to right results in a fat tree where the 2-nodes, if any, are located in the leftmost possible positions on each level, which relates the incremental to the bottom-up approach. If we additionally switch from the left- to the right-spine view, then we obtain exactly the same trees as generated by the bottom-up approach!

For the record, the slimmest fat trees have size $(1\,^2/_3^{\{h\}})_3 = 3^h$—they contain exactly two 2-nodes on each level, except for the topmost, which consists of a single 2-node. (The fattest fat trees, those with no 2-nodes, are full ternary trees of size $(2^{\{h\}})_3 = 3^h - 1$.)

4.3 Fat Trees Revisited

First building a 2-3-4 spine and then turning it into a 2-3 spine is a roundabout way. Also, the approach cannot be characterized as incremental as we cannot (easily) interleave operations on 2-3-4 spines and operations on 2-3 trees. Surely, we can do better than that.

A close inspection of *convert* reveals that the resulting "numerals" satisfy two invariants:

1. the digit $^2/_3$ is never followed by a whole digit;
2. the digit 2 is never followed by a fractional digit.

Since (3a) and (3b) are the only rewrites available, each natural number has a unique representation that satisfies these two invariants. The constraints guide the implementation of *cons* that works directly on 2-3 spines:

$$\begin{aligned}
&\mathit{cons} :: a \to \mathit{Spine23}\ a \to \mathit{Spine23}\ a\\
&\mathit{cons}\ a\ \mathit{Nil} &&= D_1\ a\ L\ \mathit{Nil}\\
&\mathit{cons}\ a\ (D_1\ b\ u\quad z) &&= d_2\ a\ L\ b\ u\ z\\
&\mathit{cons}\ a\ (D_2\ b\ u\ c\ v\ z) &&= D_1\ a\ L\ (c_{2/3}\ b\ (N_2\ u\ c\ v)\ z)
\end{aligned}$$

The smart constructor d_2 establishes the second invariant by implementing the rewrites (3b) from right to left:

$$\begin{aligned}
&d_2 :: a \to \mathit{Tree23}\ a \to a \to \mathit{Tree23}\ a \to \mathit{Spine23}\ a \to \mathit{Spine23}\ a\\
&d_2\ a\ t\ b\ u\ (D_{2/3}\ c\ (N_2\ v\ d\ w)\quad z) &&= D_1\ a\ t\ (D_1\ b\ (N_3\ u\ c\ v\ d\ w)\quad z) && \text{— rm 2-node}\\
&d_2\ a\ t\ b\ u\ (D_{5/3}\ c\ (N_2\ v\ d\ w)\ e\ x\ z) &&= D_1\ a\ t\ (d_2\ b\ (N_3\ u\ c\ v\ d\ w)\ e\ x\ z) && \text{— rm 2-node}\\
&d_2\ a\ t\ b\ u\ z &&= D_2\ a\ t\ b\ u\ z
\end{aligned}$$

The carry $c_{2/3}$ *assumes* that its spine argument does not start with a fractional digit and establishes the first invariant:

$$\begin{aligned}
&c_{2/3} :: a \to \mathit{Tree23}\ a \to \mathit{Spine23}\ a \to \mathit{Spine23}\ a\\
&c_{2/3}\ a\ t\ \mathit{Nil} &&= D_{2/3}\ a\ t\ \mathit{Nil}
\end{aligned}$$

$$c_{2/3}\ a\ t\ (D_1\ b\ u\quad z) = D_{5/3}\ a\ t\ b\ u\ z$$
$$c_{2/3}\ a\ t\ (D_2\ b\ u\ c\ v\ z) = D_{2/3}\ a\ t\ (c_{2/3}\ b\ (N_2\ u\ c\ v)\ z) \quad \text{— new 2-node}$$

The precondition is always met as $c_{2/3}$ is only called on the spine of the digit 2, which by the second invariant is never followed by a fractional digit.

Observe that there are two opposite movements: the carry function $c_{2/3}$ creates 2-nodes below the spine, which are then turned into 3-nodes by the smart constructor d_2.

5 Conclusion

We have discussed three approaches for constructing 2-3 trees: top-down, bottom-up, and incremental. The last two are intimately related: the bottom-up method produces the same trees as the incremental approach based on the right-spine view (modulo some local rearrangements). The spine views are actually interesting data structures in their own right—they can be seen as numerical representations (Okasaki, 1998), container types that are modelled after some number system.

We have used two different number systems, depending on the desired "volume" of the to-be-constructed trees. The $\{{}^2/_3, 1, {}^5/_3, 2\}$-ternary number system is something out of the ordinary, but perhaps you have come across the $\{1,2\}$-binary number system. Speaking of it, we have seen that the system also underlies the top-down construction via Braun trees, see also (Hinze, 2001). Roughly speaking, the top-down and the incremental approach are related by Horner's method for evaluating polynomials. To illustrate, consider the number 11:

$$1\times 2^0 + 2\times 2^1 + 2\times 2^2 = 11 = 1 + 2\times(2 + 2\times(2 + 2\times 0))$$

The formula on the left dictates the shape of the left-spine view: the weight of a digit is witnessed by a pennant of the corresponding size. The formula on the right guides the top-down construction: $1 + 2\times n$ instructs the builder to create two subtrees of the same size; $2 + 2\times n$ indicates that one subtree has to accommodate an additional element.

On a final note, the top-down construction is somewhat unsatisfactory as it creates all kinds of trees, not just slim or fat ones (answering Exercise 3). Perhaps, you can do better?

References

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974) *The design and analysis of computer algorithms*. Addison-Wesley Publishing Company.

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

Hinze, R. (2001) Manufacturing datatypes. *Journal of Functional Programming* **11**(5):493–524.

Hoffmann, C. M. and O'Donnell, M. J. (1982) Programming with equations. *ACM Transactions on Programming Languages and Systems* 83–112.

Okasaki, C. (1997) Functional Pearl: Three algorithms on Braun trees. *Journal of Functional Programming* **7**(6).

Okasaki, C. (1998) *Purely Functional Data Structures*. Cambridge University Press.

A Braun Trees

The function *braun* creates a Braun tree from a given list of elements such that the tree contains the elements in symmetric order.

```
braun :: [a] → Braun a
braun x = fst (make (length x) x)

make :: Int → [a] → (Braun a, [a])
make 0       x     = (Empty,     x)
make (n+1) x     = (Node l a r, z)
  where m        = n div 2
        (l, a : y) = make m       x
        (r, z)     = make (n − m) y
```