Number Systems and Data Structures

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
Email: ralf@informatik.uni-bonn.de
Homepage: http://www.informatik.uni-bonn.de/~ralf

March, 2005

(Pick up the slides at .../~ralf/talks.html#T40.)

An analogy

Natural numbers (aka Peano numerals, unary numbers etc):

$$
\begin{aligned}
&\textbf{data}\ Nat && = Zero \mid Succ\ Nat \\
&plus && :: Nat \rightarrow Nat \rightarrow Nat \\
&plus\ Zero\ n_2 && = n_2 \\
&plus\ (Succ\ n_1)\ n_2 && = Succ\ (plus\ n_1\ n_2)
\end{aligned}
$$

Lists (aka stacks, sequences etc):

$$
\begin{aligned}
&\textbf{data}\ List\ \alpha && = Nil \mid Cons\ \alpha\ (List\ \alpha) \\
&append && :: \forall \alpha\,.\, List\ \alpha \rightarrow List\ \alpha \rightarrow List\ \alpha \\
&append\ Nil\ x_2 && = x_2 \\
&append\ (Cons\ a\ x_1)\ x_2 && = Cons\ a\ (append\ x_1\ x_2)
\end{aligned}
$$

☞ There is a strong analogy between representations of numbers (n) and representations of container objects (of size n).

Numerical representations

☞ Data structures that are designed on the basis of this analogy are called numerical representations.

Idea: the data structure inherits the properties of the number system. The operations on the data structure are modelled after their numerical counterparts.

increment	$n+1$	insertion into a container
decrement	$n-1$	deletion from a container
addition	n_1+n_2	union or merge of two container objects

☞ This design technique is suitable for implementing arbitrary abstractions: sequences, priority queues, sets etc.

☞ The numerical representations we shall introduce are fully persistent: updates never destroy the original data structure.

Numerical representations—continued

arithmetic shift	$n * b$	we shall see
—— " ——	$n \,/\, b$	—— " ——
multiplication	$n * k$	—— " ——
division	$n \,/\, k$	split of a container object
number conversion		construction of a container object
—— " ——		conversion between different container types

History

- Clancy, Knuth, A programming and problem-solving seminar, 1977.
- Guibas, McCreight, Plass, Roberts, A new representation for linear lists, 1977.
- Vuillemin, A data structure for manipulating priority queues, 1978.
- Okasaki, Purely Functional Data Structures, 1998.

☞ Some material is taken from Okasaki's book, which I highly recommend.

Outline of the talk

- Exploring the analogy (7–27)
- A toolbox of number systems (29–41)
- Analysis of data structures (43–51)
- A worked-out example: 2-3 finger trees (53–64)

Random-access lists

Lists are based on the unary number system; random-access lists are based on the binary number system.

$$\begin{aligned}\textbf{data}\ \mathit{Seq}\ \alpha = {} & \mathit{Nil} \\ \mid {} & \mathit{Zero}\ (\quad \mathit{Seq}\ (\alpha, \alpha)) \\ \mid {} & \mathit{One}\ \ (\alpha, \mathit{Seq}\ (\alpha, \alpha))\end{aligned}$$

☞ The type of elements changes from position to position: the top-level possibly contains an element of type α, the next of type (α, α), the next of type $((\alpha, \alpha), (\alpha, \alpha))$ and so on. In other words, $n * 2$ corresponds to pairing.

☞ Seq is an example of a non-regular or nested data type.

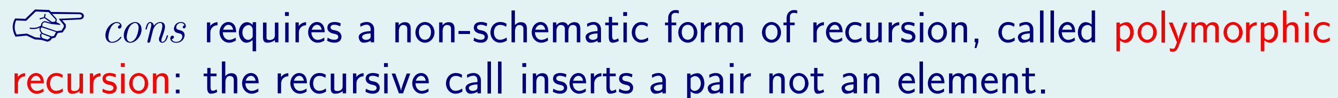
Random-access lists—examples

Nil
$One\ (11, Nil)$
$Zero\ (One\ ((10, 11), Nil))$
$One\ (9, One\ ((10, 11), Nil))$
$Zero\ (Zero\ (One\ (((8, 9), (10, 11)), Nil)))$
$One\ (7, Zero\ (One\ (((8, 9), (10, 11)), Nil)))$
$Zero\ (One\ ((6, 7), One\ (((8, 9), (10, 11)), Nil)))$
$One\ (5, One\ ((6, 7), One\ (((8, 9), (10, 11)), Nil)))$
$Zero\ (Zero\ (Zero\ (One\ ((((4, 5), (6, 7)), ((8, 9), (10, 11))), Nil))))$
$One\ (3, Zero\ (Zero\ (One\ ((((4, 5), (6, 7)), ((8, 9), (10, 11))), Nil))))$
$Zero\ (One\ ((2, 3), Zero\ (One\ ((((4, 5), (6, 7)), ((8, 9), (10, 11))), Nil))))$
$One\ (1, One\ ((2, 3), Zero\ (One\ ((((4, 5), (6, 7)), ((8, 9), (10, 11))), Nil))))$

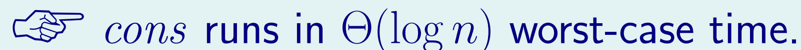
Random-access lists—insertion

Insertion corresponds to binary increment, except that the carry is explicit—the carry is witnessed by a container object of the appropriate size.

$$
\begin{array}{ll}
\mathit{cons} & :: \forall \alpha \,.\, (\alpha, \mathit{Seq}\ \alpha) \rightarrow \mathit{Seq}\ \alpha \\
\mathit{cons}\ (a, \mathit{Nil}) & = \mathit{One}\ (a, \mathit{Nil}) \\
\mathit{cons}\ (a, \mathit{Zero}\ x) & = \mathit{One}\ (a, x) \\
\mathit{cons}\ (a_1, \mathit{One}\ (a_2, x)) & = \mathit{Zero}\ (\mathit{cons}\ ((a_1, a_2), x))
\end{array}
$$



☞ *cons* requires a non-schematic form of recursion, called polymorphic recursion: the recursive call inserts a pair not an element.



☞ *cons* runs in $\Theta(\log n)$ worst-case time.

Random-access lists—deletion

Deletion corresponds to binary decrement, except that the borrow is explicit.

$$
\begin{array}{ll}
\mathit{uncons} & :: \forall\alpha\,.\,\mathit{Seq}\;\alpha \to (\alpha, \mathit{Seq}\;\alpha) \\
\mathit{uncons}\;(\mathit{One}\;(a, \mathit{Nil})) & = (a, \mathit{Nil}) \\
\mathit{uncons}\;(\mathit{One}\;(a, x)) & = (a, \mathit{Zero}\;x) \\
\mathit{uncons}\;(\mathit{Zero}\;x) & = \mathbf{let}\;((a_1, a_2), x) = \mathit{uncons}\;x\;\mathbf{in}\;(a_1, \mathit{One}\;(a_2, x))
\end{array}
$$

☞ uncons is the mirror image of cons.

$$
\begin{array}{ll}
\mathit{cons} & :: \forall\alpha\,.\,(\alpha, \mathit{Seq}\;\alpha) \to \mathit{Seq}\;\alpha \\
\mathit{cons}\;(a, \mathit{Nil}) & = \mathit{One}\;(a, \mathit{Nil}) \\
\mathit{cons}\;(a, \mathit{Zero}\;x) & = \mathit{One}\;(a, x) \\
\mathit{cons}\;(a_1, \mathit{One}\;(a_2, x)) & = \mathit{Zero}\;(\mathit{cons}\;((a_1, a_2), x))
\end{array}
$$

Random-access lists—indexing

Indexing corresponds to ... (well, it's a bit like '$\leqslant$').

$$
\begin{array}{llll}
\mathit{lookup} & & :: & \forall\alpha\,.\,\mathit{Integer} \rightarrow \mathit{Seq}\ \alpha \rightarrow \alpha \\
\mathit{lookup}\ 0 & (\mathit{One}\ (a, x)) & = & a \\
\mathit{lookup}\ (n+1) & (\mathit{One}\ (a, x)) & = & \mathit{lookup}\ n\ (\mathit{Zero}\ x) \\
\mathit{lookup}\ (2*n+0) & (\mathit{Zero}\ x) & = & \mathit{fst}\ \ (\mathit{lookup}\ n\ x) \\
\mathit{lookup}\ (2*n+1) & (\mathit{Zero}\ x) & = & \mathit{snd}\ (\mathit{lookup}\ n\ x)
\end{array}
$$

Random-access lists—construction

Container objects can be constructed in at least two different ways:

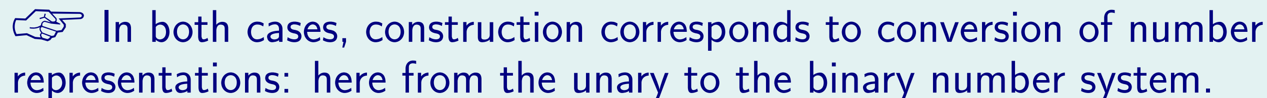
- construct a container object containing n copies of a given element:

$$replicate :: \forall\alpha\,.\,Integer \rightarrow \alpha \rightarrow Seq\ \alpha$$

- construct a container object from a given list of elements:

$$toSeq :: \forall\alpha\,.\,[\alpha] \rightarrow Seq\ \alpha$$

Often, the former operation can be implemented more efficiently.



☞ In both cases, construction corresponds to conversion of number representations: here from the unary to the binary number system.

Conversion of number representations

There are at least two ways to convert a number in one system to the equivalent number in another system:

- use the arithmetic of the target number system; this is sometimes called the expansion method; functions of this type are typically folds.
- use the arithmetic of the source number system; this is sometimes called the multiplication or division method; functions of this type are typically unfolds.

Construction—*replicate*

Using the arithmetic of the target system (unary to binary):

$$\begin{array}{lll} \mathit{replicate} & & :: \forall \alpha \,.\, \mathit{Integer} \to \alpha \to \mathit{Seq}\ \alpha \\ \mathit{replicate}\ 0 & a & = \mathit{Nil} \\ \mathit{replicate}\ (n+1) & a & = \mathit{cons}\ (a, \mathit{replicate}\ n\ a) \end{array}$$

☞ *replicate* runs in $\Theta(n)$ worst-case time; it is not polymorphically recursive.

Construction—*replicate*—continued

Using the arithmetic of the source system (unary to binary):

```
replicate     :: ∀α . Integer → α → Seq α
replicate n a = if n == 0 then Nil
                else case modDiv n 2 of
                     (0, q) → Zero (   replicate q (a, a))
                     (1, q) → One  (a, replicate q (a, a))
```

☞ *replicate* runs in $\Theta(\log n)$ worst-case time; it is polymorphically recursive.

Construction—$toSeq$

Using the arithmetic of the target system (unary to binary):

$$\begin{aligned} &toSeq && :: \forall \alpha \,.\, [\alpha] \rightarrow Seq\ \alpha \\ &toSeq\ [\,] && = Nil \\ &toSeq\ (a : x) && = cons\ (a, toSeq\ x) \end{aligned}$$

☞ $toSeq$ runs in $\Theta(n)$ worst-case time.

☞ $[\alpha]$ is the built-in list data type, which is isomorphic to $List\ \alpha$ (see page 2): $[\alpha] \cong List\ \alpha$, $[\,] \cong Nil$, and $a : x \cong Cons\ a\ x$.

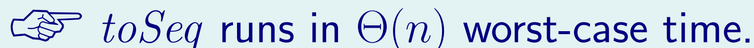
Random-access lists—conversion—continued

Using the arithmetic of the source system (unary to binary):

$$\textbf{data}\ \mathit{Digit}\ \alpha = \mathit{Zero}' \mid \mathit{One}'\ \alpha$$

$$\begin{array}{lll}
\mathit{modDiv2} & :: & [\alpha] \to (\mathit{Digit}\ \alpha, [(\alpha, \alpha)]) \\
\mathit{modDiv2}\ [\,] & = & (\mathit{Zero}', [\,]) \\
\mathit{modDiv2}\ (a : x) & = & \textbf{case}\ \mathit{modDiv2}\ x\ \textbf{of} \\
 & & \quad (\mathit{Zero}', \quad q) \to (\mathit{One}'\ a, q) \\
 & & \quad (\mathit{One}'\ a', q) \to (\mathit{Zero}', \quad (a, a') : q)
\end{array}$$

$$\begin{array}{l}
\mathit{toSeq} \quad :: \forall \alpha\,.\,[\alpha] \to \mathit{Seq}\ \alpha \\
\mathit{toSeq}\ x = \textbf{if}\ \mathit{null}\ x\ \textbf{then}\ \mathit{Nil} \\
\qquad\qquad\qquad \textbf{else}\ \ \textbf{case}\ \mathit{modDiv2}\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad (\mathit{Zero}', \quad q) \to \mathit{Zero}\ (\quad \mathit{toSeq}\ q) \\
\qquad\qquad\qquad\qquad (\mathit{One}'\ a, q) \to \mathit{One}\ \ (a, \mathit{toSeq}\ q)
\end{array}$$



☞ toSeq runs in $\Theta(n)$ worst-case time.

Exercises

Exercise 1. Implement two versions of

$$\begin{aligned} &size &&:: \forall\alpha\,.\,Seq\ \alpha \longrightarrow Integer \\ &fromSeq &&:: \forall\alpha\,.\,Seq\ \alpha \longrightarrow [\alpha] \end{aligned}$$

and determine the worst-case running times (binary to unary).

1-2 random-access lists

The container object that corresponds to '0' contains no elements. This is wasteful!

☞ Interestingly, we can also use the digits 1 and 2 instead of 0 and 1 (the base is still 2).

$$\begin{array}{lll} \textbf{data}\ \mathit{Seq}\ \alpha = & & \mathit{Nil} \\ & \mid & \mathit{One}\ (\alpha, \quad \mathit{Seq}\ (\alpha, \alpha)) \\ & \mid & \mathit{Two}\ ((\alpha, \alpha), \mathit{Seq}\ (\alpha, \alpha)) \end{array}$$

☞ Each number has a unique representation in this system; this is a so-called zeroless number system.

1-2 random-access lists—examples

Nil
$One\ (11, Nil)$
$Two\ ((10, 11), Nil)$
$One\ (9, One\ ((10, 11), Nil))$
$Two\ ((8, 9), One\ ((10, 11), Nil))$
$One\ (7, Two\ (((8, 9), (10, 11)), Nil))$
$Two\ ((6, 7), Two\ (((8, 9), (10, 11)), Nil))$
$One\ (5, One\ ((6, 7), One\ (((8, 9), (10, 11)), Nil)))$
$Two\ ((4, 5), One\ ((6, 7), One\ (((8, 9), (10, 11)), Nil)))$
$One\ (3, Two\ (((4, 5), (6, 7)), One\ (((8, 9), (10, 11)), Nil)))$
$Two\ ((2, 3), Two\ (((4, 5), (6, 7)), One\ (((8, 9), (10, 11)), Nil)))$
$One\ (1, One\ ((2, 3), Two\ ((((4, 5), (6, 7)), ((8, 9), (10, 11))), Nil)))$

1-2 random-access lists—insertion

$$
\begin{aligned}
&\mathit{cons} &&:: \forall \alpha \,.\, (\alpha, \mathit{Seq}\ \alpha) \to \mathit{Seq}\ \alpha \\
&\mathit{cons}\ (a, \mathit{Nil}) &&= \mathit{One}\ (a, \mathit{Nil}) \\
&\mathit{cons}\ (a_1, \mathit{One}\ (a_2, x)) &&= \mathit{Two}\ ((a_1, a_2), x) \\
&\mathit{cons}\ (a_1, \mathit{Two}\ ((a_2, a_3), x)) &&= \mathit{One}\ (a_1, \mathit{cons}\ ((a_2, a_3), x))
\end{aligned}
$$

1-2 random-access lists—deletion

Again, *uncons* is the mirror image of *cons*.

$$
\begin{aligned}
&uncons && :: \forall \alpha \,.\, Seq\ \alpha \rightarrow (\alpha, Seq\ \alpha) \\
&uncons\ (One\ (a, Nil)) && = (a, Nil) \\
&uncons\ (Two\ ((a_1, a_2), x)) && = (a_1, One\ (a_2, x)) \\
&uncons\ (One\ (a_1, x)) && = (a_1, Two\ (uncons\ x))
\end{aligned}
$$

☞ The term $Two\ (uncons\ x)$ corresponds to an arithmetic shift ($n * 2$):

$$
\begin{aligned}
&zero \quad :: \forall \alpha \,.\, Seq\ (\alpha, \alpha) \rightarrow Seq\ \alpha \\
&zero\ x = Two\ (uncons\ x)
\end{aligned}
$$

1-2 random-access lists—conversion

Using $zero$ we can easily convert a 0-1 random-access list into a 1-2 random-access list.

$$
\begin{aligned}
&toSeq && :: \forall\alpha\,.\,Seq_{01}\ \alpha \rightarrow Seq\ \alpha\\
&toSeq\ Nil_{01} && = Nil\\
&toSeq\ (Zero_{01}\ x) && = zero\ (toSeq\ x)\\
&toSeq\ (One_{01}\ (a, x)) && = One\ (a, toSeq\ x)
\end{aligned}
$$

Exercises

Exercise 2. Implement a version of $toSeq$ that uses the arithmetic of the source number system.

Exercise 3. Re-implement indexing for 1-2 random-access lists and show that it has a running time of $\Theta(\log i)$ rather than $\Theta(\log n)$.

0-1-2 random-access lists

If considered in isolation, insertion and deletion both have an amortised running time of $\Theta(1)$. If '*cons*' and '*uncons*' are mixed, however, the running time degrades to $\Theta(\log n)$. Consider:

$$decr\ (incr\ (222222))$$

This can be remedied using a redundant number system, which employs, for instance, the digits 0, 1, and 2 (or alternatively the digits 1, 2, and 3):

$$\begin{array}{lll} \textbf{data}\ Seq\ \alpha = & & Nil \\ & \mid & Zero\ (\qquad\quad Seq\ (\alpha, \alpha)) \\ & \mid & One\ (\alpha, \qquad Seq\ (\alpha, \alpha)) \\ & \mid & Two\ ((\alpha, \alpha), Seq\ (\alpha, \alpha)) \end{array}$$

0-1-2 random-access lists—insertion

We classify 0 and 2 as dangerous and 1 as safe; '$cons$' and '$uncons$' recurse on dangerous digits, but always leave a safe digit behind, so that the next operation to reach that digit will not propagate.

$$
\begin{array}{ll}
cons & :: \forall \alpha \,.\, (\alpha, Seq\ \alpha) \to Seq\ \alpha \\
cons\ (a, Nil) & = One\ (a, Nil) \\
cons\ (a, Zero\ x) & = One\ (a, x) \\
cons\ (a_1, One\ (a_2, x)) & = Two\ ((a_1, a_2), x) \\
cons\ (a_1, Two\ ((a_2, a_3), x)) & = One\ (a_1, cons\ ((a_2, a_3), x))
\end{array}
$$

0-1-2 random-access lists—deletion

$$
\begin{array}{ll}
\mathit{uncons} & :: \forall \alpha \,.\, \mathit{Seq}\ \alpha \rightarrow (\alpha, \mathit{Seq}\ \alpha) \\
\mathit{uncons}\ (\mathit{One}\ (a, \mathit{Nil})) & = (a, \mathit{Nil}) \\
\mathit{uncons}\ (\mathit{One}\ (a, x)) & = (a, \mathit{Zero}\ x) \\
\mathit{uncons}\ (\mathit{Two}\ ((a_1, a_2), x)) & = (a_1, \mathit{One}\ (a_2, x)) \\
\mathit{uncons}\ (\mathit{Zero}\ x) & \\
\quad = \textbf{let}\ ((a_1, a_2), x) = \mathit{uncons}\ x\ \textbf{in}\ (a_1, \mathit{One}\ (a_2, x)) &
\end{array}
$$

☞ *uncons* is no longer the mirror image of *cons*.

Outline of the talk

✔ Exploring the analogy (7–27)

✘ A toolbox of number systems (29–41)

✘ Analysis of data structures (43–51)

✘ A worked-out example: 2-3 finger trees (53–64)

Positional number systems

The most common number systems are positional number systems.

$$(d_0 \dots d_{n-1}) = \sum_{i=0}^{n-1} d_i \cdot w_i \text{ with } d_i \in D_i.$$

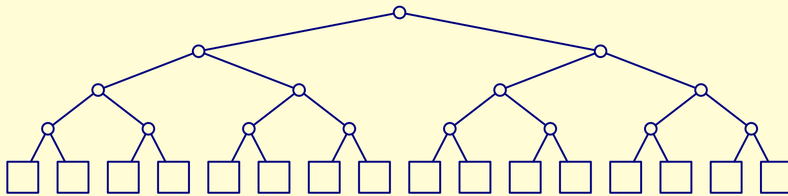
Unary number system: $w_i = 1$ and $D_i = \{1\}$ for all i.

Binary number system: $w_i = 2^i$ and $D_i = \{0, 1\}$ for all i.

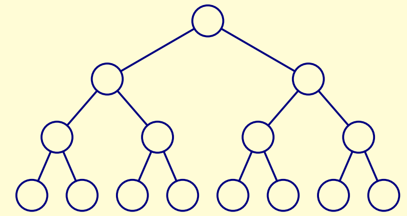
1-2 binary system: $w_i = 2^i$ and $D_i = \{1, 2\}$ for all i.

0-1-2 binary system: $w_i = 2^i$ and $D_i = \{0, 1, 2\}$ for all i.

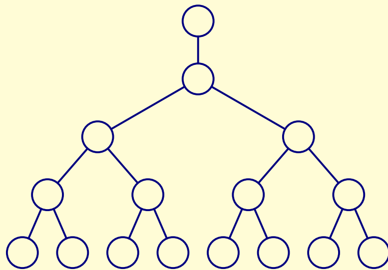
Common building blocks



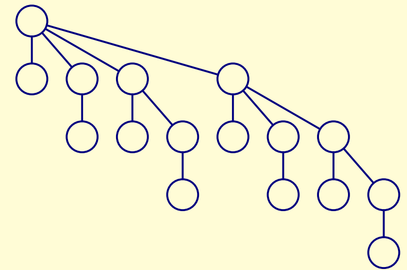
complete leaf tree



complete tree



pennant

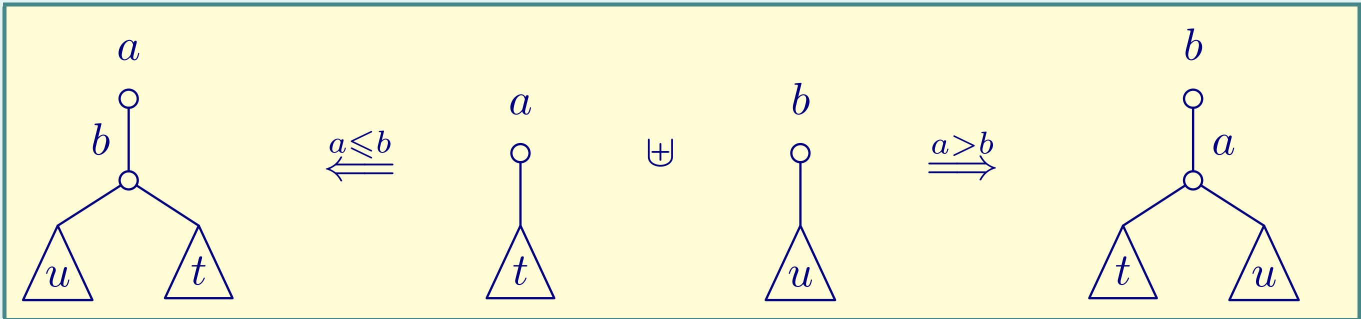


binomial tree

Common abstractions

The most elementary operation is adding two 1s of weight w_i to form a 1 of weight w_{i+1}.

- Sequences: if we use leaf trees, adding two 1s amounts to pairing (adding a new root node), see above.
- Priority queues (bags): we can either use pennants (semi-heaps) or binomial trees (heaps). For the former, adding two 1s is given by



- Sets: adding two 1s is difficult (if not impossible), since the keys in a search tree must be maintained in symmetric order, but see below.

Ternary number systems

Number systems with a larger base are often advantageous, because fewer digits are needed to represent each number. On the downside, processing a single digit may take longer.

$$\begin{aligned}
\textbf{data}\ \mathit{Seq}\ \alpha = {} & \mathit{Nil} \\
\mid\ & \mathit{Zero}\ (\mathit{Seq}\ (\alpha, \alpha, \alpha)) \\
\mid\ & \mathit{One}\ (\alpha, \mathit{Seq}\ (\alpha, \alpha, \alpha)) \\
\mid\ & \mathit{Two}\ (\alpha, \alpha, \mathit{Seq}\ (\alpha, \alpha, \alpha))
\end{aligned}$$

We use triples instead of pairs.

Ternary number systems—insertion

$$
\begin{aligned}
&\mathit{cons} && :: \forall\alpha\,.\,(\alpha, \mathit{Seq}\ \alpha) \rightarrow \mathit{Seq}\ \alpha \\
&\mathit{cons}\ (a, \mathit{Nil}) && = \mathit{One}\ (a, \mathit{Nil}) \\
&\mathit{cons}\ (a, \mathit{Zero}\ x) && = \mathit{One}\ (a, x) \\
&\mathit{cons}\ (a_1, \mathit{One}\ (a_2, x)) && = \mathit{Two}\ (a_1, a_2, x) \\
&\mathit{cons}\ (a_1, \mathit{Two}\ (a_2, a_3, x)) && = \mathit{Zero}\ (\mathit{cons}\ ((a_1, a_2, a_3), x))
\end{aligned}
$$

Fibonacci number system

Fibonacci number system: $w_i = F_i$ and $D_i = \{0, 1\}$ for all i, where

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 2 \\ F_{n+2} &= F_n + F_{n+1} \end{aligned}$$

☞ This number system is redundant: we have, for instance, $(11) = (001)$. However, if we disallow consecutive 1s, then we regain uniqueness.

The fibonacci number system is a lot like the binary number system, except that it involves only addition and subtraction, not multiplication or division by 2.

Fibonacci number system—continued

The following data type captures the invariant that the digit 1 is not followed by a second 1.

$$\begin{aligned}
&\textbf{type}\ \mathit{Seq}\ \alpha && = \mathit{Fib}\ \alpha\ (\alpha, \alpha) \\
&\textbf{data}\ \mathit{Fib}\ \alpha\ \beta && = \mathit{Nil} \\
& && \mid \mathit{Zero}\ (\quad \mathit{Fib}\ \beta\ (\alpha, \beta)) \\
& && \mid \mathit{One}\ (\alpha, \mathit{Fib}\ (\alpha, \beta)\ (\beta, (\alpha, \beta)))
\end{aligned}$$

The smart constructor one restores the invariant:

$$\begin{aligned}
&one && :: \forall \alpha\ \beta\ .\ (\alpha, \mathit{Fib}\ \beta\ (\alpha, \beta)) \rightarrow \mathit{Fib}\ \alpha\ \beta \\
&one\ (a, \mathit{Nil}) && = \mathit{One}\ (a, \mathit{Nil}) \\
&one\ (a, \mathit{Zero}\ x) && = \mathit{One}\ (a, x) \\
&one\ (a, \mathit{One}\ (b, x)) && = \mathit{Zero}\ (\mathit{Zero}\ (one\ ((a, b), x)))
\end{aligned}$$

Fibonacci number system—insertion

Insertion is easy to define using one:

$$
\begin{aligned}
&cons && :: \forall \alpha \,.\, \alpha \rightarrow Seq\ \alpha \rightarrow Seq\ \alpha \\
&cons\ a\ Nil && = One\ (a, Nil) \\
&cons\ a\ (Zero\ x) && = one\ (a, x) \\
&cons\ a\ (One\ (b, x)) && = Zero\ (one\ ((a, b), x))
\end{aligned}
$$

☞ The type $Fib\ \alpha\ \beta$ is a bit inconvenient to work with because it has two type arguments.

Factorial number system

Factorial number system: $w_i = i!$ and $D_i = \{0, \ldots, i\}$ for all i.

☞ Note that $\sum_{i=0}^{n-1} i \cdot i! = n! - 1$.

Application: Given a sequence of elements, generate a random permutation.

- generate a random number r with $0 \leqslant r \leqslant n\,! - 1$;
- convert r into the factorial number system: $(d_0 \ldots d_{n-1})$;
- for $i = 0$ to $n - 1$: exchange elements at positions i and $i + d_{n-1-i}$.

0	1	2	3

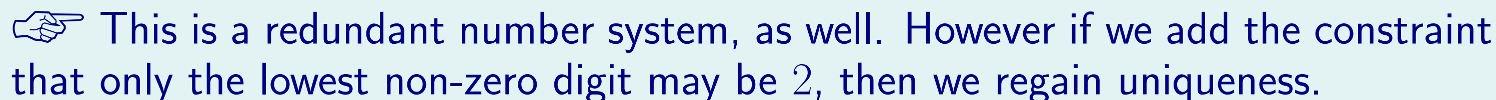
$0 \leftrightarrow 0$ or 1 or 2 or 3
$1 \leftrightarrow 1$ or 2 or 3
$2 \leftrightarrow 2$ or 3
$3 \leftrightarrow 3$

Skew binary number system

Skew binary number system: $w_i = 2^{i+1} - 1$ and $D_i = \{0, 1, 2\}$ for all i.

Why $2^{i+1} - 1$? This is the size of perfect binary trees of height $i + 1$.

Why $D_i = \{0, 1, 2\}$? Using $\{0, 1, 2\}$ we cannot represent all numbers.



This is a redundant number system, as well. However if we add the constraint that only the lowest non-zero digit may be 2, then we regain uniqueness.

Salient properties: increment and decrement work in constant time.

Skew binary number system—continued

$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Leaf}\ \alpha \mid \mathit{Node}\ \alpha\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha)$$

We use a so-called sparse representation to be able to access the lowest non-zero digit in constant time.

$$\textbf{type}\ \mathit{Size} = \mathit{Integer}$$

$$\textbf{data}\ \mathit{Seq}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ (\mathit{Size}, \mathit{Tree}\ \alpha, \mathit{Seq}\ \alpha)$$

$$\begin{array}{l}
\mathit{cons} :: \forall \alpha\,.\, \alpha \rightarrow \mathit{Seq}\ \alpha \rightarrow \mathit{Seq}\ \alpha \\
\mathit{cons}\ a\ (\mathit{Cons}\ (n_1, t_1, \mathit{Cons}\ (n_2, t_2, x))) \\
\quad \mid n_1 == n_2 = \mathit{Cons}\ (1 + n_1 + n_2, \mathit{Node}\ a\ t_1\ t_2, x) \\
\mathit{cons}\ a\ x \qquad = \mathit{Cons}\ (1, \qquad \mathit{Leaf}\ a, \qquad x)
\end{array}$$

More number systems . . .

There is a lot more to discover:

- lazy number systems: good amortised bounds even in a persistent setting;
- segmented number systems: supports constant time deque operations;
- combinatorial number system: fixed number of digits;
- mixed-radix number systems: persistent arrays with sublogarithmic access;
- . . .

Exercises

Exercise 5. Explore the 1-2 fibonacci number system.

Exercise 6. Try to come up with a suitable definition of skew fibonacci numbers.

Outline of the talk

✔ Exploring the analogy (7–27)

✔ A toolbox of number systems (29–41)

✘ Analysis of data structures (43–51)

✘ A worked-out example: 2-3 finger trees (53–64)

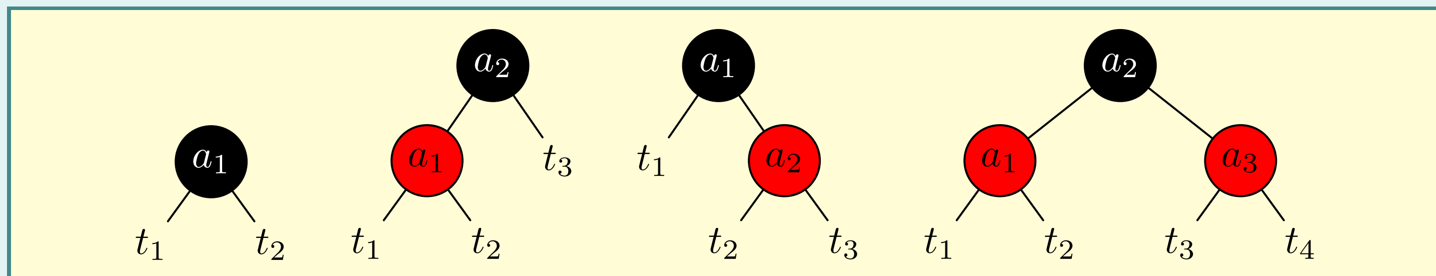
Analysis of red-black trees

Problem: We insert an ascending sequence of elements into an empty red-black tree. Which shape has the final tree?

Recap: red-black trees

Red-black trees were developed by R. Bayer under the name symmetric binary B-trees as binary tree representations of 2-3-4 trees.

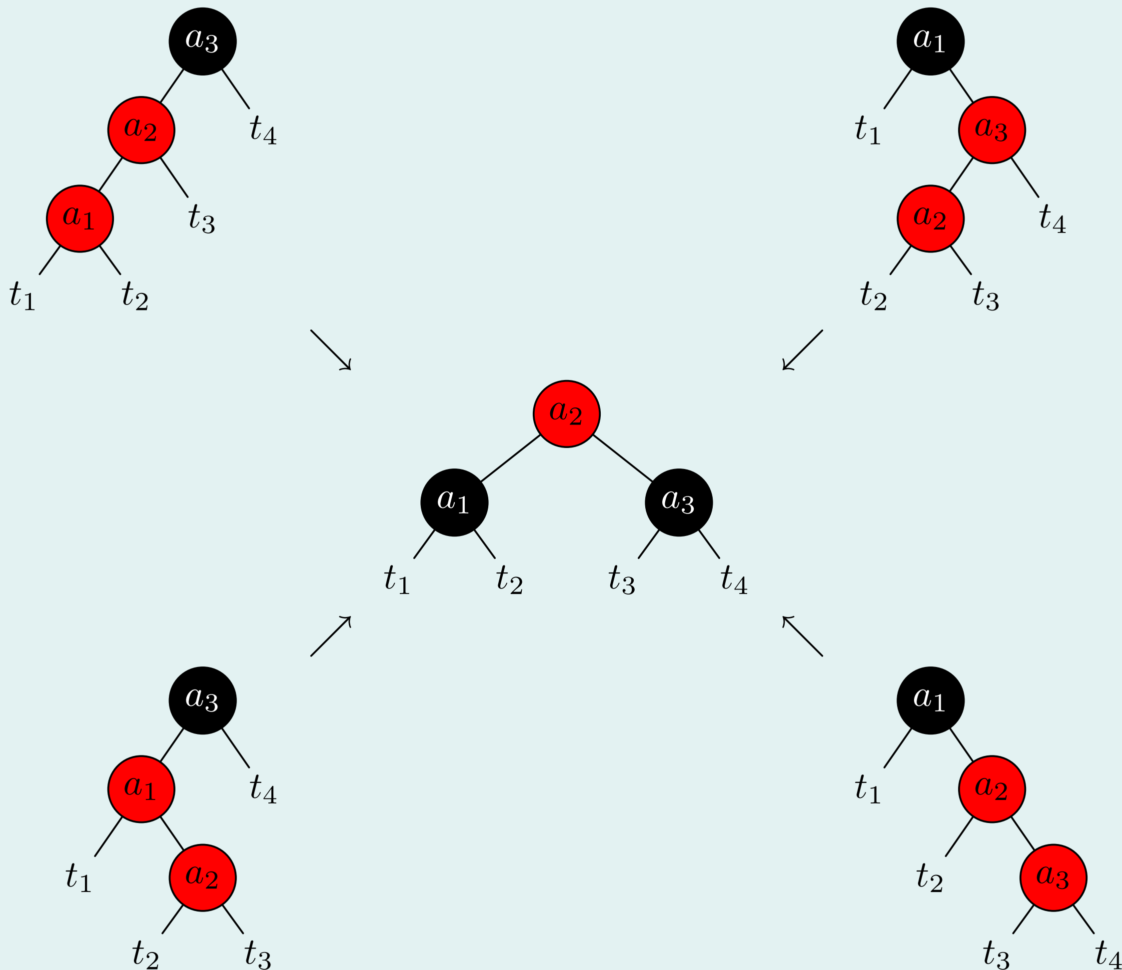
The idea of red-black trees is to represent 3- and 4-nodes by small binary trees, which consist of a black root and one or two auxiliary red children.



Red condition: Each red node has a black parent.

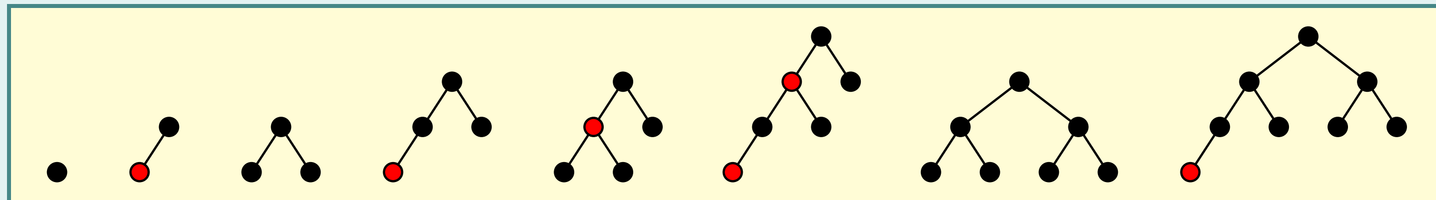
Black condition: Each path from the root to an empty node contains exactly the same number of black nodes (this is called the tree's black height).

Red-black trees: balancing



Examples

The following trees are generated for $1 \leqslant i \leqslant 8$ (a new node is always coloured red).

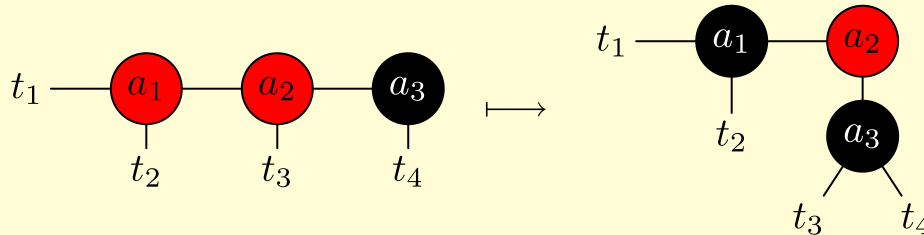


Insertion always traverses the left spine of the tree to the leftmost leaf.

Observations

☞ All the nodes below the left spine are black.

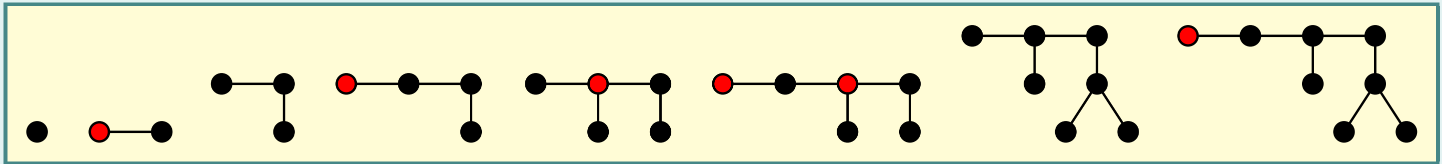
Re-consider the balancing operation—for emphasis we draw the left spine horizontally.



☞ The black condition implies that the trees below the left spine must be perfectly balanced. Thus, the generated red-black trees correspond to sequences of pennants.

Observations—continued

It is helpful to redraw the examples according to the left-spine view.



Let h be the height of the rightmost pennant; the black condition implies that a pennant of height i appears either once or twice for all $0 \leqslant i \leqslant h$.

☞ The generated red-black trees correspond to binary numbers in the 1-2 number system.

Construction of red-black trees

☞ The analogy to the 1-2 number system can be exploited to construct red-black trees from ordered sequences of elements in linear time.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Tree}\ \alpha = \mathit{Empty}\\
&\qquad\qquad\quad \mid \mathit{Red}\ \ (\mathit{Tree}\ \alpha)\ \alpha\ (\mathit{Tree}\ \alpha)\\
&\qquad\qquad\quad \mid \mathit{Black}\ (\mathit{Tree}\ \alpha)\ \alpha\ (\mathit{Tree}\ \alpha)\\
&\\
&\mathbf{data}\ \mathit{Set}\ \alpha\ \ = \mathit{Nil}\\
&\qquad\qquad\quad \mid \mathit{One}\ (\alpha, \mathit{Tree}\ \alpha)\ (\mathit{Set}\ \alpha)\\
&\qquad\qquad\quad \mid \mathit{Two}\ (\alpha, \mathit{Tree}\ \alpha)\ (\alpha, \mathit{Tree}\ \alpha)\ (\mathit{Set}\ \alpha)
\end{aligned}
$$

☞ We do not use a nested data type, so that we can easily transform the left-spine view into an ordinary red-black tree.

Construction of red-black trees—continued

$$
\begin{aligned}
&cons :: \forall\alpha\,.\,(\alpha, Tree\ \alpha) \rightarrow Set\ \alpha \rightarrow Set\ \alpha\\
&cons\ (a, t)\quad Nil \qquad\qquad\quad = One\ (a, t)\ Nil\\
&cons\ (a_1, t_1)\ (One\ (a_2, t_2)\ x) = Two\ (a_1, t_1)\ (a_2, t_2)\ x\\
&cons\ (a_1, t_1)\ (Two\ (a_2, t_2)\ (a_3, t_3)\ x)\\
&\quad = One\ (a_1, t_1)\ (cons\ (a_2, Black\ t_2\ a_3\ t_3)\ x)
\end{aligned}
$$

$$
\begin{aligned}
&insert \qquad :: \forall\alpha\,.\,\alpha \rightarrow Set\ \alpha \rightarrow Set\ \alpha\\
&insert\ a\ x = cons\ (a, Empty)\ x
\end{aligned}
$$

Exercise 4. Repeat the analysis with 1-2 brother trees.

Outline of the talk

✔ Exploring the analogy (7–27)

✔ A toolbox of number systems (29–41)

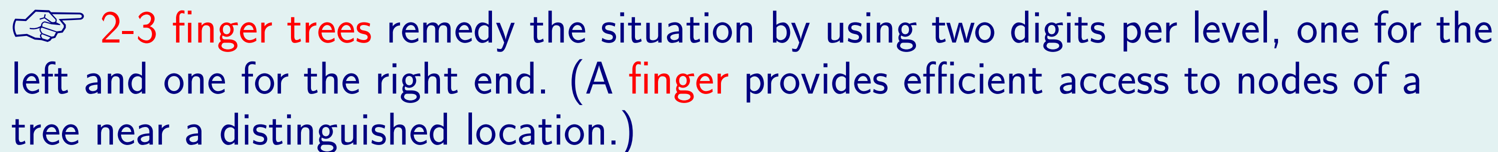
✔ Analysis of data structures (43–51)

✘ A worked-out example: 2-3 finger trees (53–64)

A more symmetric design

All the data structures we have seen so far are asymmetric:

- adding an element to the left end is easy and efficient;
- adding an element to the right end is difficult (if not impossible) and inefficient.



☞ 2-3 finger trees remedy the situation by using two digits per level, one for the left and one for the right end. (A finger provides efficient access to nodes of a tree near a distinguished location.)

2-3 nodes

Finger trees are constructed from 2-3 nodes:

$$\textbf{data}\ \mathit{Node}\ \alpha = \mathit{Node2}\ \alpha\ \alpha \mid \mathit{Node3}\ \alpha\ \alpha\ \alpha$$

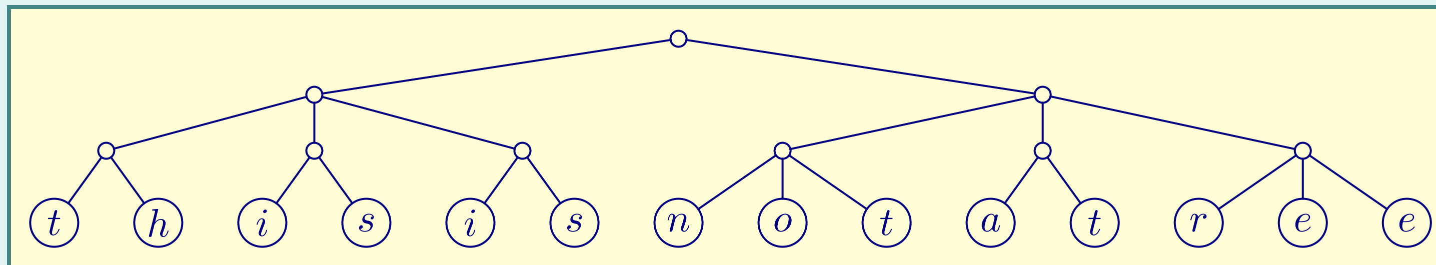
☞ 2-3 nodes contain two or three subtrees, but no keys; all data is stored in the leaves of the tree.

2-3 trees

Conventional 2-3 trees can be defined as follows:

$$\mathbf{data}\ \mathit{Tree}\ \alpha = \mathit{Zero}\ \alpha \mid \mathit{Succ}\ (\mathit{Tree}\ (\mathit{Node}\ \alpha))$$

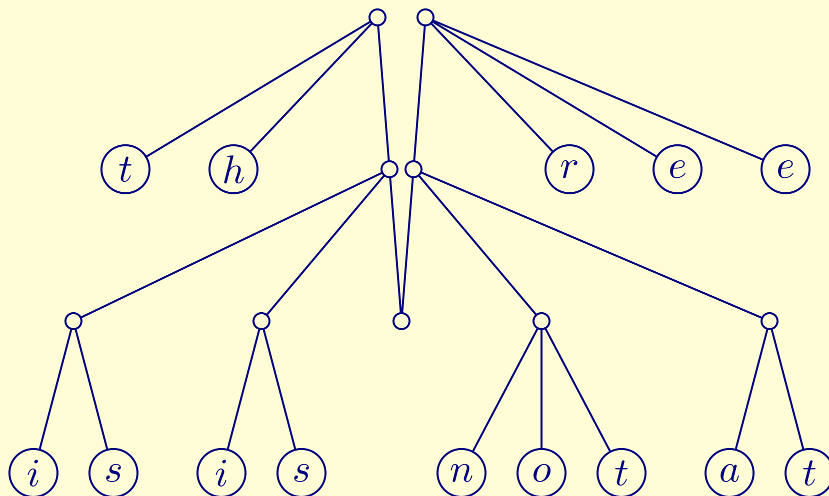
A typical 2-3 tree:



All leaves of a 2-3 tree are at the same depth, the left and right spines have the same length.

2-3 finger trees

If we take hold of the end nodes of a 2-3 tree and lift them up together, we obtain a tree that looks like this:



Each pair of nodes on the central spine is merged into a single node (called *Deep*).

☞ A sequence of n elements is represented by a tree of depth $\Theta(\log n)$; an element d positions from the nearest end is stored at a depth of $\Theta(\log d)$.

2-3 finger trees—continued

Finger trees provide efficient access to both ends of a sequence:

$$\begin{aligned}\mathbf{data}\ \mathit{FingerTree}\ \alpha &= \mathit{Empty}\\ &\mid \mathit{Single}\ \alpha\\ &\mid \mathit{Deep}\ (\mathit{Digit}\ \alpha)\ (\mathit{FingerTree}\ (\mathit{Node}\ \alpha))\ (\mathit{Digit}\ \alpha)\end{aligned}$$

The top level contains elements of type α, the next of type $\mathit{Node}\ \alpha$, and so on: the nth level contains elements of type $\mathit{Node}^n\ \alpha$, namely 2-3 trees of depth n.

A digit is a buffer of one to four elements, represented as a list to simplify the presentation.

$$\mathbf{type}\ \mathit{Digit}\ \alpha = [\alpha]$$

Finger trees—deque operations

Adding a new element to the left of the sequence:

$$
\begin{array}{ll}
(\triangleleft) & :: \alpha \to \mathit{FingerTree}\ \alpha \to \mathit{FingerTree}\ \alpha \\
a \triangleleft \mathit{Empty} & = \mathit{Single}\ a \\
a \triangleleft \mathit{Single}\ b & = \mathit{Deep}\ [a]\ \mathit{Empty}\ [b] \\
a \triangleleft \mathit{Deep}\ [b, c, d, e]\ m\ \mathit{sf} & = \mathit{Deep}\ [a, b]\ (\mathit{Node3}\ c\ d\ e \triangleleft m)\ \mathit{sf} \\
a \triangleleft \mathit{Deep}\ \mathit{pr}\ m\ \mathit{sf} & = \mathit{Deep}\ ([a] +\!\!+\ \mathit{pr})\ m\ \mathit{sf}
\end{array}
$$

Finger trees—deque operations—continued

Adding to the right end of the sequence is the mirror image of '◁':

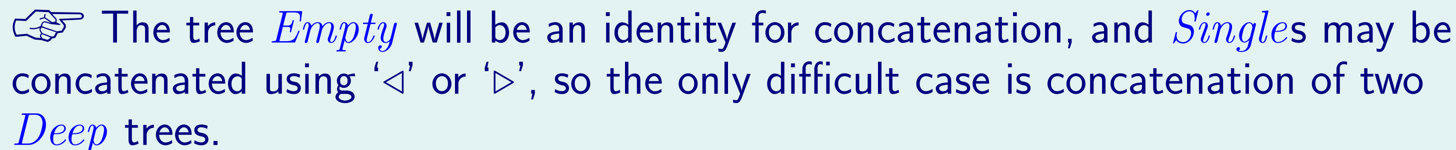
$$
\begin{array}{lll}
(\triangleright) & & :: \textit{FingerTree}\ a \to a \to \textit{FingerTree}\ a \\
\textit{Empty} & \triangleright\ a & = \textit{Single}\ a \\
\textit{Single}\ b & \triangleright\ a & = \textit{Deep}\ [b]\ \textit{Empty}\ [a] \\
\textit{Deep}\ pr\ m\ [e, d, c, b] & \triangleright\ a & = \textit{Deep}\ pr\ (m \triangleright \textit{Node3}\ e\ d\ c)\ [b, a] \\
\textit{Deep}\ pr\ m\ sf & \triangleright\ a & = \textit{Deep}\ pr\ m\ (sf +\!\!+ [a])
\end{array}
$$

We classify digits of two or three elements (which correspond to nodes) as safe, and those of one or four elements as dangerous (cf 0-1-2 random-access lists).

☞ A deque operation may only propagate to the next level from a dangerous digit, but in doing so it makes that digit safe, so that the next operation to reach that digit will not propagate.

Finger trees—concatenation

The use of 2-3 nodes rather than pairs provides enough flexibility so that we can also append two 2-3 finger trees.



☞ The tree *Empty* will be an identity for concatenation, and *Single*s may be concatenated using '$\triangleleft$' or '$\triangleright$', so the only difficult case is concatenation of two *Deep* trees.

Finger trees—concatenation—continued

Concatenation of two Deep trees:

$$\mathit{Deep}\ pr_1\ m_1\ sf_1 \bowtie \mathit{Deep}\ pr_2\ m_2\ sf_2 = \mathit{Deep}\ pr_1\ \ldots\ sf_2$$

We can use the prefix of the first tree as the prefix of the result, and the suffix of the second tree as the suffix of the result.

To combine the rest to make the new middle subtree, we require a function of type

$$\mathit{FingerTree}\ (\mathit{Node}\ \alpha) \to \mathit{Digit}\ \alpha \to \mathit{Digit}\ \alpha \to \mathit{FingerTree}\ (\mathit{Node}\ \alpha) \to \mathit{FingerTree}\ (\mathit{Node}\ \alpha)$$

For simplicity, we combine the two digit arguments into a list of Nodes.

Finger trees—concatenation—continued

$$
\begin{aligned}
&\mathit{app3} :: \mathit{FingerTree}\ \alpha \rightarrow [\alpha] \rightarrow \mathit{FingerTree}\ \alpha \rightarrow \mathit{FingerTree}\ \alpha \\
&\mathit{app3}\ \mathit{Empty}\ \mathit{ts}\ \mathit{xs} \quad = \mathit{ts} \lhd' \mathit{xs} \\
&\mathit{app3}\ \mathit{xs}\ \mathit{ts}\ \mathit{Empty} \quad = \mathit{xs} \rhd' \mathit{ts} \\
&\mathit{app3}\ (\mathit{Single}\ x)\ \mathit{ts}\ \mathit{xs} = x \lhd (\mathit{ts} \lhd' \mathit{xs}) \\
&\mathit{app3}\ \mathit{xs}\ \mathit{ts}\ (\mathit{Single}\ x) = (\mathit{xs} \rhd' \mathit{ts}) \rhd x \\
&\mathit{app3}\ (\mathit{Deep}\ \mathit{pr}_1\ m_1\ \mathit{sf}_1)\ \mathit{ts}\ (\mathit{Deep}\ \mathit{pr}_2\ m_2\ \mathit{sf}_2) \\
&\quad = \mathit{Deep}\ \mathit{pr}_1\ (\mathit{app3}\ m_1\ (\mathit{nodes}\ (\mathit{sf}_1 +\!\!+\ \mathit{ts} +\!\!+\ \mathit{pr}_2))\ m_2)\ \mathit{sf}_2
\end{aligned}
$$

☞ nodes groups a list of at most 12 elements into a list of 2-3 nodes, '$\lhd'$' adds a list of at most 4 elements to the left of a finger tree.

$$
\begin{aligned}
&(\bowtie) \quad :: \mathit{FingerTree}\ \alpha \rightarrow \mathit{FingerTree}\ \alpha \rightarrow \mathit{FingerTree}\ \alpha \\
&\mathit{xs} \bowtie \mathit{ys} = \mathit{app3}\ \mathit{xs}\ [\,]\ \mathit{ys}
\end{aligned}
$$

☞ The recursion terminates when we reach the bottom of the shallower tree, so the total time taken is $\Theta(\log(\min\{n_1, n_2\}))$.

Finger trees—concatenation—continued

$$
\begin{aligned}
&nodes && :: [\alpha] \to [Node\ \alpha] \\
&nodes\ [a, b] && = [Node2\ a\ b] \\
&nodes\ [a, b, c] && = [Node3\ a\ b\ c] \\
&nodes\ [a, b, c, d] && = [Node2\ a\ b, Node2\ c\ d] \\
&nodes\ (a : b : c : xs) && = Node3\ a\ b\ c : nodes\ xs
\end{aligned}
$$

$$
\begin{aligned}
&(\triangleleft') \quad :: [a] \to FingerTree\ a \to FingerTree\ a \\
&x \triangleleft' t = foldr\ (\triangleleft)\ t\ x
\end{aligned}
$$

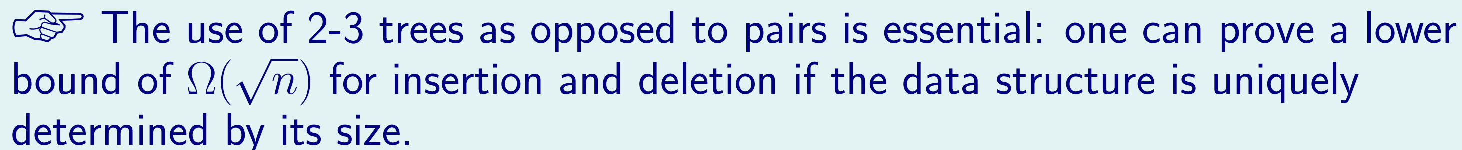
$$
\begin{aligned}
&(\triangleright') \quad :: FingerTree\ a \to [a] \to FingerTree\ a \\
&t \triangleright' x = foldl\ (\triangleright)\ t\ x
\end{aligned}
$$

2-3 finger search trees

2-3 finger trees may also serve as implementations of

- priority queues,
- search trees,
- priority search queues,
- ...

Idea: augment the internal nodes by additional information (size, split key, or both) to steer the search.



☞ The use of 2-3 trees as opposed to pairs is essential: one can prove a lower bound of $\Omega(\sqrt{n})$ for insertion and deletion if the data structure is uniquely determined by its size.

Outline of the talk

- ✔ Exploring the analogy (7–27)
- ✔ A toolbox of number systems (29–41)
- ✔ Analysis of data structures (43–51)
- ✔ A worked-out example: 2-3 finger trees (53–64)

Conclusion

- Number systems serve admirably as templates for data structures.
- Even the most exotic number systems seem to have their uses in the realm of data structures.
- We have only touched the surface of this exciting topic.

Thanks for listening!

Literature

- Snyder, On uniquely represented data structures (extended abstract)., 1977.
- Okasaki, Purely Functional Data Structures, 1998.
- Okasaki, Red-Black Trees in a Functional Setting, 1999.
- Hinze, Constructing red-black trees, 1999.
- Hinze. Bootstrapping One-sided Flexible Arrays, 2002.
- Hinze, Paterson, Finger trees: the swiss army knife of data structures, 2005.