Verified Analysis of Functional Data Structures

with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2017-8-6

Lecture 1

Introduction

1. Motivation
2. Time
3. Binary Trees

General aims

- Verification of correctness *and* complexity of functional data structures
- Algebraic proofs about functions *and* their execution time
- Verified in an interactive theorem prover (Isabelle/HOL)

Complexity analysis should be as algebraic as proving

$$\binom{r}{m}\binom{m}{k} = \binom{r}{k}\binom{r-k}{m-k}$$

Part of a project on Verified Algorithm Analysis

This course

- An introduction to the basic methods
- A whirlwind tour of verified functional data structures (and some proofs)
- Specifically:
  "Classical" search trees and priority queues
- First functional versions:

  Chris Okasaki. *Purely Functional Data Structures*.
- Now: verified algebraic proofs

  See `src/HOL/Data_Structures` in the Isabelle distribution or online

2 Time

A Lightweight Approach

A Monadic Approach

Amortized Analysis

Principle: Count function calls

For every function $f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$

define a *timing function* $t\_f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow nat$:

Translation of defining equations:

$$\frac{e \leadsto e'}{f\ p_1 \ldots p_n = e \ \leadsto\ t\_f\ p_1 \ldots p_n = e' + 1}$$

Translation of expressions:

$$\frac{s_1 \leadsto t_1 \qquad \ldots \qquad s_k \leadsto t_k}{g\ s_1 \ldots s_k \leadsto t_1 + \cdots + t_k + t\_g\ s_1 \ldots s_k}$$

- Variable ⇝ 0, Constant ⇝ 0
- Constructor calls and primitive operations on $bool$ and numbers cost 1

Example

$$app\ []\ ys = ys$$

$$\leadsto$$

$$t\_app\ []\ ys = 0 + 1$$

$$app\ (x\#xs)\ ys = x\ \#\ app\ xs\ ys$$

$$\leadsto$$

$$t\_app\ (x\#xs)\ ys = 0 + (0 + 0 + t\_app\ xs\ ys) + 1 + 1$$

A compact formulation of $e \rightsquigarrow t$

t is the sum of all $t\_g\ s_1\ \dots\ s_k$
such that $g\ s_1\ \dots\ s_n$ is a subterm of e

If g is

- a constructor or
- a predefined function on $bool$ or numbers

then $t\_g\ \dots = 1$.

if and *case*

So far we model a call-by-value semantics

Conditionals and case expressions are evaluated lazily.

Translation:

$$\frac{b \leadsto t \quad s_1 \leadsto t_1 \quad s_2 \leadsto t_2}{\textit{if}\ b\ \textit{then}\ s_1\ \textit{else}\ s_2 \leadsto t + (\textit{if}\ b\ \textit{then}\ t_1\ \textit{else}\ t_2)}$$

Similarly for *case*

$O(.)$ is enough

$\Longrightarrow$ Reduce all additive constants to 1

Example

$t\_app\ (x\#xs)\ ys = t\_app\ xs\ ys + 1$

② Time

- A Lightweight Approach
- A Monadic Approach
- Amortized Analysis

- Start with the definition of a function f_tm that computes both the result and the running time simultaneously
- Define f_tm via a monad that counts function calls behind the scene
- Derive f and t_f from f_tm

The time monad

datatype $'a\ tm = TM\ 'a\ nat$

$val\ (TM\ v\ n) = v$
$time\ (TM\ v\ n) = n$

The programmer must only use:

Bind:
$TM\ u\ m \gg\!= f\ =$
$(\text{let } TM\ v\ n = f\ u \text{ in } TM\ v\ (m{+}n))$

$return\ v = TM\ v\ 0$

Equality plus 1 clock tick: ⇒

Simple example

Using do-notation rather than $\ggg\!=$:

$$app\_tm :: {}'a\ list \Rightarrow {}'a\ list \Rightarrow {}'a\ list\ tm$$

$$
\begin{array}{l}
app\_tm\ []\ ys \Rightarrow return\ ys \\
app\_tm\ (x \# xs)\ ys \Rightarrow do\ \{ \\
\quad zs \leftarrow app\_tm\ xs\ ys; \\
\quad return\ (x \# zs) \\
\}
\end{array}
$$

Simple example

Define

$app\ xs\ ys = val\ (app\_tm\ xs\ ys)$
$t\_app\ xs\ ys = time\ (app\_tm\ xs\ ys)$

Prove automatically:

$app\ []\ ys = []$
$app\ (x\ \#\ xs)\ ys = (\text{let } zs = app\ xs\ ys \text{ in } x\ \#\ zs)$

Prove with (some) insight:

$t\_app\ xs\ ys = length\ xs + 1$

- ② Time
  - A Lightweight Approach
  - A Monadic Approach
  - Amortized Analysis

Recall

Amortized complexity $=$ average complexity of single operation in a sequence of operations, in the worst case

Example

Consider a k-bit binary counter.
An individual increment has complexity $O(k)$.
In a sequence of increments starting from 0,
each increment has amortized constant complexity.

Formalization

Implementation: Data structure of type τ with

- Operation(s) $f :: \tau \Rightarrow \tau$
  (may have additional parameters)
- Initial value: $init :: \tau$
  (function "empty")

Claim: Operation f has a certain amortized complexity of at most $a\_f :: \tau \Rightarrow num$.

(Typically a_f is constant or logarithmic)

Potential method

Find a *potential function* $\Phi :: \tau \Rightarrow num$ and prove

- $\Phi\ init = 0$
- $\Phi\ s \geq 0$
- for each operation f:
  $$\underbrace{t\_f\ s + \Phi(f\ s) - \Phi\ s}_{\text{amortized cost}} \leq a\_f\ s$$

What does this imply?

Amortized and real cost

A sequence of operations f_1, . . . , f_n
induces a sequence of states:

$$s_0 = init,\ \ s_1 = f_1\ s_0,\ \ \ldots,\ \ s_n = f_n\ s_{n-1}$$

The real cost of performing f_1, . . . , f_n
is upper-bounded by amortized costs:

$$\sum_{i=1}^{n} t\_f_i\ s_{i-1} \le \sum_{i=1}^{n} a\_f_i\ s_{i-1}$$

Warning

Amortized analysis is only correct for single threaded uses of a data structure.

Single threaded = no value is used more than once

Otherwise:

$$\begin{aligned} \textit{let}\quad & counter = 0; \\ & bad = \text{increment } counter\, 2^n - 1 \text{ times}; \\ & \_ = incr\ bad; \\ & \_ = incr\ bad; \\ & \_ = incr\ bad; \\ & \vdots \end{aligned}$$

Okasaki's insight

Lazy evaluation can help to design
persistent data structures
with amortized complexity bounds

Not covered in this course

1. Motivation
2. Time
3. Binary Trees

Binary trees

datatype $'a\ tree = Leaf \mid Node\ ('a\ tree)\ 'a\ ('a\ tree)$

Abbreviations:
$$\begin{array}{rcl} \langle\rangle & \equiv & Leaf \\ \langle l,\ a,\ r\rangle & \equiv & Node\ l\ a\ r \end{array}$$

Most of the time: tree = binary tree

Tree traversal

$inorder :: {'a}\ tree \Rightarrow {'a}\ list$

$inorder\ \langle\rangle = []$
$inorder\ \langle l,\ x,\ r\rangle = inorder\ l\ @\ [x]\ @\ inorder\ r$

Size

Number of nodes:

$size :: {}'a\ tree \Rightarrow nat$

$|\langle\rangle| = 0$
$|\langle l,\ \_,\ r\rangle| = |l| + |r| + 1$

Number of leaves:

$size1 :: {}'a\ tree \Rightarrow nat$

$|t|_1 = |t| + 1$

Height

$height :: {}'a\ tree \Rightarrow nat$

$h(\langle\rangle) = 0$

$h(\langle l, \_, r\rangle) = max\ (h(l))\ (h(r)) + 1$

Lemma $|t|_1 \leq 2^{h(t)}$

Lecture 2

Search Trees

❹ Correctness

❺ 2-3 Trees

❻ Red-Black Trees

We assume that the elements in the search trees are linearly ordered

Mechanism: type classes
Implicit constraint: type $'a$ is in class $linorder$

Instead of using $<$ and $\leq$ directly:

datatype $cmp\_val = LT \mid EQ \mid GT$

$cmp :: \ 'a \Rightarrow \ 'a \Rightarrow \ cmp\_val$
$cmp \ x \ y =$
(if $x < y$ then LT else if $x = y$ then EQ else GT)

4. Correctness

5. 2-3 Trees

6. Red-Black Trees

Search trees represent sets

Specification for sets?

Set interface

An implementation of sets of elements of type $'a$ must provide

- An implementation type $'t$
- $empty :: 't$
- $insert :: 'a \Rightarrow 't \Rightarrow 't$
- $delete :: 'a \Rightarrow 't \Rightarrow 't$
- $isin :: \quad 't \Rightarrow 'a \Rightarrow bool$

④ Correctness

The Standard Specification

Correctness via Sorted Lists

Set specification

A correct implementation must also provide

- an *abstraction function* $set :: {}'t \Rightarrow {}'a\ set$
- and an *invariant* $invar :: {}'t \Rightarrow bool$

such that

$$\begin{array}{l} set\ empty = \{\} \\ set\ (insert\ x\ t) = set\ t \cup \{x\} \\ set\ (delete\ x\ t) = set\ t - \{x\} \\ isin\ t\ x = (x \in set\ t) \end{array}$$

under the assumption $invar\ t$

Also: $invar$ must be invariant

$$
\begin{array}{l}
invar\ empty \\
invar\ t \Longrightarrow invar\ (insert\ x\ t) \\
invar\ t \Longrightarrow invar\ (delete\ x\ t)
\end{array}
$$

Example: type $'a\ tree$

Abstraction function: $set\_tree :: {}'a\ tree \Rightarrow {}'a\ set$

$set\_tree\ \langle\rangle = \{\}$
$set\_tree\ \langle l,\ x,\ r\rangle = set\_tree\ l \cup \{x\} \cup set\_tree\ r$

Invariant: $bst :: {}'a\ tree \Rightarrow bool$

$bst\ \langle\rangle = True$
$bst\ \langle l,\ a,\ r\rangle =$
$(bst\ l \wedge$
$bst\ r \wedge$
$(\forall\, x{\in}set\_tree\ l.\ x < a) \wedge$
$(\forall\, x{\in}set\_tree\ r.\ a < x))$

This is the *standard approach*.

Empirical evidence: proofs are hard(er) to automate

④ Correctness

The Standard Specification

Correctness via Sorted Lists

$$sorted :: 'a\ list \Rightarrow bool$$

$$sorted\ [] = True$$
$$sorted\ [x] = True$$
$$sorted\ (x \# y \# zs) = (x < y \wedge sorted\ (y \# zs))$$

No duplicates!

Structural invariant

We assume that there is some structural invariant on the search tree:

$inv : {}'t \Rightarrow bool$

e.g. some balance criterion.

Correctness of $insert$

$$inv\ t \wedge sorted\ (inorder\ t) \Longrightarrow$$
$$inorder\ (insert\ x\ t) = ins\_list\ x\ (inorder\ t)$$

where

$$ins\_list :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$$

inserts an element into a sorted list.

Also covers preservation of bst

Correctness of $delete$

$$inv\ t \wedge sorted\ (inorder\ t) \Longrightarrow$$
$$inorder\ (delete\ x\ t) = del\_list\ x\ (inorder\ t)$$

where

$del\_list :: {'a} \Rightarrow {'a}\ list \Rightarrow {'a}\ list$

deletes an element from a sorted list.

Also covers preservation of bst

Correctness of $isin$

$$inv\ t \wedge sorted\ (inorder\ t) \Longrightarrow$$
$$isin\ t\ x = (x \in elems\ (inorder\ t))$$

where

$$elems :: 'a\ list \Rightarrow 'a\ set$$

converts a list into a set.

Correctness w.r.t. sorted lists
implies the standard set specification
and can be automated for all search trees in this course.

Except for the structural invariants.

Therefore we concentrate on the latter.

For details of the automation see

T. Nipkow. *Automatic Functional Correctness Proofs for Functional Search Trees.* ITP 2016

2-3 Trees

datatype $'a\ tree23 = \langle\rangle$
$\ \ \mid Node2\ ('a\ tree23)\ 'a\ ('a\ tree23)$
$\ \ \mid Node3\ ('a\ tree23)\ 'a\ ('a\ tree23)\ 'a\ ('a\ tree23)$

Abbreviations:

$$\begin{array}{rcl} \langle l,\ a,\ r\rangle & \equiv & Node2\ l\ a\ r \\ \langle l,\ a,\ m,\ b,\ r\rangle & \equiv & Node3\ l\ a\ m\ b\ r \end{array}$$

Invariant bal

All leaves are at the same level:

$$bal\ \langle\rangle\ =\ True$$

$$bal\ \langle l,\ \_,\ r\rangle\ =\ (bal\ l \wedge bal\ r \wedge h(l) = h(r))$$

$$bal\ \langle l,\ \_,\ m,\ \_,\ r\rangle\ =$$
$$(bal\ l \wedge bal\ m \wedge bal\ r \wedge h(l) = h(m) \wedge h(m) = h(r))$$

Lemma

$$bal\ t \Longrightarrow 2^{h(t)} \leq |t| + 1$$

Insertion

The idea:

$$\begin{array}{rcl} \mathit{Leaf} & \rightsquigarrow & \mathit{Node2} \\ \mathit{Node2} & \rightsquigarrow & \mathit{Node3} \\ \mathit{Node3} & \rightsquigarrow & \text{overflow, pass 1 element back up} \end{array}$$

Insertion

Two possible return values:

- tree accommodates new element without increasing height: $T_i\ t$
- tree overflows: $Up_i\ l\ x\ r$

datatype $'a\ up_i = T_i\ ('a\ tree23)$
$\mid Up_i\ ('a\ tree23)\ 'a\ ('a\ tree23)$

$tree_i :: 'a\ up_i \Rightarrow 'a\ tree23$

$tree_i\ (T_i\ t) = t$
$tree_i\ (Up_i\ l\ a\ r) = \langle l,\ a,\ r\rangle$

Insertion

$insert :: {'a} \Rightarrow {'a}\ tree23 \Rightarrow {'a}\ tree23$

$insert\ x\ t = tree_i\ (ins\ x\ t)$

$ins :: {'a} \Rightarrow {'a}\ tree23 \Rightarrow {'a}\ up_i$

Insertion

```
$ins\ x\ \langle l,\ a,\ m,\ b,\ r\rangle =$
case $cmp\ x\ a$ of
  $LT \Rightarrow$ case $ins\ x\ l$ of
              $T_i\ l' \Rightarrow\ T_i\ \langle l',\ a,\ m,\ b,\ r\rangle$
            | $Up_i\ l_1\ c\ l_2 \Rightarrow\ Up_i\ \langle l_1,\ c,\ l_2\rangle\ a\ \langle m,\ b,\ r\rangle$
| $EQ \Rightarrow\ T_i\ \langle l,\ a,\ m,\ b,\ r\rangle$
| $GT \Rightarrow$
    case $cmp\ x\ b$ of
      $LT \Rightarrow$
         case $ins\ x\ m$ of
           $T_i\ m' \Rightarrow T_i\ \langle l,\ a,\ m',\ b,\ r\rangle$
         | $Up_i\ m_1\ c\ m_2 \Rightarrow\ Up_i\ \langle l,\ a,\ m_1\rangle\ c\ \langle m_2,\ b,\ r\rangle$
    | $EQ \Rightarrow\ T_i\ \langle l,\ a,\ m,\ b,\ r\rangle$
    | $GT \Rightarrow$
```

Insertion preserves bal

Lemma

$bal\ t \Longrightarrow bal\ (tree_i\ (ins\ a\ t)) \wedge h(ins\ a\ t) = h(t)$

where $h :: {}'a\ up_i \Rightarrow nat$

$h(T_i\ t) = h(t)$

$h(Up_i\ l\ a\ r) = h(l)$

Proof by induction on t. Step automatic.

Corollary

$bal\ t \Longrightarrow bal\ (insert\ a\ t)$

Deletion

The idea:

$$\begin{array}{lcl} Node3 & \leadsto & Node2 \\ Node2 & \leadsto & \text{underflow, height -1} \end{array}$$

Underflow: merge with siblings on the way up

Deletion

Two possible return values:

- height unchanged: $T_d\ t$
- height decreased by 1: $Up_d\ t$

datatype $'a\ up_d = T_d\ ('a\ tree23) \mid Up_d\ ('a\ tree23)$

$tree_d\ (T_d\ t) = t$
$tree_d\ (Up_d\ t) = t$

Deletion

$$delete :: {'a} \Rightarrow {'a}\ tree23 \Rightarrow {'a}\ tree23$$
$$delete\ x\ t = tree_d\ (del\ x\ t)$$

$$del :: {'a} \Rightarrow {'a}\ tree23 \Rightarrow {'a}\ up_d$$

$$
\begin{array}{l}
del\ x\ \langle l,\ a,\ r\rangle\ = \\
(\textsf{case}\ cmp\ x\ a\ \textsf{of} \\
\quad LT \Rightarrow node21\ (del\ x\ l)\ a\ r \\
\ \mid EQ \Rightarrow \textsf{let}\ (a',\ t) = del\_min\ r\ \textsf{in}\ node22\ l\ a'\ t \\
\ \mid GT \Rightarrow node22\ l\ a\ (del\ x\ r))
\end{array}
$$

$$
\begin{array}{l}
node21\ (T_d\ t_1)\ a\ t_2 = T_d\ \langle t_1,\ a,\ t_2\rangle \\
node21\ (Up_d\ t_1)\ a\ \langle t_2,\ b,\ t_3\rangle = Up_d\ \langle t_1,\ a,\ t_2,\ b,\ t_3\rangle \\
node21\ (Up_d\ t_1)\ a\ \langle t_2,\ b,\ t_3,\ c,\ t_4\rangle = \\
T_d\ \langle\langle t_1,\ a,\ t_2\rangle,\ b,\ \langle t_3,\ c,\ t_4\rangle\rangle
\end{array}
$$

Deletion preserves bal

After 13 simple lemmas:

Lemma

$bal\ t \Longrightarrow bal\ (tree_d\ (del\ x\ t))$

Corollary

$bal\ t \Longrightarrow bal\ (delete\ x\ t)$

Beyond 2-3 trees

datatype $'a\ tree234 =$
$\quad Leaf \mid Node2\ \ldots \mid Node3\ \ldots \mid Node4\ \ldots$

Like 2-3 tress, but with many more cases

The general case:

B-trees and (a, b)-trees

④ Correctness

⑤ 2-3 Trees

⑥ Red-Black Trees

Relationship to 2-3-4 trees

Idea: encode 2-3-4 trees as binary trees;
use color to express grouping

$$\begin{array}{rcl}
\langle\rangle & \approx & \langle\rangle \\
\langle t_1, a, t_2\rangle & \approx & \langle t_1, a, t_2\rangle \\
\langle t_1, a, t_2, b, t_3\rangle & \approx & \langle\langle t_1, a, t_2\rangle, b, t_3\rangle \ \langle t_1, a, \langle t_2, b, t_3\rangle\rangle \\
\langle t_1, a, t_2, b, t_3, c, t_4\rangle & \approx & \langle\langle t_1, a, t_2\rangle, b, \langle t_3, c, t_4\rangle\rangle
\end{array}$$

Red means “I am part of a bigger node”

Red-black trees

datatype $color = Red \mid Black$

datatype

$'a\ rbt = Leaf \mid Node\ color\ ('a\ tree)\ 'a\ ('a\ tree)$

Abbreviations:

$$\begin{array}{rcl} \langle\rangle & \equiv & Leaf \\ \langle c,\ l,\ a,\ r\rangle & \equiv & Node\ c\ l\ a\ r \\ R\ l\ a\ r & \equiv & Node\ Red\ l\ a\ r \\ B\ l\ a\ r & \equiv & Node\ Black\ l\ a\ r \end{array}$$

Color

$$color :: {}'a\ rbt \Rightarrow color$$

$$color\ \langle\rangle = Black$$

$$color\ \langle c,\ \_,\ \_,\ \_\rangle = c$$

$$paint :: color \Rightarrow {}'a\ rbt \Rightarrow {}'a\ rbt$$

$$paint\ c\ \langle\rangle = \langle\rangle$$

$$paint\ c\ \langle \_,\ l,\ a,\ r\rangle = \langle c,\ l,\ a,\ r\rangle$$

Invariants

$rbt :: 'a\ rbt \Rightarrow bool$

$rbt\ t = (invc\ t \wedge invh\ t \wedge color\ t = Black)$

$invc :: 'a\ rbt \Rightarrow bool$

$invc\ \langle\rangle = True$

$invc\ \langle c,\ l,\ \_,\ r\rangle =$

$(invc\ l \wedge$

$\ invc\ r \wedge$

$\ (c = Red \longrightarrow color\ l = Black \wedge color\ r = Black))$

Invariants

$invh :: {}'a\ rbt \Rightarrow bool$

$invh\ \langle\rangle = True$

$invh\ \langle\_,\ l,\ \_,\ r\rangle = (invh\ l \wedge invh\ r \wedge bh(l) = bh(r))$

$bheight :: {}'a\ rbt \Rightarrow nat$

$bh(\langle\rangle) = 0$

$bh(\langle c,\ l,\ \_,\ \_\rangle) =$

$(\textsf{if}\ c = Black\ \textsf{then}\ bh(l) + 1\ \textsf{else}\ bh(l))$

Logarithmic height

Lemma

$rbt\ t \Longrightarrow h(t) \leq 2 * \log_2 |t|_1$

Insertion

$$insert :: {}'a \Rightarrow {}'a\ rbt \Rightarrow {}'a\ rbt$$

$$insert\ x\ t = paint\ Black\ (ins\ x\ t)$$

$$ins :: {}'a \Rightarrow {}'a\ rbt \Rightarrow {}'a\ rbt$$

$$\begin{array}{l}
ins\ x\ \langle\rangle\ =\ R\ \langle\rangle\ x\ \langle\rangle \\
ins\ x\ (B\ l\ a\ r) = (\textsf{case}\ cmp\ x\ a\ \textsf{of} \\
\qquad\qquad\qquad\qquad\quad LT \Rightarrow baliL\ (ins\ x\ l)\ a\ r \\
\qquad\qquad\qquad\qquad\ \mid EQ \Rightarrow B\ l\ a\ r \\
\qquad\qquad\qquad\qquad\ \mid GT \Rightarrow baliR\ l\ a\ (ins\ x\ r))
\end{array}$$

$$\begin{array}{l}
ins\ x\ (R\ l\ a\ r) = (\textsf{case}\ cmp\ x\ a\ \textsf{of} \\
\qquad\qquad\qquad\qquad\quad LT \Rightarrow R\ (ins\ x\ l)\ a\ r \\
\qquad\qquad\qquad\qquad\ \mid EQ \Rightarrow R\ l\ a\ r \\
\qquad\qquad\qquad\qquad\ \mid GT \Rightarrow R\ l\ a\ (ins\ x\ r))
\end{array}$$

Adjusting colors

$baliL$, $baliR :: {}'a\ rbt \Rightarrow {}'a \Rightarrow {}'a\ rbt \Rightarrow {}'a\ rbt$

- Combine arguments $l\ a\ r$ into tree, ideally $\langle l,\ a,\ r\rangle$
- Treat invariant violation Red-Red in l/r
  $baliL\ (R\ (R\ t_1\ a_1\ t_2)\ a_2\ t_3)\ a_3\ t_4$
  $\quad = R\ (B\ t_1\ a_1\ t_2)\ a_2\ (B\ t_3\ a_3\ t_4)$
  $baliL\ (R\ t_1\ a_1\ (R\ t_2\ a_2\ t_3))\ a_3\ t_4$
  $\quad = R\ (B\ t_1\ a_1\ t_2)\ a_2\ (B\ t_3\ a_3\ t_4)$
- Principle: replace Red-Red by Red-Black
- Final equation:
  $baliL\ l\ a\ r = B\ l\ a\ r$
- Symmetric: $baliR$

Preservation of invariant

After 14 simple lemmas:

Theorem

$rbt\ t \Longrightarrow rbt\ (insert\ x\ t)$

Proof in CLRS

The **while** loop in lines 1–15 maintains the following three-part invariant at the start of each iteration of the loop:

a. Node z is red.

b. If $z.p$ is the root, then $z.p$ is black.

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4. If the tree violates property 2, it is because z is the root and is red. If the tree violates property 4, it is because both z and $z.p$ are red.

Part (c), which deals with violations of red-black properties, is more central to showing that RB-INSERT-FIXUP restores the red-black properties than parts (a) and (b), which we use along the way to understand situations in the code. Because we'll be focusing on node z and nodes near it in the tree, it helps to know from part (a) that z is red. We shall use part (b) to show that the node $z.p.p$ exists when we reference it in lines 2, 3, 7, 8, 13, and 14.

Recall that we need to show that a loop invariant is true prior to the first iteration of the loop, that each iteration maintains the loop invariant, and that the loop invariant gives us a useful property at loop termination.

We start with the initialization and termination arguments. Then, as we examine how the body of the loop works in more detail, we shall argue that the loop maintains the invariant upon each iteration. Along the way, we shall also demonstrate that each iteration of the loop has two possible outcomes: either the pointer z moves up the tree, or we perform some rotations and then the loop terminates.

Initialization: Prior to the first iteration of the loop, we started with a red-black tree with no violations, and we added a red node z. We show that each part of the invariant holds at the time RB-INSERT-FIXUP is called:

a. When RB-INSERT-FIXUP is called, z is the red node that was added.

b. If $z.p$ is the root, then $z.p$ started out black and did not change prior to the call of RB-INSERT-FIXUP.

c. We have already seen that properties 1, 3, and 5 hold when RB-INSERT-FIXUP is called.

If the tree violates property 2, then the red root must be the newly added node z, which is the only internal node in the tree. Because the parent and both children of z are the sentinel, which is black, the tree does not also violate property 4. Thus, this violation of property 2 is the only violation of red-black properties in the entire tree.

If the tree violates property 4, then, because the children of node z are black sentinels and the tree had no other violations prior to z being added, the violation must be because both z and $z.p$ are red. Moreover, the tree violates no other red-black properties.

Termination: When the loop terminates, it does so because $z.p$ is black. (If z is the root, then $z.p$ is the sentinel $T.nil$, which is black.) Thus, the tree does not violate property 4 at loop termination. By the loop invariant, the only property that might fail to hold is property 2. Line 16 restores this property, too, so that when RB-INSERT-FIXUP terminates, all the red-black properties hold.

Maintenance: We actually need to consider six cases in the **while** loop, but three of them are symmetric to the other three, depending on whether line 2 determines z's parent $z.p$ to be a left child or a right child of z's grandparent $z.p.p$. We have given the code only for the situation in which $z.p$ is a left child. The node $z.p.p$ exists, since by part (b) of the loop invariant, if $z.p$ is the root, then $z.p$ is black. Since we enter a loop iteration only if $z.p$ is red, we know that $z.p$ cannot be the root. Hence, $z.p.p$ exists.

We distinguish case 1 from cases 2 and 3 by the color of z's parent's sibling, or "uncle." Line 3 makes y point to z's uncle $z.p.p.right$, and line 4 tests y's color. If y is red, then we execute case 1. Otherwise, control passes to cases 2 and 3. In all three cases, z's grandparent $z.p.p$ is black, since its parent $z.p$ is red, and property 4 is violated only between z and $z.p$.

Case 1: z's uncle y is red

Figure 13.5 shows the situation for case 1 (lines 5–8), which occurs when both $z.p$ and y are red. Because $z.p.p$ is black, we can color both $z.p$ and y black, thereby fixing the problem of z and $z.p$ both being red, and we can color $z.p.p$ red, thereby maintaining property 5. We then repeat the **while** loop with $z.p.p$ as the new node z. The pointer z moves up two levels in the tree.

Now, we show that case 1 maintains the loop invariant at the start of the next iteration. We use z to denote node z in the current iteration, and $z' = z.p.p$ to denote the node that will be called node z at the test in line 1 upon the next iteration.

a. Because this iteration colors $z.p.p$ red, node z' is red at the start of the next iteration.

b. The node $z'.p$ is $z.p.p.p$ in this iteration, and the color of this node does not change. If this node is the root, it was black prior to this iteration, and it remains black at the start of the next iteration.

c. We have already argued that case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.

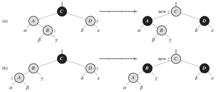


Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since z and its parent $z.p$ are both red. We take the same action whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward simple paths from a node to a leaf have the same number of blacks. The **while** loop continues with node z's grandparent $z.p.p$ as the new z. Any violation of property 4 can now occur only between the new z, which is red, and its parent, if it is red as well.

If node z' is the root at the start of the next iteration, then case 1 corrected the lone violation of property 4 in this iteration. Since z' is red and it is the root, property 2 becomes the only one that is violated, and this violation is due to z'.

If node z' is not the root at the start of the next iteration, then case 1 has not created a violation of property 2. Case 1 corrected the lone violation of property 4 that existed at the start of this iteration. It then made z' red and left $z'.p$ alone. If $z'.p$ was black, there is no violation of property 4. If $z'.p$ was red, coloring z' red created one violation of property 4 between z' and $z'.p$.

Case 2: z's uncle y is black and z is a right child
Case 3: z's uncle y is black and z is a left child

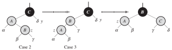


Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $z.p$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise we would be in case 1), and each has the same black-height. We transform case 2 into case 3 by a left rotation, which preserves property 5: all downward simple paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The **while** loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

In cases 2 and 3, the color of z's uncle y is black. We distinguish the two cases according to whether z is a right or left child of $z.p$. Lines 10–11 constitute case 2, which is shown in Figure 13.6 together with case 3. In case 2, node z is a right child of its parent. We immediately use a left rotation to transform the situation into case 3 (lines 12–14), in which node z is a left child. Because both z and $z.p$ are red, the rotation affects neither the black-height of nodes nor property 5. Whether we enter case 3 directly or through case 2, z's uncle y is black, since otherwise we would have executed case 1. Additionally, the node $z.p.p$ exists, since we have argued that this node existed at the time that lines 2 and 3 were executed, and after moving z up one level in line 10 and then down one level in line 11, the identity of $z.p.p$ remains unchanged. In case 3, we execute some color changes and a right rotation, which preserve property 5, and then, since we no longer have two red nodes in a row, we are done. The **while** loop does not iterate another time, since $z.p$ is now black.

We now show that cases 2 and 3 maintain the loop invariant. (As we have just argued, $z.p$ will be black upon the next test in line 1, and the loop body will not execute again.)

a. Case 2 makes z point to $z.p$, which is red. No further change to z or its color occurs in cases 2 and 3.

b. Case 3 makes $z.p$ black, so that if $z.p$ is the root at the start of the next iteration, it is black.

c. As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.

Since node z is not the root in cases 2 and 3, we know that there is no violation of property 2. Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3.

Cases 2 and 3 correct the lone violation of property 4, and they do not introduce another violation.

Deletion

```
$delete\ x\ t = paint\ Black\ (del\ x\ t)$

$del\ \_\ \langle\rangle = \langle\rangle$
$del\ x\ \langle\_,\ l,\ a,\ r\rangle =$
(case $cmp\ x\ a$ of
   $LT \Rightarrow$
     if $l \neq \langle\rangle \wedge color\ l = Black$
     then $baldL\ (del\ x\ l)\ a\ r$ else $R\ (del\ x\ l)\ a\ r$
 | $EQ \Rightarrow combine\ l\ r$
 | $GT \Rightarrow$
     if $r \neq \langle\rangle \wedge color\ r = Black$
     then $baldR\ l\ a\ (del\ x\ r)$ else $R\ l\ a\ (del\ x\ r)$)
```

Deletion

Tricky functions: $baldL$, $baldR$, $combine$

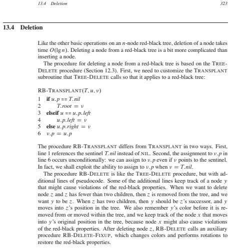
12 short but tricky to find invariant lemmas with short proofs. The worst:

$$\begin{array}{l}
[\![invh\ t;\ invc\ t]\!] \\
\Longrightarrow invh\ (del\ x\ t) \wedge \\
\qquad (color\ t = Red \wedge \\
\qquad\ \ bh(del\ x\ t) = bh(t) \wedge invc\ (del\ x\ t) \vee \\
\qquad\ \ color\ t = Black \wedge \\
\qquad\ \ bh(del\ x\ t) = bh(t) - 1 \wedge invc2\ (del\ x\ t))
\end{array}$$

Theorem

$rbt\ t \Longrightarrow rbt\ (delete\ k\ t)$

Code and proof in CLRS



13.4 Deletion

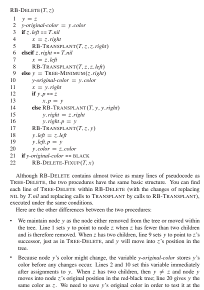
Like the other basic operations on an n-node red-black tree, deletion of a node takes time $O(\lg n)$. Deleting a node from a red-black tree is a bit more complicated than inserting a node.

The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure (Section 12.3). First, we need to customize the TRANSPLANT subroutine that TREE-DELETE calls so that it applies to a red-black tree:

```
RB-TRANSPLANT($T, u, v$)
1  if $u.p == T.nil$
2      $T.root = v$
3  elseif $u == u.p.left$
4      $u.p.left = v$
5  else $u.p.right = v$
6  $v.p = u.p$
```

The procedure RB-TRANSPLANT differs from TRANSPLANT in two ways. First, line 1 references the sentinel $T.nil$ instead of NIL. Second, the assignment to $v.p$ in line 6 occurs unconditionally: we can assign to $v.p$ even if v points to the sentinel. In fact, we shall exploit the ability to assign to $v.p$ when $v = T.nil$.

The procedure RB-DELETE is like the TREE-DELETE procedure, but with additional lines of pseudocode. Some of the additional lines keep track of a node y that might cause violations of the red-black properties. When we want to delete node z and z has fewer than two children, then z is removed from the tree, and we want y to be z. When z has two children, then y should be z's successor, and y moves into z's position in the tree. We also remember y's color before it is removed from or moved within the tree, and we keep track of the node x that moves into y's original position in the tree, because node x might also cause violations of the red-black properties. After deleting node z, RB-DELETE calls an auxiliary procedure RB-DELETE-FIXUP, which changes colors and performs rotations to restore the red-black properties.

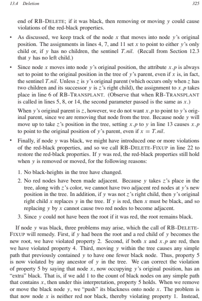


```
RB-DELETE($T, z$)
 1  $y = z$
 2  $y$-original-color $= y.color$
 3  if $z.left == T.nil$
 4      $x = z.right$
 5      RB-TRANSPLANT($T, z, z.right$)
 6  elseif $z.right == T.nil$
 7      $x = z.left$
 8      RB-TRANSPLANT($T, z, z.left$)
 9  else $y$ = TREE-MINIMUM($z.right$)
10      $y$-original-color $= y.color$
11      $x = y.right$
12      if $y.p == z$
13          $x.p = y$
14      else RB-TRANSPLANT($T, y, y.right$)
15          $y.right = z.right$
16          $y.right.p = y$
17      RB-TRANSPLANT($T, z, y$)
18      $y.left = z.left$
19      $y.left.p = y$
20      $y.color = z.color$
21  if $y$-original-color == BLACK
22      RB-DELETE-FIXUP($T, x$)
```

Although RB-DELETE contains almost twice as many lines of pseudocode as TREE-DELETE, the two procedures have the same basic structure. You can find each line of TREE-DELETE within RB-DELETE (with the changes of replacing NIL by $T.nil$ and replacing calls to TRANSPLANT by calls to RB-TRANSPLANT), executed under the same conditions.

Here are the other differences between the two procedures:

- We maintain node y as the node either removed from the tree or moved within the tree. Line 1 sets y to point to node z when z has fewer than two children and is therefore removed. When z has two children, line 9 sets y to point to z's successor, just as in TREE-DELETE, and y will move into z's position in the tree.
- Because node y's color might change, the variable *y-original-color* stores y's color before any changes occur. Lines 2 and 10 set this variable immediately after assignments to y. When z has two children, then $y \neq z$ and node y moves into node z's original position in the red-black tree; line 20 gives y the same color as z. We need to save y's original color in order to test it at the



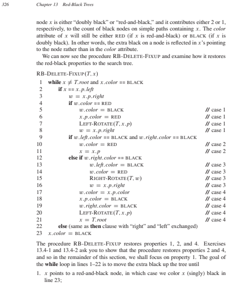
end of RB-DELETE; if it was black, then removing or moving y could cause violations of the red-black properties.

- As discussed, we keep track of the node x that moves into node y's original position. The assignments in lines 4, 7, and 11 set x to point to either y's only child or, if y has no children, the sentinel $T.nil$. (Recall from Section 12.3 that y has no left child.)
- Since node x moves into node y's original position, the attribute $x.p$ is always set to point to the original position in the tree of y's parent, even if x is, in fact, the sentinel $T.nil$. Unless z is y's original parent (which occurs only when z has two children and its successor y is z's right child), the assignment to $x.p$ takes place in line 6 of RB-TRANSPLANT. (Observe that when RB-TRANSPLANT is called in lines 5, 8, or 14, the second parameter passed is the same as x.)

  When y's original parent is z, however, we do not want $x.p$ to point to y's original parent, since we are removing that node from the tree. Because node y will move up to take z's position in the tree, setting $x.p$ to y in line 13 causes $x.p$ to point to the original position of y's parent, even if $x = T.nil$.
- Finally, if node y was black, we might have introduced one or more violations of the red-black properties, and so we call RB-DELETE-FIXUP in line 22 to restore the red-black properties. If y was red, the red-black properties still hold when y is removed or moved, for the following reasons:

  1. No black-heights in the tree have changed.
  2. No red nodes have been made adjacent. Because y takes z's place in the tree, along with z's color, we cannot have two adjacent red nodes at y's new position in the tree. In addition, if y was not z's right child, then y's original right child x replaces y in the tree. If y is red, then x must be black, and so replacing y by x cannot cause two red nodes to become adjacent.
  3. Since y could not have been the root if it was red, the root remains black.

If node y was black, three problems may arise, which the call of RB-DELETE-FIXUP will remedy. First, if y had been the root and a red child of y becomes the new root, we have violated property 2. Second, if both x and $x.p$ are red, then we have violated property 4. Third, moving y within the tree causes any simple path that previously contained y to have one fewer black node. Thus, property 5 is now violated by any ancestor of y in the tree. We can correct the violation of property 5 by saying that node x, now occupying y's original position, has an "extra" black. That is, if we add 1 to the count of black nodes on any simple path that contains x, then under this interpretation, property 5 holds. When we remove or move the black node y, we "push" its blackness onto node x. The problem is that now node x is neither red nor black, thereby violating property 1. Instead,



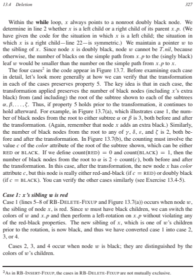
node x is either "doubly black" or "red-and-black," and it contributes either 2 or 1, respectively, to the count of black nodes on simple paths containing x. The *color* attribute of x will still be either RED (if x is red-and-black) or BLACK (if x is doubly black). In other words, the extra black on a node is reflected in x's pointing to the node rather than in the *color* attribute.

We can now see the procedure RB-DELETE-FIXUP and examine how it restores the red-black properties to the search tree.

```
RB-DELETE-FIXUP($T, x$)
 1  while $x \neq T.root$ and $x.color$ == BLACK
 2      if $x == x.p.left$
 3          $w = x.p.right$
 4          if $w.color$ == RED
 5              $w.color$ = BLACK                            // case 1
 6              $x.p.color$ = RED                            // case 1
 7              LEFT-ROTATE($T, x.p$)                        // case 1
 8              $w = x.p.right$                              // case 1
 9          if $w.left.color$ == BLACK and $w.right.color$ == BLACK
10              $w.color$ = RED                              // case 2
11              $x = x.p$                                    // case 2
12          else if $w.right.color$ == BLACK
13                  $w.left.color$ = BLACK                   // case 3
14                  $w.color$ = RED                          // case 3
15                  RIGHT-ROTATE($T, w$)                     // case 3
16                  $w = x.p.right$                          // case 3
17              $w.color = x.p.color$                        // case 4
18              $x.p.color$ = BLACK                          // case 4
19              $w.right.color$ = BLACK                      // case 4
20              LEFT-ROTATE($T, x.p$)                        // case 4
21              $x = T.root$                                 // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  $x.color$ = BLACK
```

The procedure RB-DELETE-FIXUP restores properties 1, 2, and 4. Exercises 13.4-1 and 13.4-2 ask you to show that the procedure restores properties 2 and 4, and so in the remainder of this section, we shall focus on property 1. The goal of the **while** loop in lines 1–22 is to move the extra black up the tree until

1. x points to a red-and-black node, in which case we color x (singly) black in line 23;



Within the **while** loop, x always points to a nonroot doubly black node. We determine in line 2 whether x is a left child or a right child of its parent $x.p$. (We have given the code for the situation in which x is a left child; the situation in which x is a right child—line 22—is symmetric.) We maintain a pointer w to the sibling of x. Since node x is doubly black, node w cannot be $T.nil$, because otherwise, the number of blacks on the simple path from $x.p$ to the (singly black) leaf w would be smaller than the number on the simple path from $x.p$ to x.

The four cases[2] in the code appear in Figure 13.7. Before examining each case in detail, let's look more generally at how we can verify that the transformation in each of the cases preserves property 5. The key idea is that in each case, the transformation applied preserves the number of black nodes (including x's extra black) from (and including) the root of the subtree shown to each of the subtrees $\alpha, \beta, \ldots, \zeta$. Thus, if property 5 holds prior to the transformation, it continues to hold afterward. For example, in Figure 13.7(a), which illustrates case 1, the number of black nodes from the root to either subtree α or β is 3, both before and after the transformation. (Again, remember that node x adds an extra black.) Similarly, the number of black nodes from the root to any of γ, δ, ε, and ζ is 2, both before and after the transformation. In Figure 13.7(b), the counting must involve the value c of the *color* attribute of the root of the subtree shown, which can be either RED or BLACK. If we define $\text{count}(\text{RED}) = 0$ and $\text{count}(\text{BLACK}) = 1$, then the number of black nodes from the root to α is $2 + \text{count}(c)$, both before and after the transformation. In this case, after the transformation, the new node x has *color* attribute c, but this node is really either red-and-black (if $c = \text{RED}$) or doubly black (if $c = \text{BLACK}$). You can verify the other cases similarly (see Exercise 13.4-5).

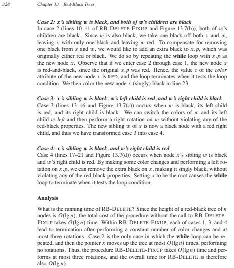
Case 1: x's sibling w is red

Case 1 (lines 5–8 of RB-DELETE-FIXUP and Figure 13.7(a)) occurs when node w, the sibling of node x, is red. Since w must have black children, we can switch the colors of w and $x.p$ and then perform a left-rotation on $x.p$ without violating any of the red-black properties. The new sibling of x, which is one of w's children prior to the rotation, is now black, and thus we have converted case 1 into case 2, 3, or 4.

Cases 2, 3, and 4 occur when node w is black; they are distinguished by the colors of w's children.

---

[2] As in RB-INSERT-FIXUP, the cases in RB-DELETE-FIXUP are not mutually exclusive.



Case 2: x's sibling w is black, and both of w's children are black

In case 2 (lines 10–11 of RB-DELETE-FIXUP and Figure 13.7(b)), both of w's children are black. Since w is also black, we take one black off both x and w, leaving x with only one black and leaving w red. To compensate for removing one black from x and w, we would like to add an extra black to $x.p$, which was originally either red or black. We do so by repeating the **while** loop with $x.p$ as the new node x. Observe that if we enter case 2 through case 1, the new node x is red-and-black, since the original $x.p$ was red. Hence, the value c of the *color* attribute of the new node x is RED, and the loop terminates when it tests the loop condition. We then color the new node x (singly) black in line 23.

Case 3: x's sibling w is black, w's left child is red, and w's right child is black

Case 3 (lines 13–16 and Figure 13.7(c)) occurs when w is black, its left child is red, and its right child is black. We can switch the colors of w and its left child $w.left$ and then perform a right rotation on w without violating any of the red-black properties. The new sibling w of x is now a black node with a red right child, and thus we have transformed case 3 into case 4.

Case 4: x's sibling w is black, and w's right child is red

Case 4 (lines 17–21 and Figure 13.7(d)) occurs when node x's sibling w is black and w's right child is red. By making some color changes and performing a left rotation on $x.p$, we can remove the extra black on x, making it singly black, without violating any of the red-black properties. Setting x to be the root causes the **while** loop to terminate when it tests the loop condition.

Analysis

What is the running time of RB-DELETE? Since the height of a red-black tree of n nodes is $O(\lg n)$, the total cost of the procedure without the call to RB-DELETE-FIXUP takes $O(\lg n)$ time. Within RB-DELETE-FIXUP, each of cases 1, 3, and 4 lead to termination after performing a constant number of color changes and at most three rotations. Case 2 is the only case in which the **while** loop can be repeated, and then the pointer x moves up the tree at most $O(\lg n)$ times, performing no rotations. Thus, the procedure RB-DELETE-FIXUP takes $O(\lg n)$ time and performs at most three rotations, and the overall time for RB-DELETE is therefore also $O(\lg n)$.

Source of code

Insertion:

Okasaki's *Purely Functional Data Structures*

Deletion:

Stefan Kahrs. *Red Black Trees with Types*.
J. Functional Programming. 1996.

Lecture 3

Priority Queues

❼ Priority Queues

❽ Leftist Heap

❾ Priority Queue via Braun Tree

❿ Binomial Heap

7 Priority Queues

8 Leftist Heap

9 Priority Queue via Braun Tree

10 Binomial Heap

Priority queue informally

Collection of elements with priorities

We focus on the priorities:
element = priority

The same element can be contained multiple times
in a priority queue

$\Longrightarrow$

The abstract view of a priority queue is a multiset

Interface of implementation

The type of elements (= priorities) $'a$ is a linear order

An implementation of a priority queue of elements of type $'a$ must provide

- An implementation type $'q$
- $empty :: 'q$
- $is\_empty :: 'q \Rightarrow bool$
- $insert :: 'a \Rightarrow 'q \Rightarrow 'q$
- $get\_min :: 'q \Rightarrow 'a$
- $del\_min :: 'q \Rightarrow 'q$

More operations

- $merge :: 'q \Rightarrow 'q \Rightarrow 'q$
  Often provided
- decrease key/priority
  Not easy in functional setting

Correctness of implementation

A priority queue represents a multiset of priorities.
Correctness proof requires:

Abstraction function: $mset :: 'q \Rightarrow 'a\ multiset$
Invariant: $invar :: 'q \Rightarrow bool$

Correctness of implementation

Must prove $invar\ q \Longrightarrow$

$mset\ (insert\ x\ q) = mset\ q + \{\#x\#\}$

$mset\ q \neq \{\#\} \Longrightarrow$
$mset\ (del\_min\ q) = mset\ q - \{\#Min\_mset\ (mset\ q)\#\}$

$invar\ (insert\ x\ q)$
$invar\ (del\_min\ q)$

Terminology

A tree is a *heap* if for every subtree
the root is $\leq$ all elements in that subtree.

The term "heap" is frequently used synonymously with "priority queue".

Priority queue via heap

- $get\_min\ \langle\_,\ a,\ \_\rangle = a$
- Assume we have $merge$
- $insert\ a\ t = merge\ \langle\langle\rangle,\ a,\ \langle\rangle\rangle\ t$
- $del\_min\ \langle l,\ a,\ r\rangle = merge\ l\ r$

Leftist tree informally

The *rank* of a tree is the depth of the rightmost leaf.

In a *leftist tree*, the rank of every left child is $\geq$ the rank of its right sibling.

Merge descends along the right spine.
Thus rank bounds number of steps.

If rank of right child gets too large: swap with left child.

Implementation type

datatype

$'a\ lheap = Leaf \mid Node\ nat\ ('a\ tree)\ 'a\ ('a\ tree)$

Abbreviations $\langle\rangle$ and $\langle h,\ l,\ a,\ r\rangle$ as usual

Abstraction function:

$mset\_tree :: 'a\ lheap \Rightarrow 'a\ multiset$

$mset\_tree\ \langle\rangle = \{\#\}$

$mset\_tree\ \langle\_,\ l,\ a,\ r\rangle =$

$\{\#a\#\} + mset\_tree\ l + mset\_tree\ r$

Leftist tree

$rank :: 'a\ lheap \Rightarrow nat$

$rank\ \langle\rangle = 0$

$rank\ \langle\_,\ \_,\ \_,\ r\rangle = rank\ r + 1$

Node $\langle n,\ l,\ a,\ r\rangle$: $n =$ rank of node

$ltree :: 'a\ lheap \Rightarrow bool$

$ltree\ \langle\rangle = True$

$ltree\ \langle n,\ l,\ \_,\ r\rangle =$
$(n = rank\ r + 1 \wedge rank\ r \leq rank\ l \wedge ltree\ l \wedge ltree\ r)$

Leftist heap invariant

$$invar\ h = (heap\ h \wedge ltree\ h)$$

$merge$

Principle: descend on the right

$merge\ \langle\rangle\ t_2 = t_2$
$merge\ t_1\ \langle\rangle = t_1$
$merge\ \langle n_1, l_1, a_1, r_1\rangle\ \langle n_2, l_2, a_2, r_2\rangle =$
(if $a_1 \le a_2$ then $node\ l_1\ a_1\ (merge\ r_1\ \langle n_2, l_2, a_2, r_2\rangle)$
 else $node\ l_2\ a_2\ (merge\ r_2\ \langle n_1, l_1, a_1, r_1\rangle))$

$node :: {}'a\ lheap \Rightarrow {}'a \Rightarrow {}'a\ lheap \Rightarrow {}'a\ lheap$

$node\ l\ a\ r =$
(let $rl = rk\ l;\ rr = rk\ r$
 in if $rr \le rl$ then $\langle rr + 1, l, a, r\rangle$ else $\langle rl + 1, r, a, l\rangle)$
where $rk\ \langle n, \_, \_, \_\rangle = n$

Functional correctness proofs

including preservation of $invar$

Straightforward

Logarithmic complexity

Lemma

$ltree\ t \Longrightarrow 2^{rank\ t} \leq |t|_1$

Corollary

$[\![ltree\ l;\ ltree\ r]\!]$
$\Longrightarrow t\_merge\ l\ r \leq \log_2\ |l|_1 + \log_2\ |r|_1 + 1$

Can we avoid the rank info in each node?

7 Priority Queues

8 Leftist Heap

9 Priority Queue via Braun Tree

10 Binomial Heap

Archive of Formal Proofs

`https://www.isa-afp.org/entries/Priority_Queue_Braun.shtml`

What is a Braun tree?

$braun :: {}'a\ tree \Rightarrow bool$

$braun\ \langle\rangle = True$
$braun\ \langle l,\ x,\ r\rangle =$
$(|r| \le |l| \wedge |l| \le |r| + 1 \wedge braun\ l \wedge braun\ r)$

1

Lemma $braun\ t \Longrightarrow 2^{h(t)} \le 2 * |t| + 1$

Idea of invariant maintenance

$$
\begin{aligned}
&braun\ \langle\rangle = True\\
&braun\ \langle l,\ x,\ r\rangle =\\
&(|r| \leq |l| \wedge |l| \leq |r| + 1 \wedge braun\ l \wedge braun\ r)
\end{aligned}
$$

Add element: to right subtree, then swap subtrees

Remove element: from left subtree, then swap subtrees

Priority queue implementation

Implementation type: $'a\ tree$

Invariants: $heap$ and $braun$

No $merge$ — $insert$ and del_min defined explicitly

$insert$

$insert :: {'a} \Rightarrow {'a}\ tree \Rightarrow {'a}\ tree$
$insert\ a\ \langle\rangle = \langle\langle\rangle,\ a,\ \langle\rangle\rangle$
$insert\ a\ \langle l,\ x,\ r\rangle =$
$(\textsf{if}\ a < x\ \textsf{then}\ \langle insert\ x\ r,\ a,\ l\rangle\ \textsf{else}\ \langle insert\ a\ r,\ x,\ l\rangle)$

Correctness and preservation of invariant straightforward.

del_min

$del\_min :: {}'a\ tree \Rightarrow {}'a\ tree$

$del\_min\ \langle\rangle = \langle\rangle$

$del\_min\ \langle\langle\rangle,\ x,\ r\rangle = \langle\rangle$

$del\_min\ \langle l,\ x,\ r\rangle =$

$(\mathsf{let}\ (y,\ l') = del\_left\ l\ \mathsf{in}\ sift\_down\ r\ y\ l')$

1. Delete leftmost element y
2. Sift y from the root down

del_left

$del\_left :: 'a\ tree \Rightarrow 'a \times 'a\ tree$
$del\_left\ \langle\langle\rangle,\ x,\ \langle\rangle\rangle = (x,\ \langle\rangle)$
$del\_left\ \langle l,\ x,\ r\rangle =$
$(\mathsf{let}\ (y,\ l') = del\_left\ l\ \mathsf{in}\ (y,\ \langle r,\ x,\ l'\rangle))$

$sift\_down$

$sift\_down :: {}'a\ tree \Rightarrow {}'a \Rightarrow {}'a\ tree \Rightarrow {}'a\ tree$

$sift\_down\ (t_1 = (Node\ l_1\ x_1\ r_1))\ a\ (t_2 = (Node\ l_2\ x_2\ r_2))$

$=$

if $a \leq x_1 \wedge a \leq x_2$ then $\langle t_1,\ a,\ t_2\rangle$

else if $x_1 \leq x_2$ then $\langle sift\_down\ l_1\ a\ r_1,\ x_1,\ t_2\rangle$

else $\langle t_1,\ x_2,\ sift\_down\ l_2\ a\ r_2\rangle$

Functional correctness proofs for del_min

Many lemmas, mostly straightforward

Logarithmic complexity

Running time of $insert$, del_left and $sift\_down$ (and therefore del_min) bounded by height

Remember: $braun\ t \Longrightarrow 2^{h(t)} \leq 2 * |t| + 1$

$\Longrightarrow$

Above running times logarithmic in size

Source of code

Based on code from

L.C. Paulson. *ML for the Working Programmer*. 1996

based on code from Chris Okasaki.

⑦ Priority Queues

⑧ Leftist Heap

⑨ Priority Queue via Braun Tree

⑩ Binomial Heap

Numerical method

Idea: only use trees t_i of size 2^i

Example

To store (in binary) 11001 elements: $[t_0, 0, 0, t_3, t_4]$

Merge $\approx$ addition with carry
Needs function to combine two trees of size 2^i
into one tree of size 2^{i+1}

Binomial tree

datatype $'a\ tree =$
$\quad Node\ (rank{:}\ nat)\ (root{:}\ 'a)\ ('a\ tree\ list)$

Invariant: Node of rank i has children $[t_{i-1},\dots t_0]$ of ranks $[i{-}1,\dots,0]$:

$invar\_btree\ (Node\ r\ x\ ts) =$
$((\forall\ t{\in}set\ ts.\ invar\_btree\ t) \wedge map\ rank\ ts = rev\ [0..{<}r])$

Lemma

$invar\_btree\ t \Longrightarrow |t| = 2^{rank\ t}$

Combining two trees

How to combine two trees of rank i
into one tree of rank $i{+}1$

```
$link\ t_1\ t_2 =$
(case $(t_1,\ t_2)$ of
  $(Node\ r\ x_1\ c_1,\ Node\ x\ x_2\ c_2) \Rightarrow$
    if $x_1 \le x_2$ then $Node\ (r + 1)\ x_1\ (t_2\ \#\ c_1)$
    else $Node\ (r + 1)\ x_2\ (t_1\ \#\ c_2))$
```

Binomial heap

Use sparse representation for binary numbers:
$[t_0,0,0,t_3,t_4]$ represented as $[\ (0,t_0),\ (3,t_3),(4,t_4)\ ]$

type_synonym $'a\ heap = 'a\ tree\ list$

Remember: $tree$ contains rank

Invariant:

$invar\_bheap\ ts =$
$((\forall\ t{\in}set\ ts.\ invar\_btree\ t) \wedge$
$\ strictly\_ascending\ (map\ rank\ ts))$

Inserting a tree

$ins\_tree\ t\ []\ =\ [t]$
$ins\_tree\ t_1\ (t_2\ \#\ ts)\ =$
$(\mathsf{if}\ rank\ t_1 < rank\ t_2\ \mathsf{then}\ t_1\ \#\ t_2\ \#\ ts$
$\ \mathsf{else}\ ins\_tree\ (link\ t_1\ t_2)\ ts)$

Intuition: Handle a carry

Precondition:
Rank of inserted tree $\leq$ ranks of trees in heap

$merge$

$$
\begin{array}{l}
merge\ ts_1\ []\ =\ ts_1 \\
merge\ []\ ts_2\ =\ ts_2 \\
merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2)\ = \\
(\textsf{if}\ rank\ t_1 < rank\ t_2\ \textsf{then}\ t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2) \\
\ \ \textsf{else if}\ rank\ t_2 < rank\ t_1\ \textsf{then}\ t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2 \\
\qquad\quad \textsf{else}\ ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))
\end{array}
$$

Intuition: Addition of binary numbers

Note: Handling of carry after recursive call

Find/delete minimum element

All trees are min-heaps.

Smallest element may be any root node:

$$ts \neq [] \Longrightarrow \mathit{find\_min}\ ts = \mathit{Min}\ (\mathit{set}\ (\mathit{map}\ \mathit{root}\ ts))$$

Similar: $\mathit{get\_min} :: {'a}\ \mathit{tree}\ \mathit{list} \Rightarrow {'a}\ \mathit{tree} \times {'a}\ \mathit{tree}\ \mathit{list}$
Returns tree with minimal root, and remaining trees

$$\begin{array}{l} \mathit{delete\_min}\ ts = \\ (\textsf{case}\ \mathit{get\_min}\ ts\ \textsf{of} \\ \quad (\mathit{Node}\ r\ x\ ts_1,\ ts_2) \Rightarrow \mathit{merge}\ (\mathit{rev}\ ts_1)\ ts_2) \end{array}$$

Complexity

Recall: $|t| = 2^{rank\ t}$

Similarly for heap: $2^{length\ ts} \leq |ts| + 1$

Complexity of operations: linear in length of heap

i.e., logarithmic in number of elements

Proofs: straightforward?

Complexity of $merge$

```
$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
(if $rank\ t_1 < rank\ t_2$ then $t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
 else if $rank\ t_2 < rank\ t_1$ then $t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
      else $ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$
```

Complexity of ins_tree: $t\_ins\_tree\ t\ ts \leq length\ ts + 1$

A call $merge\ t_1\ t_2$ (where $length\ t_1 = length\ t_2 = n$) can lead to calls of ins_tree on lists of length 1, ..., n.

$\sum \in O(n^2)$

Complexity of $merge$

$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
$($if $rank\ t_1 < rank\ t_2$ then $t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
else if $rank\ t_2 < rank\ t_1$ then $t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
else $ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Relate time and length of input/output:

$t\_ins\_tree\ t\ ts + length\ (ins\_tree\ t\ ts) = 2 + length\ ts$

$length\ (merge\ ts_1\ ts_2) + t\_merge\ ts_1\ ts_2$
$\leq 2 * (length\ ts_1 + length\ ts_2) + 1$

Yields desired linear bound!

Lecture 4

Amortized Complexity

⑪ Skew Heap

⑫ Splay Tree

⑬ Pairing Heap

Archive of Formal Proofs

`https://www.isa-afp.org/entries/Amortized_Analysis.shtml`

11. Skew Heap

12. Splay Tree

13. Pairing Heap

A *skew heap* is a self-adjusting heap (priority queue)

Almost like leftist heap, but no size info needed

$merge$

$merge\ \langle\rangle\ h = h$
$merge\ h\ \langle\rangle = h$

Swap subtrees when descending:

$merge\ (h_1 = \langle l_1,\ a,\ r_1\rangle)\ (h_2 = \langle l_2,\ b,\ r_2\rangle) =$
$(\text{if } a \le b \text{ then } \langle merge\ h_2\ r_1,\ a,\ l_1\rangle$
$\ \text{else } \langle merge\ h_1\ r_2,\ b,\ l_2\rangle)$

Functional correctness proofs

Straightforward

Logarithmic amortized complexity

Theorem

$$t\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$$
$$\leq 3 * \log_2\ (|t_1|_1 + |t_2|_1) + 1$$

Towards the proof

$$rheavy\ \langle l,\ \_,\ r\rangle = (|l| < |r|)$$

$$lpath\ \langle\rangle = []$$
$$lpath\ \langle l,\ a,\ r\rangle = \langle l,\ a,\ r\rangle\ \#\ lpath\ l$$

$$G\ h = length\ (filter\ rheavy\ (lpath\ h))$$

Lemma

$$2^{G\ h} \le |h| + 1$$

Corollary

$$G\ h \le \log_2\ |h|_1$$

Towards the proof

$$rheavy\ \langle l,\ \_,\ r\rangle = (|l| < |r|)$$

$$rpath\ \langle\rangle = []$$
$$rpath\ \langle l,\ a,\ r\rangle = \langle l,\ a,\ r\rangle\ \#\ rpath\ r$$

$$D\ h = length(filter\ (\lambda p.\ \neg\ rheavy\ p)\ (rpath\ h))$$

Lemma

$$2^{D\ h} \leq |h| + 1$$

Corollary

$$D\ h \leq \log_2\ |h|_1$$

Potential

The potential is the number of *rheavy* nodes:

$\Phi\ \langle\rangle = 0$

$\Phi\ \langle l,\ \_,\ r\rangle = \Phi\ l + \Phi\ r + (\mathsf{if}\ |l| < |r|\ \mathsf{then}\ 1\ \mathsf{else}\ 0)$

Lemma

$t\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$
$\leq G\ (merge\ t_1\ t_2) + D\ t_1 + D\ t_2 + 1$

```
by(induction t1 t2 rule: merge.induct)(auto)
```

Main proof

$$
\begin{aligned}
& t\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\
& \leq G\ (merge\ t_1\ t_2) + D\ t_1 + D\ t_2 + 1 \\
& \leq \log_2\ |merge\ t_1\ t_2|_1 + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
& = \log_2\ (|t_1|_1 + |t_2|_1 - 1) + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
& \leq \log_2\ (|t_1|_1 + |t_2|_1) + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
& \leq \log_2\ (|t_1|_1 + |t_2|_1) + 2 * \log_2\ (|t_1|_1 + |t_2|_1) + 1 \\
& \quad \text{because } \log_2\ x + \log_2\ y \leq 2 * \log_2\ (x + y) \text{ if } x,y > 0 \\
& = 3 * \log_2\ (|t_1|_1 + |t_2|_1) + 1
\end{aligned}
$$

Sources

The inventors of skew heaps:

Daniel Sleator and Robert Tarjan.
Self-adjusting Heaps.
SIAM J. Computing, 1986.

The formalization is based on

Anne Kaldewaij and Berry Schoenmakers.
The Derivation of a Tighter Bound for Top-down Skew Heaps. *Information Processing Letters*, 1991.

A *splay tree* is a self-adjusting binary search tree.

Functions $isin$, $insert$ and $delete$
have amortized logarithmic complexity.

⓬ Splay Tree

Algorithm

Amortized Analysis

Splay tree

Implementation type = binary tree

Key operation $splay\ a$:

1. Search for a ending up at x
   where $x = a$ or x is a leaf node.
2. Move x to the root of the tree by rotations.

Derived operations $isin/insert/delete\ a$:

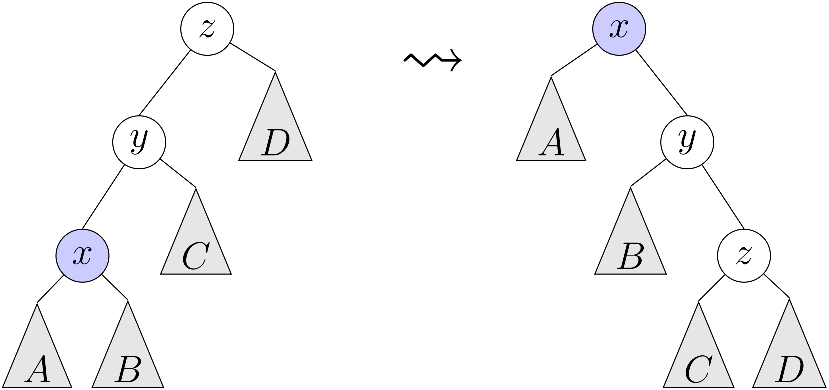
1. $splay\ a$
2. Perform $isin/insert/delete$ action

Key ideas

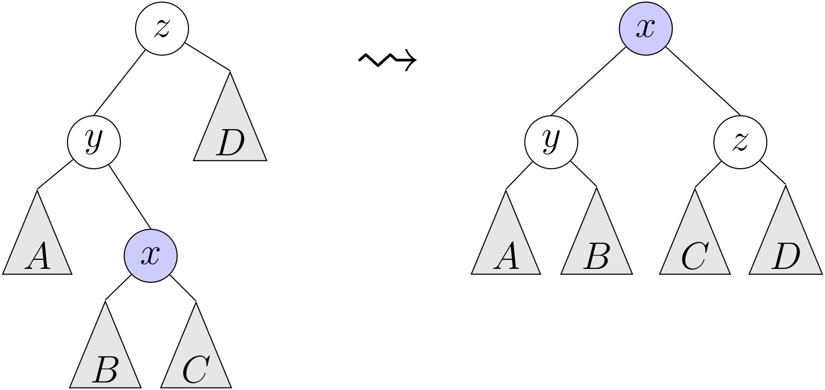
Move to root

Double rotations

Zig-zig



Zig-zag



Zig-zig and zig-zag

Zig-zig $\neq$ two single rotations

Zig-zag $=$ two single rotations

Functional definition

$splay :: {}'a \Rightarrow {}'a\ tree \Rightarrow {}'a\ tree$

Zig-zig and zig-zag

$$\llbracket a < y;\ a < z;\ T \neq \langle\rangle \rrbracket$$
$$\Longrightarrow splay\ a\ \langle\langle T,\ y,\ C\rangle,\ z,\ D\rangle =$$
$$(\textsf{case}\ splay\ a\ T\ \textsf{of}$$
$$\langle A,\ x,\ B\rangle \Rightarrow \langle A,\ x,\ \langle B,\ y,\ \langle C,\ z,\ D\rangle\rangle\rangle)$$

$$\llbracket a < z;\ y < a;\ T \neq \langle\rangle \rrbracket$$
$$\Longrightarrow splay\ a\ \langle\langle A,\ y,\ T\rangle,\ z,\ D\rangle =$$
$$(\textsf{case}\ splay\ a\ T\ \textsf{of}$$
$$\langle B,\ x,\ C\rangle \Rightarrow \langle\langle A,\ y,\ B\rangle,\ x,\ \langle C,\ z,\ D\rangle\rangle)$$

Functional correctness proofs

Automatic

⑫ Splay Tree

Algorithm

Amortized Analysis

Potential

Sum of logarithms of the size of all nodes:

$\Phi\ \langle\rangle = 0$

$\Phi\ \langle l,\ a,\ r\rangle = \Phi\ l + \Phi\ r + \varphi\ \langle l,\ a,\ r\rangle$

where $\varphi\ t = \log_2\ (|t| + 1)$

Amortized complexity of $splay$:

$$a\_splay\ a\ t = t\_splay\ a\ t + \Phi\ (splay\ a\ t) - \Phi\ t$$

Analysis of *splay*

Theorem

$$\llbracket bst\ t;\ \langle l,\ x,\ r\rangle \in subtrees\ t\rrbracket$$
$$\Longrightarrow a\_splay\ x\ t \le 3 * (\varphi\ t - \varphi\ \langle l,\ x,\ r\rangle) + 1$$

Corollary

$$\llbracket bst\ t;\ a \in set\_tree\ t\rrbracket$$
$$\Longrightarrow a\_splay\ a\ t \le 3 * (\varphi\ t - 1) + 1$$

Corollary

$$bst\ t \Longrightarrow a\_splay\ a\ t \le 3 * \varphi\ t + 1$$

Lemma

$$\llbracket t \neq \langle\rangle;\ bst\ t\rrbracket$$
$$\Longrightarrow \exists\, a'\in set\_tree\ t.$$
$$splay\ a'\ t = splay\ a\ t \wedge t\_splay\ a'\ t = t\_splay\ a\ t$$

insert

Definition

```
$insert\ x\ t =$
(if $t = \langle\rangle$ then $\langle\langle\rangle,\ x,\ \langle\rangle\rangle$
 else case $splay\ x\ t$ of
        $\langle l,\ a,\ r\rangle \Rightarrow$
          if $x = a$ then $\langle l,\ a,\ r\rangle$
          else if $x < a$ then $\langle l,\ x,\ \langle\langle\rangle,\ a,\ r\rangle\rangle$
               else $\langle\langle l,\ a,\ \langle\rangle\rangle,\ x,\ r\rangle)$
```

Counting only the cost of *splay*:

Lemma

$$bst\ t \Longrightarrow$$

$$t\_splay\ a\ t + \Phi\ (insert\ a\ t) - \Phi\ t \le 4 * \varphi\ t + 2$$

$delete$

Definition

```
$delete\ x\ t =$
(if $t = \langle\rangle$ then $\langle\rangle$
 else case $splay\ x\ t$ of
        $\langle l,\ a,\ r\rangle \Rightarrow$
          if $x = a$
          then if $l = \langle\rangle$ then $r$
               else case $splay\_max\ l$ of
                      $\langle l',\ m,\ r'\rangle \Rightarrow \langle l',\ m,\ r\rangle$
          else $\langle l,\ a,\ r\rangle$)
```

Lemma

$bst\ t \Longrightarrow$

$$t\_delete\ a\ t + \Phi\ (delete\ a\ t) - \Phi\ t \leq 6 * \varphi\ t + 2$$

$$isin :: {}'a\ tree \Rightarrow {}'a \Rightarrow bool$$

Single threaded $\Longrightarrow$ $isin\ t\ a$ eats up t

Otherwise:

$$\begin{aligned} \textit{let}\quad & bad = \text{build unbalanced splay tree;}\\ & \_ = isin\ bad\ a;\\ & \_ = isin\ bad\ a;\\ & \_ = isin\ bad\ a;\\ & \vdots \end{aligned}$$

Solution 1:

$$isin :: {}'a\ tree \Rightarrow {}'a \Rightarrow bool \times {}'a\ tree$$

Observer function returns new data structure:

Definition

$isin\ t\ a =$
(let $t' = splay\ a\ t$ in (case t' of

$$\langle\rangle \Rightarrow False$$
$$\mid \langle l,\ x,\ r\rangle \Rightarrow a = x,$$
$$t'))$$

Solution 2:
$isin = splay; is\_root$

Client uses $splay$ before calling is_root:

Definition

$is\_root :: {}'a \Rightarrow {}'a\ tree \Rightarrow bool$

$$\begin{aligned} is\_root\ a\ t = (&\text{case } t \text{ of} \\ &\langle\rangle \Rightarrow False \\ &|\ \langle l, x, r\rangle \Rightarrow x = a) \end{aligned}$$

May call $is\_root\ \_\ t$ multiple times (with the same t!) because is_root takes constant time

$\Longrightarrow$ $is\_root\ \_\ t$ does not eat up t

$isin$

Splay trees have an imperative flavour and are a bit awkward to use in a purely functional language

Sources

The inventors of splay trees:

Daniel Sleator and Robert Tarjan.
Self-adjusting Binary Search Trees. *J. ACM*, 1985.

The formalization is based on

Berry Schoenmakers. A Systematic Analysis of Splaying.
Information Processing Letters, 1993.

⑪ Skew Heap

⑫ Splay Tree

⑬ Pairing Heap

Implementation type

datatype $'a\ heap = Empty \mid Hp\ 'a\ ('a\ heap\ list)$

Heap invariant:

$pheap\ Empty = True$
$pheap\ (Hp\ x\ hs) =$
$(\forall\, h{\in}set\ hs.\ (\forall\, y{\in}set\_heap\ h.\ x \le y) \wedge pheap\ h)$

Also: $Empty$ must only occur at the root

$insert$

$insert\ x\ h = merge\ (Hp\ x\ [])\ h$

$merge\ h\ Empty = h$
$merge\ Empty\ h = h$
$merge\ (xh = Hp\ x\ xhs)\ (yh = Hp\ y\ yhs) =$
$(\text{if } x < y \text{ then } Hp\ x\ (yh\ \#\ xhs) \text{ else } Hp\ y\ (xh\ \#\ yhs))$

Like function $link$ for binomial heaps

del_min

$del\_min\ Empty = Empty$
$del\_min\ (Hp\ x\ hs) = merge\_pairs\ hs$

$merge\_pairs\ [] = Empty$
$merge\_pairs\ [h] = h$
$merge\_pairs\ (h_1\ \#\ h_2\ \#\ hs) =$
$merge\ (merge\ h_1\ h_2)\ (merge\_pairs\ hs)$

$merge\_pairs$

$$merge\_pairs\ [] = Empty$$
$$merge\_pairs\ [h] = h$$
$$merge\_pairs\ (h_1 \# h_2 \# hs) =$$
$$merge\ (merge\ h_1\ h_2)\ (merge\_pairs\ hs)$$

$$merge\_pairs\ hs = pass_2\ (pass_1\ hs)$$

$$pass_1\ [] = []$$
$$pass_1\ [h] = [h]$$
$$pass_1\ (h_1 \# h_2 \# hs) = merge\ h_1\ h_2 \# pass_1\ hs$$

$$pass_2\ [] = Empty$$
$$pass_2\ (h \# hs) = merge\ h\ (pass_2\ hs)$$

Functional correctness proofs

Straightforward

⑬ Pairing Heap

Amortized Analysis

Analysis

Analysis easier (more uniform) if a pairing heap is viewed as a binary tree:

$homs :: {'a}\ heap\ list \Rightarrow {'a}\ tree$
$homs\ [] = \langle\rangle$
$homs\ (Hp\ x\ hs_1\ \#\ hs_2) = \langle homs\ hs_1,\ x,\ homs\ hs_2\rangle$

$hom :: {'a}\ heap \Rightarrow {'a}\ tree$
$hom\ Empty = \langle\rangle$
$hom\ (Hp\ x\ hs) = \langle homs\ hs,\ x,\ \langle\rangle\rangle$

Potential function same as for splay trees

Verified:

The functions $insert$, del_min and $merge$ all have $O(\log_2 n)$ amortized complexity.

These bounds are not tight.
Better amortized bounds in the literature:

$insert \in O(1)$, $del\_min \in O(\log_2 n)$, $merge \in O(1)$

The exact complexity is still open.

Sources

The inventors of the pairing heap:

M. Fredman, R. Sedgewick, D. Sleator and R. Tarjan.
The Pairing Heap: A New Form of Self-Adjusting Heap.
Algorithmica, 1986.

The functional version:

Chris Okasaki. *Purely Functional Data Structures.*
Cambridge University Press, 1998.