Functional Data Structures

with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2022-5-31

Part II

Functional Data Structures

Chapter 6

Sorting

$sorted :: ('a::linorder)\ list \Rightarrow bool$

$sorted\ [] = True$
$sorted\ (x \# ys) = ((\forall y \in set\ ys.\ x \leq y) \land sorted\ ys)$

Correctness of sorting

Specification of $sort :: ('a::linorder)\ list \Rightarrow 'a\ list$:

$$sorted\ (sort\ xs)$$

Is that it? How about

$$set\ (sort\ xs) = set\ xs$$

Better: every x occurs as often in $sort\ xs$ as in xs.

More succinctly:

$$mset\ (sort\ xs) = mset\ xs$$

where $mset :: 'a\ list \Rightarrow 'a\ multiset$

What are multisets?

Sets with (possibly) repeated elements

Some operations:

$$
\begin{array}{rcl}
\{\#\} & :: & 'a\ multiset \\
add\_mset & :: & 'a \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset \\
+ & :: & 'a\ multiset \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset \\
mset & :: & 'a\ list \Rightarrow 'a\ multiset \\
set\_mset & :: & 'a\ multiset \Rightarrow 'a\ set
\end{array}
$$

Import $HOL{-}Library.Multiset$

HOL/Data_Structures/Sorting.thy

Insertion Sort Correctness

Principle: Count function calls

For every function $f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau$
define a *timing function* $T_f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow nat$:

Translation of defining equations:

$$\mathcal{E}[\![f\ p_1\ \ldots\ p_n = e]\!] \ = \ (T_f\ p_1\ \ldots\ p_n = \mathcal{T}[\![e]\!] + 1)$$

Translation of expressions:

$$\mathcal{T}[\![g\ e_1\ \ldots\ e_k]\!] \ = \ \mathcal{T}[\![e_1]\!] + \ldots + \mathcal{T}[\![e_k]\!] + T_g\ e_1\ \ldots\ e_k$$

All other operations (variable access, constants, constructors, primitive operations on $bool$ and numbers) cost 1 time unit

Example: @

$$\begin{aligned}
&\mathcal{E}[\![\ [] \ @ \ ys = ys \ ]\!] \\
&= \ (T_{@} \ [] \ ys = \mathcal{T}[\![ys]\!] + 1) \\
&= \ (T_{@} \ [] \ ys = 2)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\ (x \ \# \ xs) \ @ \ ys = x \ \# \ (xs \ @ \ ys) \ ]\!] \\
&= \ (T_{@} \ (x \ \# \ xs) \ ys = \mathcal{T}[\![x \ \# \ (xs \ @ \ ys)]\!] + 1) \\
&= \ (T_{@} \ (x \ \# \ xs) \ ys = T_{@} \ xs \ ys + 5)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}[\![x \ \# \ (xs \ @ \ ys)]\!] \\
&= \mathcal{T}[\![x]\!] + \mathcal{T}[\![xs \ @ \ ys]\!] + T_{\#} \ x \ (xs \ @ \ ys) \\
&= 1 + (\mathcal{T}[\![xs]\!] + \mathcal{T}[\![ys]\!] + T_{@} \ xs \ ys) + 1 \\
&= 1 + (1 + 1 + T_{@} \ xs \ ys) + 1
\end{aligned}$$

if and *case*

So far we model a call-by-value semantics

Conditionals and case expressions are evaluated lazily.

$\mathcal{T}[\![\text{if } b \text{ then } e_1 \text{ else } e_2]\!]$
$= \mathcal{T}[\![b]\!] + (\text{if } b \text{ then } \mathcal{T}[\![e_1]\!] \text{ else } \mathcal{T}[\![e_2]\!])$

$\mathcal{T}[\![\text{case } e \text{ of } p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k]\!]$
$= \mathcal{T}[\![e]\!] + (\text{case } e \text{ of } p_1 \Rightarrow \mathcal{T}[\![e_1]\!] \mid \ldots \mid p_k \Rightarrow \mathcal{T}[\![e_k]\!])$

Also special: let $x = t_1$ in t_2

$O(.)$ is enough

$\Longrightarrow$ Reduce all additive constants to 1

Example

$$T_{@}\ (x \ \#\ xs)\ ys = T_{@}\ xs\ ys + 5 \rightsquigarrow$$
$$T_{@}\ (x \ \#\ xs)\ ys = T_{@}\ xs\ ys + 1$$

This means we count only

- the defined functions via T_f and
- $+1$ for the function call itself.

All other operations (variables etc) cost 0, not 1.

Discussion

- The definition of T_f from f can be automated.
- The correctness of T_f could be proved w.r.t. a semantics that counts computation steps.
- Precise complexity bounds (as opposed to $O(.)$) would require a formal model of (at least) the compiler and the hardware.

HOL/Data_Structures/Sorting.thy

Insertion sort complexity

4 Merge Sort

Top-Down

Bottom-Up

$merge :: {}'a\ list \Rightarrow {}'a\ list \Rightarrow {}'a\ list$

```
$merge\ []\ ys = ys$
$merge\ xs\ []\ = xs$
$merge\ (x \# xs)\ (y \# ys) =$
(if $x \le y$ then $x \# merge\ xs\ (y \# ys)$
 else $y \# merge\ (x \# xs)\ ys$)
```

$msort :: {}'a\ list \Rightarrow {}'a\ list$

```
$msort\ xs =$
(let $n = length\ xs$
 in if $n \le 1$ then $xs$
    else $merge\ (msort\ (take\ (n\ div\ 2)\ xs))$
            $(msort\ (drop\ (n\ div\ 2)\ xs)))$
```

Number of comparisons

$C\_merge :: 'a\ list \Rightarrow 'a\ list \Rightarrow nat$
$C\_msort :: 'a\ list \Rightarrow nat$

Lemma

$C\_merge\ xs\ ys$

Theorem

$length\ xs = 2^k \Longrightarrow C\_msort\ xs \le k * 2^k$

`HOL/Data_Structures/Sorting.thy`

Merge Sort

4 Merge Sort

Top-Down

Bottom-Up

$msort\_bu :: 'a\ list \Rightarrow 'a\ list$

$msort\_bu\ xs = merge\_all\ (map\ (\lambda x.\ [x])\ xs)$

$merge\_all :: 'a\ list\ list \Rightarrow 'a\ list$

$merge\_all\ [] = []$

$merge\_all\ [xs] = xs$

$merge\_all\ xss = merge\_all\ (merge\_adj\ xss)$

$merge\_adj :: 'a\ list\ list \Rightarrow 'a\ list\ list$

$merge\_adj\ [] = []$

$merge\_adj\ [xs] = [xs]$

$merge\_adj\ (xs\ \#\ ys\ \#\ zss) =$

$merge\ xs\ ys\ \#\ merge\_adj\ zss$

Number of comparisons

$C\_merge\_adj :: 'a\ list\ list \Rightarrow nat$
$C\_merge\_all :: 'a\ list\ list \Rightarrow nat$
$C\_msort\_bu :: 'a\ list \Rightarrow nat$

Theorem
$length\ xs = 2^k \Longrightarrow C\_msort\_bu\ xs \leq k * 2^k$

HOL/Data_Structures/Sorting.thy

Bottom-Up Merge Sort

Even better

Make use of already sorted subsequences

Example

Sorting	$[7, 3, 1, 2, 5]$:
do not start with	$[[7], [3], [1], [2], [5]]$
but with	$[[1, 3, 7], [2, 5]]$

Archive of Formal Proofs

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

Chapter 7

Binary Trees

HOL/Library/Tree.thy

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

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

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

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

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

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

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

Size

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

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

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

$|\langle\rangle|_1 = 1$
$|\langle l,\ \_,\ r\rangle|_1 = |l|_1 + |r|_1$

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

Warning: $|.|$ and $|.|_1$ only on slides

Height

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

$h(\langle\rangle) = 0$
$h(\langle l,\ \_,\ r\rangle) = max\ (h(l))\ (h(r)) + 1$

Warning: $h(.)$ only on slides

Lemma $h(t) \leq |t|$

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

Minimal height

$min\_height :: {}'a\ tree \Rightarrow nat$

$mh(\langle\rangle) = 0$
$mh(\langle l,\ \_,\ r\rangle) = min\ (mh(l))\ (mh(r)) + 1$

Warning: $mh(.)$ only on slides

Lemma $mh(t) \leq h(t)$

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

Complete tree

$complete :: 'a\ tree \Rightarrow bool$
$complete\ \langle\rangle\ =\ True$
$complete\ \langle l,\ \_,\ r\rangle\ =$
$(h(l) = h(r) \wedge complete\ l \wedge complete\ r)$

Lemma $complete\ t = (mh(t) = h(t))$

Lemma $complete\ t \Longrightarrow |t|_1 = 2^{h(t)}$

Lemma $\neg\ complete\ t \Longrightarrow |t|_1 < 2^{h(t)}$
Lemma $\neg\ complete\ t \Longrightarrow 2^{mh(t)} < |t|_1$

Corollary $|t|_1 = 2^{h(t)} \Longrightarrow complete\ t$
Corollary $|t|_1 = 2^{mh(t)} \Longrightarrow complete\ t$

Almost complete tree

$acomplete :: 'a\ tree \Rightarrow bool$

$acomplete\ t = (h(t) - mh(t) \leq 1)$

Almost complete trees have optimal height:
Lemma If $acomplete\ t$ and $|t| \leq |t'|$ then $h(t) \leq h(t')$.

Warning

- The terms *complete* and *almost complete* are not defined uniquely in the literature.

- For example,
  Knuth calls *complete* what we call *almost complete*.

Chapter 8

Search Trees

Most of the material focuses on
BSTs $=$ binary search trees

BSTs represent sets

Any tree represents a set:

$set\_tree :: 'a\ tree \Rightarrow 'a\ set$

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

A BST represents a set that can be searched in time $O(h(t))$

Function set_tree is called an *abstraction function*
because it maps the implementation
to the abstract mathematical object

bst

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

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

Type $'a$ must be in class $linorder$ ($'a :: linorder$) where $linorder$ are *linear orders* (also called *total orders*).

Note: nat, int and $real$ are in class $linorder$

Set interface

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

- An implementation type $'s$
- $empty :: \ 's$
- $insert :: \ 'a \Rightarrow \ 's \Rightarrow \ 's$
- $delete :: \ 'a \Rightarrow \ 's \Rightarrow \ 's$
- $isin :: \quad 's \Rightarrow \ 'a \Rightarrow bool$

Map interface

Instead of a set, a search tree can also implement a map from $'a$ to $'b$:

- An implementation type $'m$
- $empty :: 'm$
- $update :: 'a \Rightarrow 'b \Rightarrow 'm \Rightarrow 'm$
- $delete :: 'a \Rightarrow 'm \Rightarrow 'm$
- $lookup :: 'm \Rightarrow 'a \Rightarrow 'b\ option$

Sets are a special case of maps

Comparison of elements

We assume that the element type $'a$ is a linear order

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

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

$cmp\ x\ y =$
(if $x < y$ then LT else if $x = y$ then EQ else GT)

Implementation type: $'a\ tree$

$empty = Leaf$

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

$$
\begin{array}{l}
isin\ \langle\rangle\ x = False \\
isin\ \langle l,\ a,\ r\rangle\ x = (\textsf{case}\ cmp\ x\ a\ \textsf{of} \\
\qquad\qquad\qquad\qquad\quad LT \Rightarrow isin\ l\ x \\
\qquad\qquad\qquad\qquad\ \mid EQ \Rightarrow True \\
\qquad\qquad\qquad\qquad\ \mid GT \Rightarrow isin\ r\ x)
\end{array}
$$

```
$delete\ x\ \langle\rangle = \langle\rangle$
$delete\ x\ \langle l,\ a,\ r\rangle =$
(case $cmp\ x\ a$ of
   $LT \Rightarrow \langle delete\ x\ l,\ a,\ r\rangle$
 | $EQ \Rightarrow$ if $r = \langle\rangle$ then $l$
              else let $(a',\ r') = split\_min\ r$ in $\langle l,\ a',\ r'\rangle$
 | $GT \Rightarrow \langle l,\ a,\ delete\ x\ r\rangle$)

$split\_min\ \langle l,\ a,\ r\rangle =$
(if $l = \langle\rangle$ then $(a,\ r)$
 else let $(x,\ l') = split\_min\ l$ in $(x,\ \langle l',\ a,\ r\rangle)$)
```

Why is this implementation correct?

Because	$empty$	$insert$	$delete$	$isin$
simulate	$\{\}$	$\cup \{.\}$	$- \{.\}$	$\in$

$$set\_tree\ empty = \{\}$$
$$set\_tree\ (insert\ x\ t) = set\_tree\ t \cup \{x\}$$
$$set\_tree\ (delete\ x\ t) = set\_tree\ t - \{x\}$$
$$isin\ t\ x = (x \in set\_tree\ t)$$

Under the assumption $bst\ t$

Also: bst must be invariant

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

Key idea

Local definition:

$sorted$ means sorted w.r.t. $<$

No duplicates!

$\Longrightarrow$ $bst\ t$ can be expressed as $sorted(inorder\ t)$

$\boxed{\text{Conduct proofs on sorted lists, not sets}}$

Two kinds of invariants

- Unbalanced trees only need the invariant bst
- More efficient search trees come with additional *structural invariants* $=$ balance criteria.

Correctness via sorted lists

Correctness proofs of (almost) all search trees
covered in this course
can be automated.

Except for the structural invariants.

Therefore we concentrate on the latter.

For details see file See `HOL/Data_Structures/Set_Specs.thy` and T. Nipkow. *Automatic Functional Correctness Proofs for Functional Search Trees.* Interactive Theorem Proving, LNCS, 2016.

A methodological interlude:

A closer look at ADT principles
and their realization in Isabelle

Set and binary search tree as examples
(ignoring $delete$)

9 Abstract Data Types

Defining ADTs

Using ADTs

Implementing ADTs

$\text{ADT} = \textit{interface} + \textit{specification}$

Example (Set interface)

$empty :: {}'s$
$insert :: {}'a \Rightarrow {}'s \Rightarrow {}'s$
$isin :: \quad {}'s \Rightarrow {}'a \Rightarrow bool$

We assume that each ADT describes one

Type of Interest T

Above: $T = {}'s$

Model-oriented specification

Specify type T via a model $=$ existing HOL type A
Motto: T should behave like A

Specification of “behaves like” via an

- *abstraction function* $\alpha :: T \Rightarrow A$

Only some elements of T represent elements of A:

- *invariant* $invar :: T \Rightarrow bool$

α and $invar$ are part of the interface,
but only for specification and verification purposes

Example (Set ADT)

$empty ::$ ...

$insert ::$ ...

$isin ::$ ...

$set :: \quad 's \Rightarrow 'a\ set$ (name arbitrary)

$invar :: \quad 's \Rightarrow bool$ (name arbitrary)

$$\begin{array}{rl}
 & set\ empty = \{\} \\
invar\ s \Longrightarrow & set(insert\ x\ s) = set\ s \cup \{x\} \\
invar\ s \Longrightarrow & isin\ s\ x = (x \in set\ s) \\
 & invar\ empty \\
invar\ s \Longrightarrow & invar(insert\ x\ s)
\end{array}$$

In Isabelle: **locale**

locale $Set =$

fixes $empty :: 's$
fixes $insert :: 'a \Rightarrow 's \Rightarrow 's$
fixes $isin :: 's \Rightarrow 'a \Rightarrow bool$
fixes $set :: 's \Rightarrow 'a\ set$
fixes $invar :: 's \Rightarrow bool$
assumes $set\ empty = \{\}$
assumes $invar\ s \Longrightarrow isin\ s\ x = (x \in set\ s)$
assumes $invar\ s \Longrightarrow set(insert\ x\ s) = set\ s \cup \{x\}$
assumes $invar\ empty$
assumes $invar\ s \Longrightarrow invar(insert\ x\ s)$

See `HOL/Data_Structures/Set_Specs.thy`

Formally, in general

To ease notation, generalize α and $invar$ (conceptually):
α is the identity and $invar$ is $True$
on types other than T

Specification of each interface function f (on T):

- f must behave like some function f_A (on A):
  $invar\ t_1 \wedge \ldots \wedge invar\ t_n \Longrightarrow$
  $\alpha(f\ t_1\ \ldots\ t_n) = f_A\ (\alpha\ t_1)\ \ldots\ (\alpha\ t_n)$
  (α is a homomorphism)
- f must preserve the invariant:
  $invar\ t_1 \wedge \ldots \wedge invar\ t_n \Longrightarrow invar(f\ t_1\ \ldots\ t_n)$

The purpose of an ADT is to provide a context
for implementing generic algorithms
parameterized with the interface functions of the ADT.

Example

locale $Set =$
fixes ...
assumes ...
begin

fun set_of_list **where**
$set\_of\_list\ [] = empty\ |$
$set\_of\_list\ (x \# xs) = insert\ x\ (set\_of\_list\ xs)$

lemma $invar(set\_of\_list\ xs)$
by$(induction\ xs)$
  $(auto\ simp\colon invar\_empty\ invar\_insert)$

end

9 Abstract Data Types

Defining ADTs

Using ADTs

Implementing ADTs

1. Implement interface
2. Prove specification

Example

Define functions $isin$ and $insert$ on type $'a\ tree$ with invariant bst.

Now implement locale Set:

In Isabelle: **interpretation**

interpretation Set
where $empty = Leaf$ **and** $isin = isin$
and $insert = insert$ **and** $set = set\_tree$ **and** $invar = bst$
proof
  show $set\_tree\ Leaf = \{\}$ $\langle proof \rangle$
next
  fix s **assume** $bst\ s$
  show $set\_tree\ (insert\ x\ s) = set\_tree\ s \cup \{x\}$
  $\langle proof \rangle$
next
⋮
qed

Interpretation of Set also yields

- function $set\_of\_list :: 'a\ list \Rightarrow 'a\ tree$
- theorem $bst\ (set\_of\_list\ xs)$

Now back to search trees ...

HOL/Data_Structures/ Tree23_Set.thy

2-3 Trees

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

Abbreviations:

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

$isin$

```
$isin\ \langle l,\ a,\ m,\ b,\ r\rangle\ x =$
(case $cmp\ x\ a$ of
   $LT \Rightarrow isin\ l\ x$
 | $EQ \Rightarrow True$
 | $GT \Rightarrow$ case $cmp\ x\ b$ of
               $LT \Rightarrow isin\ m\ x$
             | $EQ \Rightarrow True$
             | $GT \Rightarrow isin\ r\ x$)
```

Assumes the usual ordering invariant

Structural invariant *complete*

All leaves are at the same level:

$complete\ \langle\rangle = True$

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

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

Lemma

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

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: $TI\ t$
- tree overflows: $OF\ l\ x\ r$

datatype $'a\ upI = \ TI\ ('a\ tree23)$
$\ \ |\ OF\ ('a\ tree23)\ 'a\ ('a\ tree23)$

$treeI :: 'a\ upI \Rightarrow 'a\ tree23$

$treeI\ (TI\ t) = t$
$treeI\ (OF\ l\ a\ r) = \langle l,\ a,\ r\rangle$

Insertion

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

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

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

Insertion

```
$ins\ x\ \langle\rangle\ =\ OF\ \langle\rangle\ x\ \langle\rangle$
$ins\ x\ \langle l,\ a,\ r\rangle\ =$
case $cmp\ x\ a$ of
  $LT \Rightarrow$ case $ins\ x\ l$ of
                 $TI\ l' \Rightarrow\ TI\ \langle l',\ a,\ r\rangle$
               $|\ OF\ l_1\ b\ l_2 \Rightarrow\ TI\ \langle l_1,\ b,\ l_2,\ a,\ r\rangle$
$|\ EQ \Rightarrow\ TI\ \langle l,\ a,\ r\rangle$
$|\ GT \Rightarrow$ case $ins\ x\ r$ of
                 $TI\ r' \Rightarrow\ TI\ \langle l,\ a,\ r'\rangle$
               $|\ OF\ r_1\ b\ r_2 \Rightarrow\ TI\ \langle l,\ a,\ r_1,\ b,\ r_2\rangle$
```

Insertion

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

Insertion preserves *complete*

Lemma

$complete\ t \Longrightarrow$
$complete\ (treeI\ (ins\ a\ t)) \wedge hI\ (ins\ a\ t) = h(t)$

where $hI :: 'a\ upI \Rightarrow nat$
$hI\ (TI\ t) = h(t)$
$hI\ (OF\ l\ a\ r) = h(l)$

Proof by induction on t. Base and step automatic.

Corollary

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

Deletion

The idea:

$$\begin{array}{lcl} Node3 & \rightsquigarrow & Node2 \\ Node2 & \rightsquigarrow & \text{underflow, height decreases by 1} \end{array}$$

Underflow: merge with siblings on the way up

Deletion

Two possible return values:

- height unchanged: $TD\ t$
- height decreased by 1: $UF\ t$

datatype $'a\ upD = TD\ ('a\ tree23) \mid UF\ ('a\ tree23)$

$treeD\ (TD\ t) = t$
$treeD\ (UF\ t) = t$

Deletion

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

$delete\ x\ t = treeD\ (del\ x\ t)$

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

Deletion

$$del\ x\ \langle\rangle = TD\ \langle\rangle$$

$$del\ x\ \langle\langle\rangle,\ a,\ \langle\rangle\rangle =$$

$$(\textsf{if}\ x = a\ \textsf{then}\ UF\ \langle\rangle\ \textsf{else}\ TD\ \langle\langle\rangle,\ a,\ \langle\rangle\rangle)$$

$$del\ x\ \langle\langle\rangle,\ a,\ \langle\rangle,\ b,\ \langle\rangle\rangle = \ldots$$

```
$del\ x\ \langle l,\ a,\ r\rangle =$
(case $cmp\ x\ a$ of
   $LT \Rightarrow node21\ (del\ x\ l)\ a\ r$
 | $EQ \Rightarrow$ let $(a',\ t) = split\_min\ r$ in $node22\ l\ a'\ t$
 | $GT \Rightarrow node22\ l\ a\ (del\ x\ r))$
```

$node21\ (TD\ t_1)\ a\ t_2 = TD\ \langle t_1,\ a,\ t_2\rangle$
$node21\ (UF\ t_1)\ a\ \langle t_2,\ b,\ t_3\rangle = UF\ \langle t_1,\ a,\ t_2,\ b,\ t_3\rangle$
$node21\ (UF\ t_1)\ a\ \langle t_2,\ b,\ t_3,\ c,\ t_4\rangle =$
$TD\ \langle\langle t_1,\ a,\ t_2\rangle,\ b,\ \langle t_3,\ c,\ t_4\rangle\rangle$

Analogous: $node22$

Deletion preserves $complete$

After 13 simple lemmas:

Lemma

$complete\ t \Longrightarrow complete\ (treeD\ (del\ x\ t))$

Corollary

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

Beyond 2-3 trees

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

Like 2-3 trees, but with many more cases

The general case:

B-trees and (a, b)-trees

HOL/Data_Structures/ RBT_Set.thy

Relationship to 2-3-4 trees

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

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

Red means “I am part of a bigger node”

Structural invariants

- The root is
- Every $\langle\rangle$ is considered Black.
- If a node is Red,
- All paths from a node to a leaf have the same number of

Red-black trees

datatype $color = Red \mid Black$

type_synonym $'a\ rbt = ('a \times color)\ tree$

Abbreviations:

$$\begin{array}{lcl} R\ l\ a\ r & \equiv & Node\ l\ (a,\ Red)\ r \\ B\ l\ a\ r & \equiv & Node\ l\ (a,\ Black)\ r \end{array}$$

Color

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

$$color\ \langle\rangle = Black$$
$$color\ \langle\_,\ (\_,\ c),\ \_\rangle = c$$

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

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

Structural invariants

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

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

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

$invc\ \langle\rangle = True$

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

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

$\ invc\ l \wedge invc\ r)$

Structural invariants

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

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

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

$$bh(\langle\rangle) = 0$$
$$bh(\langle l,\ (\_,\ c),\ \_\rangle) =$$
$$(\textsf{if}\ c = Black\ \textsf{then}\ bh(l) + 1\ \textsf{else}\ bh(l))$$

Logarithmic height

Lemma

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

Intuition: $h(t)\ /\ 2 \leq bh(t) \leq mh(t) \leq \log_2 |t|_1$

Insertion

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

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

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

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

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: $baliR$

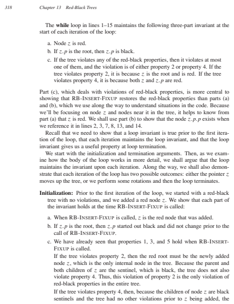
Preservation of invariant

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:

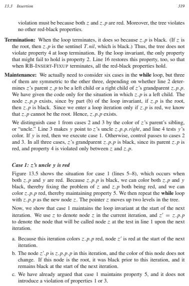
a. When RB-INSERT-FIXUP is called, z is the red node that was added.

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

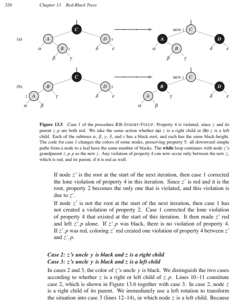
Figure 13.5 shows the situation for case 1 (lines 5–8), which occurs when both $z.p$ and y are red. Because $z.p.p$ is black, we can color both $z.p$ and y black, thereby fixing the problem of z and $z.p$ both being red, and we can color $z.p.p$ red, thereby maintaining property 5. We then repeat the **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.



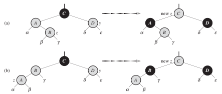


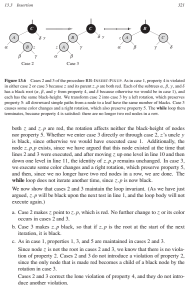
Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since z and its parent $z.p$ are both red. We take the same action whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. 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

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



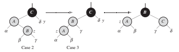


Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $z.p$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise we would be in case 1), and 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.

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 code

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

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

Deletion code

```
$split\_min\ \langle l,\ (a,\ \_),\ r\rangle =$
(if $l = \langle\rangle$ then $(a,\ r)$
 else let $(x,\ l') = split\_min\ l$
      in $(x,$ if $color\ l = Black$ then $baldL\ l'\ a\ r$
             else $R\ l'\ a\ r))$
```

```
$baldL\ (R\ t_1\ a\ t_2)\ b\ t_3 = R\ (B\ t_1\ a\ t_2)\ b\ t_3$
$baldL\ t_1\ a\ (B\ t_2\ b\ t_3) = baliR\ t_1\ a\ (R\ t_2\ b\ t_3)$
$baldL\ t_1\ a\ (R\ (B\ t_2\ b\ t_3)\ c\ t_4) =$
$R\ (B\ t_1\ a\ t_2)\ b\ (baliR\ t_3\ c\ (paint\ Red\ t_4))$
$baldL\ t_1\ a\ t_2 = R\ t_1\ a\ t_2$
```

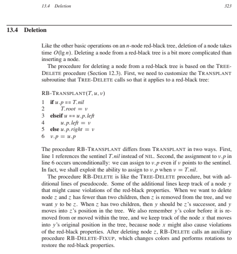
Deletion proof

After a number of lemmas:

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

$$\mathit{rbt}\ t \Longrightarrow \mathit{rbt}\ (\mathit{delete}\ x\ t)$$

Code and proof in CLRS



13.4 Deletion

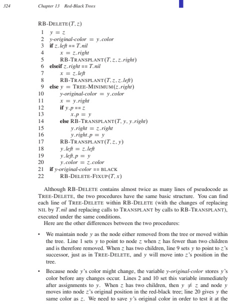
Like the other basic operations on an n-node red-black tree, deletion of a node takes time $O(\lg n)$. Deleting a node from a red-black tree is a bit more complicated than inserting a node.

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.

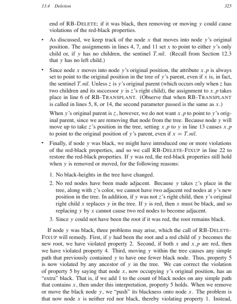


```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      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

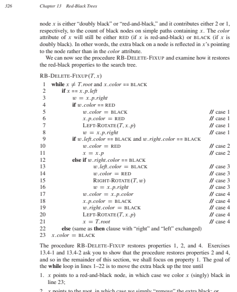


end of RB-DELETE; if it was black, then removing or moving y could cause violations of the red-black properties.

- 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,



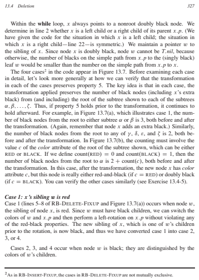
node x is either "doubly black" or "red-and-black," and it contributes either 2 or 1, respectively, to the count of black nodes on simple paths containing x. The *color* attribute of x will still be either RED (if x is red-and-black) or BLACK (if x is doubly black). In other words, the extra black on 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 ≠ 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;
2. x points to the root, in which case we simply "remove" the extra black; or



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.

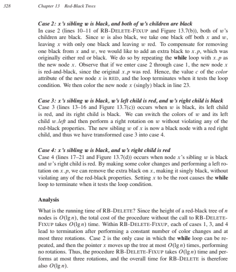
The four cases[2] in the code appear in Figure 13.7. Before examining each case in detail, let's look more generally at how we can verify that the transformation in each of the cases preserves property 5. The key idea is that in each case, the transformation applied preserves the number of black nodes (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 count(RED) $= 0$ and count(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 $c =$ RED) or doubly black (if $c =$ 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 partly based on:

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

12 More Search Trees

AVL Trees

[Adelson-Velskii & Landis 62]

- Every node $\langle l, \_, r\rangle$ must be balanced:
  $|h(l) - h(r)| \leq 1$
- Verified Isabelle implementation:
  `HOL/Data_Structures/AVL_Set.thy`

12 More Search Trees

Weight-Balanced Trees

[Nievergelt & Reingold 72,73]

- Parameter: balance factor $0 < \alpha \leq 0.5$
- Every node $\langle l, \_, r\rangle$ must be balanced:
  $\alpha \leq |l|_1 / (|l|_1 + |r|_1) \leq 1{-}\alpha$
- Insertion and deletion: single and double rotations depending on subtle numeric conditions
- Nievergelt and Reingold incorrect
- Mistakes discovered and corrected by [Blum & Mehlhorn 80] and [Hirai & Yamamoto 2011]
- Verified implementation
  in Isabelle's *Archive of Formal Proofs*.

AA trees

[Arne Andersson 93, Ragde 14]

- Simulation of 2-3 trees by binary trees
  $\langle t_1, a, t_2, b, t_3\rangle \;\rightsquigarrow\; \langle t_1, a, \langle t_2, b, t_3\rangle\rangle$
- Height field (or single bit) to distinguish single from double node
- Code short but opaque
- 4 bugs in $delete$ in [Ragde 14]:
  non-linear pattern; going down wrong subtree; missing function call; off by 1

AA trees

[Arne Andersson 93, Ragde 14]

After corrections, the proofs:

- Code relies on tricky pre- and post-conditions that need to be found
- Structural invariant preservation requires most of the work

Scapegoat trees

[Anderson 89, Igal & Rivest 93]

Central idea:

Don't rebalance every time,
Rebuild when the tree gets "too unbalanced"

- Tricky: amortized logarithmic complexity analysis
- Verified implementation
  in Isabelle's *Archive of Formal Proofs*.

One by one (Union)

Let $c(x) =$ cost of adding 1 element to set of size x

Cost of adding m elements to a set of n elements:

$$c(n) + \cdots + c(n + m - 1)$$

$\Longrightarrow$ choose $m \leq n \Longrightarrow$ smaller into bigger

If $c(x) = \log_2 x \Longrightarrow$
Cost $= O(m * \log_2(n + m)) = O(m * \log_2 n)$

Similar for intersection and difference.

- We can do better than $O(m * \log_2 n)$
- This section:

  A parallel divide and conquer approach
- Cost: $\Theta(m * \log_2(\frac{n}{m} + 1))$
- Works for many kinds of balanced trees
- For ease of presentation: use concrete type $tree$

Uniform $tree$ type

Red-Black trees, AVL trees, weight-balanced trees, etc can all be implemented with $'b$-augmented trees:

$('a \times \ 'b) \ tree$

We work with this type of trees without committing to any particular kind of balancing schema.

Just $join$

Can synthesize all BST interface functions from just one function:

$$join\ l\ a\ r\ \approx\ Node\ l\ (a,\ \_)\ r + \text{rebalance}$$

Thus $join$ determines the balancing schema

Just $join$

Given $join :: tree \Rightarrow {'a} \Rightarrow tree \ \Rightarrow tree$
(where $tree$ abbreviates $('a,'b)\ tree$), implement

$union :: tree \Rightarrow tree \Rightarrow tree$
$inter :: tree \Rightarrow tree \Rightarrow tree$
$diff :: tree \Rightarrow tree \Rightarrow tree$

```
$union\ t_1\ t_2 =$
(if $t_1 = \langle\rangle$ then $t_2$
 else if $t_2 = \langle\rangle$ then $t_1$
      else case $t_1$ of
              $\langle l_1, (a, b), r_1\rangle \Rightarrow$
                let $(l_2, x, r_2) = split\ t_2\ a;$
                    $l' = union\ l_1\ l_2;$
                    $r' = union\ r_1\ r_2$
                in $join\ l'\ a\ r'$)
```

```
$split :: tree \Rightarrow {}'a \Rightarrow tree \times bool \times tree$
$split\ \langle\rangle\ \_ = (\langle\rangle,\ False,\ \langle\rangle)$
$split\ \langle l,\ (a,\ \_),\ r\rangle\ x =$
(case $cmp\ x\ a$ of
   $LT \Rightarrow$
     let $(l_1,\ b,\ l_2) = split\ l\ x$
     in $(l_1,\ b,\ join\ l_2\ a\ r)$
 | $EQ \Rightarrow (l,\ True,\ r)$
 | $GT \Rightarrow$
     let $(r_1,\ b,\ r_2) = split\ r\ x$
     in $(join\ l\ a\ r_1,\ b,\ r_2)$)
```

```
$inter\ t_1\ t_2 =$
(if $t_1 = \langle\rangle$ then $\langle\rangle$
 else if $t_2 = \langle\rangle$ then $\langle\rangle$
      else case $t_1$ of
             $\langle l_1, (a, x), r_1\rangle \Rightarrow$
               let $(l_2, b, r_2) = split\ t_2\ a;$
                   $l' = inter\ l_1\ l_2;$
                   $r' = inter\ r_1\ r_2$
               in if $b$ then $join\ l'\ a\ r'$
                  else $join2\ l'\ r'$)
```

$join2 :: tree \Rightarrow tree \Rightarrow tree$

$join2\ l\ r =$
(if $r = \langle\rangle$ then l
 else let $(m,\ r') = split\_min\ r$ in $join\ l\ m\ r'$)

$split\_min :: tree \Rightarrow {}'a \times tree$

$split\_min\ \langle l,\ (a,\ \_),\ r\rangle =$
(if $l = \langle\rangle$ then $(a,\ r)$
 else let $(m,\ l') = split\_min\ l$ in $(m,\ join\ l'\ a\ r)$)

```
$diff\ t_1\ t_2 =$
(if $t_1 = \langle\rangle$ then $\langle\rangle$
 else if $t_2 = \langle\rangle$ then $t_1$
       else case $t_2$ of
               $\langle l_2,\ (a,\ b),\ r_2\rangle \Rightarrow$
                 let $(l_1,\ x,\ r_1) = split\ t_1\ a;$
                     $l' = diff\ l_1\ l_2;$
                     $r' = diff\ r_1\ r_2$
                 in $join2\ l'\ r'$)
```

$insert$ and $delete$

$$insert\ x\ t = (\textsf{let}\ (l,\ b,\ r) = split\ t\ x\ \textsf{in}\ join\ l\ x\ r)$$

$$delete\ x\ t = (\textsf{let}\ (l,\ b,\ r) = split\ t\ x\ \textsf{in}\ join2\ l\ r)$$

13 Union, Intersection, Difference on BSTs

Correctness

Join for Red-Black Trees

Specification of $join$ and inv

- $set\_tree\ (join\ l\ a\ r) = set\_tree\ l \cup \{a\} \cup set\_tree\ r$
- $bst\ \langle l,\ (a,\ b),\ r\rangle \Longrightarrow bst\ (join\ l\ a\ r)$

Also required: structural invariant inv:

- $inv\ \langle\rangle$
- $inv\ \langle l,\ (a,\ b),\ r\rangle \Longrightarrow inv\ l \wedge inv\ r$
- $[\![inv\ l;\ inv\ r]\!] \Longrightarrow inv\ (join\ l\ a\ r)$

Locale context for def of $union$ etc

Specification of $union$, $inter$, $diff$

ADT/Locale $Set2$ = extension of locale Set with

- $union$, $inter$, $diff$ $:: 's \Rightarrow 's \Rightarrow 's$
- $[\![invar\ s_1;\ invar\ s_2]\!]$
  $\Longrightarrow set\ (union\ s_1\ \ s_2) = set\ s_1 \cup set\ s_2$
- $[\![invar\ s_1;\ invar\ s_2]\!] \Longrightarrow invar\ (union\ s_1\ \ s_2)$
- ... $inter$ ...
- ... $diff$ ...

We focus on $union$.

See `HOL/Data_Structures/Set_Specs.thy`

Correctness lemmas for $union$ etc code

In the context of $join$ specification:

- $bst\ t_2 \Longrightarrow$
  $set\_tree\ (union\ t_1\ t_2) = set\_tree\ t_1 \cup set\_tree\ t_2$
- $[\![bst\ t_1;\ bst\ t_2]\!] \Longrightarrow bst\ (union\ t_1\ t_2)$
- $[\![inv\ t_1;\ inv\ t_2]\!] \Longrightarrow inv\ (union\ t_1\ t_2)$

Proofs automatic (more complex for $inter$ and $diff$)

Implementation of locale $Set2$:

interpretation $Set2$ **where** $union = union$ ...
and $set = set\_tree$ **and** $invar = (\lambda t.\ bst\ t \wedge inv\ t)$

HOL/Data_Structures/ Set2_Join.thy

⓭ Union, Intersection, Difference on BSTs

Correctness

Join for Red-Black Trees

$join\ l\ a\ r$ — The idea

Assume l is “smaller” than r :

- Descend along the left spine of r
  until you find a subtree t of the same “size” as l.
- Replace t by $\langle l, a, t \rangle$.
- Rebalance on the way up.

```
$join\ l\ x\ r =$
(if $bheight\ r < bheight\ l$
 then $paint\ Black\ (joinR\ l\ x\ r)$
 else if $bheight\ l < bheight\ r$
      then $paint\ Black\ (joinL\ l\ x\ r)$ else $B\ l\ x\ r)$

$joinL\ l\ x\ r =$
(if $bheight\ r \leq bheight\ l$ then $R\ l\ x\ r$
 else case $r$ of
        $\langle l', (x', Red), r'\rangle \Rightarrow R\ (joinL\ l\ x\ l')\ x'\ r'$
      | $\langle l', (x', Black), r'\rangle \Rightarrow baliL\ (joinL\ l\ x\ l')\ x'\ r')$
```

Need to store black height in each node
for logarithmic complexity

Thys/Set2_Join_RBT.thy

Literature

The idea of "just $join$":

Stephen Adams. *Efficient Sets — A Balancing Act*.
J. Functional Programming, volume 3, number 4, 1993.

The precise analysis:

Guy E. Blelloch, D. Ferizovic, Y. Sun.
Just Join for Parallel Ordered Sets.
ACM Symposium on Parallelism in Algorithms and Architectures 2016.

Trie

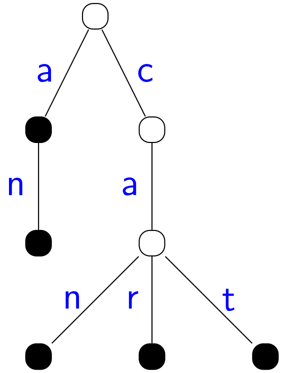
[Fredkin, CACM 1960]

Name: *reTRIEval*

- Tries are search trees indexed by lists
- Tries are tree-shaped DFAs

Example Trie

$\{$ a, an, can, car, cat $\}$



14 Tries and Patricia Tries

Tries via Functions

Binary Tries and Patricia Tries

HOL/Data_Structures/ Trie_Fun.thy

Trie

datatype $'a\ trie = Nd\ bool\ ('a \Rightarrow 'a\ trie\ option)$

Function update notation:

$f(a := b) = (\lambda x.\ \textsf{if}\ x = a\ \textsf{then}\ b\ \textsf{else}\ f\ x)$

$f(a \mapsto b) = f(a := Some\ b)$

Next: Implementation of ADT Set

$empty$

$$empty = Nd\ False\ (\lambda\_.\ None)$$

$isin$

$$isin\ (Nd\ b\ m)\ []\ =\ b$$

$$\begin{aligned} isin\ (Nd\ b\ m)\ (k\ \#\ xs)\ =\ (&\textsf{case}\ m\ k\ \textsf{of} \\ &\ \ None \Rightarrow False \\ &\mid Some\ t \Rightarrow isin\ t\ xs) \end{aligned}$$

$insert$

$insert\ []\ (Nd\ b\ m) = Nd\ True\ m$

```
$insert\ (x \# xs)\ (Nd\ b\ m) =$
let $s$ = case $m\ x$ of
            $None \Rightarrow empty$
          | $Some\ t \Rightarrow t$
in $Nd\ b\ (m(x \mapsto insert\ xs\ s))$
```

$delete$

$$delete\ []\ (Nd\ b\ m) = Nd\ False\ m$$

$$\begin{aligned}
&delete\ (x \mathbin{\#} xs)\ (Nd\ b\ m) = \\
&Nd\ b\ (\text{case } m\ x \text{ of} \\
&\qquad\qquad None \Rightarrow m \\
&\qquad\quad \mid Some\ t \Rightarrow m(x \mapsto delete\ xs\ t))
\end{aligned}$$

Does not shrink trie — exercise!

Correctness: Abstraction function

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

$set\ t = \{xs.\ isin\ t\ xs\}$

Invariant is $True$

Correctness theorems

- $set\ empty = \{\}$
- $isin\ t\ xs = (xs \in set\ t)$
- $set\ (insert\ xs\ t) = set\ t \cup \{xs\}$
- $set\ (delete\ xs\ t) = set\ t - \{xs\}$

No lemmas required

Abstraction function via $isin$

$$set\ t = \{xs.\ isin\ t\ xs\}$$

- Trivial definition
- Reusing code ($isin$) *may* complicate proofs.
- Separate abstract mathematical definition *may* simplify proofs

Also possible for some other ADTs, e.g. for Map:
$lookup :: {'t} \Rightarrow ({'a} \Rightarrow {'b}\ option)$

14 Tries and Patricia Tries

Tries via Functions

Binary Tries and Patricia Tries

HOL/Data_Structures/ Tries_Binary.thy

Trie

datatype $trie = Lf \mid Nd\ bool\ (trie \times trie)$

Auxiliary functions on pairs:

$sel2 :: bool \Rightarrow {}'a \times {}'a \Rightarrow {}'a$
$sel2\ b\ (a_1,\ a_2) = (\text{if } b \text{ then } a_2 \text{ else } a_1)$

$mod2 :: ({}'a \Rightarrow {}'a) \Rightarrow bool \Rightarrow {}'a \times {}'a \Rightarrow {}'a \times {}'a$
$mod2\ f\ b\ (a_1,\ a_2) = (\text{if } b \text{ then } (a_1,\ f\ a_2) \text{ else } (f\ a_1,\ a_2))$

$empty$

$$empty = Lf$$

$isin$

$isin\ Lf\ ks = False$

$$\begin{aligned} isin\ (Nd\ b\ lr)\ ks = (&\text{case } ks \text{ of} \\ &\quad [] \Rightarrow b \\ &\ \mid k \# x \Rightarrow isin\ (sel2\ k\ lr)\ x) \end{aligned}$$

insert

$insert\ []\ Lf = Nd\ True\ (Lf,\ Lf)$

$insert\ []\ (Nd\ b\ lr) = Nd\ True\ lr$

$insert\ (k\ \#\ ks)\ Lf =$
$Nd\ False\ (mod2\ (insert\ ks)\ k\ (Lf,\ Lf))$

$insert\ (k\ \#\ ks)\ (Nd\ b\ lr) =$
$Nd\ b\ (mod2\ (insert\ ks)\ k\ lr)$

$delete$

$delete\ ks\ Lf = Lf$

$delete\ ks\ (Nd\ b\ lr) =$
case ks of
  $[] \Rightarrow node\ False\ lr$
$\mid k \# ks' \Rightarrow node\ b\ (mod2\ (delete\ ks')\ k\ lr)$

Shrink trie if possible:

$node\ b\ lr =$ (if $\neg\ b \wedge lr = (Lf, Lf)$ then Lf else $Nd\ b\ lr$)

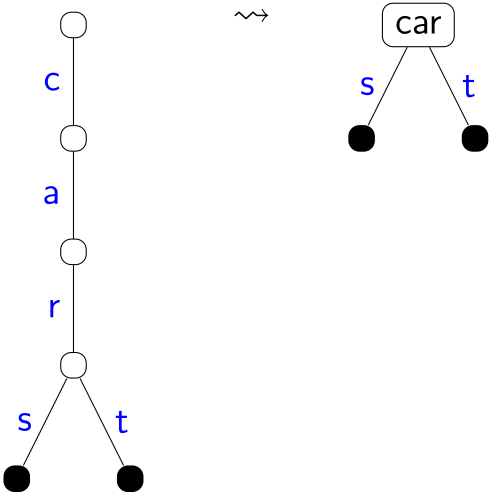
Correctness of implementation

Abstraction function:

$$set\_trie\ t = \{xs.\ isin\ t\ xs\}$$

- $isin\ (insert\ xs\ t)\ ys = (xs = ys \vee isin\ t\ ys)$
  $\Longrightarrow\ set\_trie\ (insert\ xs\ t) = set\_trie\ t \cup \{xs\}$
- $isin\ (delete\ xs\ t)\ ys = (xs \neq ys \wedge isin\ t\ ys)$
  $\Longrightarrow\ set\_trie\ (delete\ xs\ t) = set\_trie\ t - \{xs\}$

From tries to Patricia tries



Patricia trie

datatype $trieP = LfP$
$\mid NdP\ (bool\ list)\ bool\ (trieP \times trieP)$

$isinP$

```
$isinP\ LfP\ ks = False$

$isinP\ (NdP\ ps\ b\ lr)\ ks =$
(let $n = length\ ps$
 in if $ps = take\ n\ ks$
    then case $drop\ n\ ks$ of
              $[] \Rightarrow b$
            $|\ k\ \#\ ks' \Rightarrow isinP\ (sel2\ k\ lr)\ ks'$
    else $False$)
```

Splitting lists

$split\ xs\ ys = (zs,\ xs',\ ys')$
iff zs is the longest common prefix of xs and ys
and xs'/ys' is the remainder of xs/ys

$insertP$

```
$insertP\ ks\ LfP = NdP\ ks\ True\ (LfP,\ LfP)$

$insertP\ ks\ (NdP\ ps\ b\ lr) =$
case $split\ ks\ ps$ of
  $(qs,\ [],\ []) \Rightarrow NdP\ ps\ True\ lr$
| $(qs,\ [],\ p\ \#\ ps') \Rightarrow$
    let $t = NdP\ ps'\ b\ lr$
    in $NdP\ qs\ True$ (if $p$ then $(LfP,\ t)$ else $(t,\ LfP)$)
| $(qs,\ k\ \#\ ks',\ []) \Rightarrow NdP\ ps\ b\ (mod2\ (insertP\ ks')\ k\ lr)$
| $(qs,\ k\ \#\ ks',\ p\ \#\ ps') \Rightarrow$
    let $tp = NdP\ ps'\ b\ lr;\ tk = NdP\ ks'\ True\ (LfP,\ LfP)$
    in $NdP\ qs\ False$ (if $k$ then $(tp,\ tk)$ else $(tk,\ tp)$)
```

$deleteP$

```
$deleteP\ ks\ LfP = LfP$

$deleteP\ ks\ (NdP\ ps\ b\ lr) =$
$(case\ split\ ks\ ps\ of$
  $(qs,\ ks',\ p\#ps') \Rightarrow NdP\ ps\ b\ lr\ |$
  $(qs,\ k\#ks',\ []) \Rightarrow$
    $nodeP\ ps\ b\ (mod2\ (deleteP\ ks')\ k\ lr)\ |$
  $(qs,\ [],\ []) \Rightarrow nodeP\ ps\ False\ lr)$
```

Stepwise data refinement

View $trieP$ as an implementation ("refinement") of $trie$

Type	Abstraction function
$bool\ list\ set$	
$\uparrow$	set_trie
$trie$	
$\uparrow$	abs_trieP
$trieP$	

$\Longrightarrow$ Modular correctness proof of $trieP$

$abs\_trieP :: trieP \Rightarrow trie$

$abs\_trieP\ LfP = Lf$

$abs\_trieP\ (NdP\ ps\ b\ (l,\ r)) =$
$prefix\_trie\ ps\ (Nd\ b\ (abs\_trieP\ l,\ abs\_trieP\ r))$

$prefix\_trie :: bool\ list \Rightarrow trie \Rightarrow trie$

Correctness of $trieP$ w.r.t. $trie$

- $isinP\ t\ ks = isin\ (abs\_trieP\ t)\ ks$
- $abs\_trieP\ (insertP\ ks\ t) = insert\ ks\ (abs\_trieP\ t)$
- $abs\_trieP\ (deleteP\ ks\ t) = delete\ ks\ (abs\_trieP\ t)$

$isin\ (prefix\_trie\ ps\ t)\ ks =$
$(ps = take\ (length\ ps)\ ks \wedge isin\ t\ (drop\ (length\ ps)\ ks))$
$prefix\_trie\ ks\ (Nd\ True\ (Lf,\ Lf)) = insert\ ks\ Lf$
$insert\ ps\ (prefix\_trie\ ps\ (Nd\ b\ lr)) = prefix\_trie\ ps\ (Nd\ True\ lr)$
$insert\ (ks\ @\ ks')\ (prefix\_trie\ ks\ t) = prefix\_trie\ ks\ (insert\ ks'\ t)$
$prefix\_trie\ (ps\ @\ qs)\ t = prefix\_trie\ ps\ (prefix\_trie\ qs\ t)$
$split\ ks\ ps = (qs,\ ks',\ ps') \Longrightarrow$
$ks = qs\ @\ ks' \wedge ps = qs\ @\ ps' \wedge (ks' \neq [] \wedge ps' \neq [] \longrightarrow hd\ ks' \neq hd\ ps')$
$(prefix\_trie\ xs\ t = Lf) = (xs = [] \wedge t = Lf)$
$(abs\_trieP\ t = Lf) = (t = LfP)$
$delete\ xs\ (prefix\_trie\ xs\ (Nd\ b\ (l,\ r))) =$
(if $(l,\ r) = (Lf,\ Lf)$ then Lf else $prefix\_trie\ xs\ (Nd\ False\ (l,\ r))$)
$delete\ (xs\ @\ ys)\ (prefix\_trie\ xs\ t) =$
(if $delete\ ys\ t = Lf$ then Lf else $prefix\_trie\ xs\ (delete\ ys\ t)$)

Correctness of $trieP$ w.r.t. $bool\ list\ set$

Define $set\_trieP = set\_trie \circ abs\_trieP$

$\Longrightarrow$ Overall correctness by trivial composition of correctness theorems for $trie$ and $trieP$

Example:

$set\_trieP\ (insertP\ xs\ t) = set\_trieP\ t \cup \{xs\}$

follows directly from

$abs\_trieP\ (insertP\ ks\ t) = insert\ ks\ (abs\_trieP\ t)$
$set\_trie\ (insert\ xs\ t) = set\_trie\ t \cup \{xs\}$

Chapter 9

Priority Queues

Priority queue informally

Collection of elements with priorities

Operations:

- empty
- emptiness test
- insert
- get element with minimal priority
- delete element with minimal priority

We focus on the priorities:
element = priority

Priority queues are multisets

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
  A bit tricky 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 $invar\ q \Longrightarrow$

$mset\ empty = \{\#\}$

$is\_empty\ q = (mset\ q = \{\#\})$

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

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

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

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

Terminology

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

$$
\begin{array}{l}
heap\ \langle\rangle = True \\
heap\ \langle l,\ m,\ r\rangle = \\
((\forall\, x{\in}set\_tree\ l \cup set\_tree\ r.\ m \leq x) \wedge \\
\ heap\ l \wedge heap\ r)
\end{array}
$$

The term “heap” is frequently used synonymously with “priority queue”.

Priority queue via heap

- $empty = \langle\rangle$
- $is\_empty\ h = (h = \langle\rangle)$
- $get\_min\ \langle\_,\ a,\ \_\rangle = a$
- Assume we have $merge$
- $insert\ a\ t = merge\ \langle\langle\rangle,\ a,\ \langle\rangle\rangle\ t$
- $del\_min\ \langle l,\ a,\ r\rangle = merge\ l\ r$

Priority queue via heap

A naive merge:

```
$merge\ t_1\ t_2 =$ (case $(t_1,t_2)$ of
    $(\langle\rangle,\ \_) \Rightarrow t_2\ |$
    $(\_,\ \langle\rangle) \Rightarrow t_1\ |$
    $(\langle l_1,a_1,r_1\rangle,\ \langle l_2,a_2,r_2\rangle) \Rightarrow$
        if $a_1 \le a_2$ then $\langle merge\ l_1\ r_1,\ a_1,\ t_2\rangle$
        else $\langle t_1,\ a_2,\ merge\ l_2\ r_2\rangle$
```

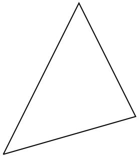
Challenge: how to maintain some kind of balance

HOL/Data_Structures/ Leftist_Heap.thy

Leftist tree informally

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

$\Longrightarrow$ m.h. $=$ length of right spine



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

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

Implementation type

type_synonym $'a\ lheap = ('a \times nat)\ tree$

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

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

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

$mht :: {}'a\ lheap \Rightarrow nat$

$mht\ \langle\rangle = 0$
$mht\ \langle\_,\ (\_,\ n),\ \_\rangle = n$

Leftist heap invariant

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

$merge$

Principle: descend on the right

```
$merge\ \langle\rangle\ t = t$
$merge\ t\ \langle\rangle = t$
$merge\ (\langle l_1, (a_1, \_), r_1\rangle =: t_1)\ (\langle l_2, (a_2, \_), r_2\rangle =: t_2) =$
(if $a_1 \leq a_2$ then $node\ l_1\ a_1\ (merge\ r_1\ t_2)$
 else $node\ l_2\ a_2\ (merge\ t_1\ r_2)$)
```

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

```
$node\ l\ a\ r =$
(let $mhl = mht\ l$; $mhr = mht\ r$
 in if $mhr \leq mhl$ then $\langle l, (a, mhr + 1), r\rangle$
    else $\langle r, (a, mhl + 1), l\rangle$)
```

$merge$

$$
\begin{array}{l}
merge\ (\langle l_1,\ (a_1,\ n_1),\ r_1\rangle =:\ t_1)\\
\ (\langle l_2,\ (a_2,\ n_2),\ r_2\rangle =:\ t_2) =\\
(\mathsf{if}\ a_1 \le a_2\ \mathsf{then}\ node\ l_1\ a_1\ (merge\ r_1\ t_2)\\
\ \mathsf{else}\ node\ l_2\ a_2\ (merge\ t_1\ r_2))
\end{array}
$$

Function $merge$ terminates because
decreases with every recursive call.

Functional correctness proofs

including preservation of $invar$

Straightforward

Logarithmic complexity

Correlation of rank and size:

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

Complexity measures T_merge, T_insert T_del_min: count calls of $merge$.

Lemma $[\![ltree\ l;\ ltree\ r]\!]$
$\Longrightarrow T\_merge\ l\ r \leq mh(l) + mh(r) + 1$

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

Corollary
$ltree\ t \Longrightarrow T\_insert\ x\ t \leq \log_2 |t|_1 + 3$

Corollary
$ltree\ t \Longrightarrow T\_del\_min\ t \leq 2 * \log_2 |t|_1 + 1$

Can we avoid the height info in each node?

Archive of Formal Proofs

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

What is a Braun tree?

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

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

1

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

Idea of invariant maintenance

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

Let $t = \langle l, x, r\rangle$. Assume $braun\ t$

Add element: to r, then swap subtrees: $t' = \langle r', x, l\rangle$
To prove $braun\ t'$: $|l| \leq |r'| \wedge |r'| \leq |l| + 1$ $\square$

Delete element: from l, then swap subtrees: $t' = \langle r, x, l'\rangle$
To prove $braun\ t'$: $|l'| \leq |r| \wedge |r| \leq |l'| + 1$ $\square$

Priority queue implementation

Implementation type: $'a\ tree$

Invariants: $heap$ and $braun$

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

$insert$

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

Correctness and preservation of invariant straightforward.

del_min

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

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

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

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

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

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

Reminiscent of heapsort, but not quite ...

del_left

$del\_left :: {}'a\ tree \Rightarrow {}'a \times {}'a\ tree$

$del\_left\ \langle\langle\rangle,\ x,\ r\rangle = (x,\ r)$

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

$(\text{let } (y,\ l') = del\_left\ l \text{ in } (y,\ \langle r,\ x,\ l'\rangle))$

$sift\_down$

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

$sift\_down\ \langle\rangle\ a\ \_ = \langle\langle\rangle,\ a,\ \langle\rangle\rangle$

$sift\_down\ \langle\langle\rangle,\ x,\ \_\rangle\ a\ \langle\rangle =$

$(\textsf{if}\ a \le x\ \textsf{then}\ \langle\langle\langle\rangle,\ x,\ \langle\rangle\rangle,\ a,\ \langle\rangle\rangle$

$\ \textsf{else}\ \langle\langle\langle\rangle,\ a,\ \langle\rangle\rangle,\ x,\ \langle\rangle\rangle)$

$sift\_down\ (\langle l_1,\ x_1,\ r_1\rangle =: t_1)\ a\ (\langle l_2,\ x_2,\ r_2\rangle =: t_2) =$

$\textsf{if}\ a \le x_1 \wedge a \le x_2\ \textsf{then}\ \langle t_1,\ a,\ t_2\rangle$

$\textsf{else if}\ x_1 \le x_2\ \textsf{then}\ \langle sift\_down\ l_1\ a\ r_1,\ x_1,\ t_2\rangle$

$\qquad \textsf{else}\ \langle t_1,\ x_2,\ sift\_down\ l_2\ a\ r_2\rangle$

Maintains $braun$

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.

Sorting with priority queue

$pq\ [] = empty$
$pq\ (x\#xs) = insert\ x\ (pq\ xs)$

$mins\ q =$
(if $is\_empty\ q$ then $[]$
 else $get\_min\ h\ \#\ mins\ (del\_min\ h)$)

$sort\_pq = mins \circ pq$

Complexity of $sort$: $O(n \log n)$
if all priority queue functions have complexity $O(\log n)$

HOL/Data_Structures/ Binomial_Heap.thy

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 $'a\ tree =$
$\quad Node\ (rank\colon nat)\ (root\colon 'a)\ ('a\ tree\ list)$

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

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

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 $i{+}1$

$link\ (Node\ r\ x_1\ ts_1 =: t_1)\ (Node\ r'\ x_2\ ts_2 =: t_2) =$
$(\textsf{if}\ x_1 \leq x_2\ \textsf{then}\ Node\ (r + 1)\ x_1\ (t_2\ \#\ ts_1)$
$\ \textsf{else}\ Node\ (r + 1)\ x_2\ (t_1\ \#\ ts_2))$

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:

$invar\ ts =$
$((\forall\ t{\in}set\ ts.\ invar\_tree\ t) \wedge$
$\ sorted\_wrt\ (<)\ (map\ rank\ ts))$

Inserting a tree into a heap

Intuition: propagate a carry

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

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

$merge$

```
$merge\ ts_1\ [] = ts_1$
$merge\ []\ ts_2 = ts_2$
$merge\ (t_1 \# ts_1 =: h_1)\ (t_2 \# ts_2 =: h_2) =$
($\textsf{if}$ $rank\ t_1 < rank\ t_2$ $\textsf{then}$ $t_1 \# merge\ ts_1\ h_2$
 $\textsf{else if}$ $rank\ t_2 < rank\ t_1$ $\textsf{then}$ $t_2 \# merge\ h_1\ ts_2$
      $\textsf{else}$ $ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2)$)
```

Intuition: Addition of binary numbers

Note: Handling of carry *after* recursive call

Get/delete minimum element

All trees are min-heaps.

Smallest element may be any root node:

$ts \neq [] \Longrightarrow get\_min\ ts = Min\ (set\ (map\ root\ ts))$

Similar:

$get\_min\_rest :: 'a\ tree\ list \Rightarrow 'a\ tree \times 'a\ tree\ list$

Returns tree with minimal root, and remaining trees

$del\_min\ ts =$
(case $get\_min\_rest\ ts$ of
  $(Node\ r\ x\ ts_1,\ ts_2) \Rightarrow merge\ (rev\ ts_1)\ ts_2)$

Why rev? Rank decreasing in ts_1 but increasing in ts_2

Complexity

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

Similarly for heap: $|t| = 2^{rank\ t}$

Complexity of operations: linear in length of heap

i.e. logarithmic in number of elements

$(invar\ ts \Longrightarrow length\ ts \leq \log_2\ (|ts| + 1))$

Proofs: straightforward?

Complexity of $merge$

```
$merge\ (t_1\ \#\ ts_1 =: h_1)\ (t_2\ \#\ ts_2 =: h_2) =$
(if $rank\ t_1 < rank\ t_2$ then $t_1\ \#\ merge\ ts_1\ h_2$
 else if $rank\ t_2 < rank\ t_1$ then $t_2\ \#\ merge\ h_1\ ts_2$
      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 $length\ ts_1 = length\ ts_2 = n$) can lead to calls of ins_tree on lists of length 1, ..., n.

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

Complexity of $merge$

```
$merge\ (t_1 \# ts_1 =: h_1)\ (t_2 \# ts_2 =: h_2) =$
(if $rank\ t_1 < rank\ t_2$ then $t_1 \# merge\ ts_1\ h_2$
 else if $rank\ t_2 < rank\ t_1$ then $t_2 \# merge\ h_1\ ts_2$
      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!

Sources

The inventor of the binomial heap:

Jean Vuillemin.
A Data Structure for Manipulating Priority Queues.
CACM, 1978.

The functional version:

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

Priority queues so far

$insert$, del_min (and $merge$) have logarithmic complexity

Skew Binomial Heap

Similar to binomial heap, but involving also *skew binary numbers*:

$d_1 \dots d_n$ represents $\sum_{i=1}^{n} d_i * (2^{i+1} - 1)$
where $d_i \in \{0, 1, 2\}$

Complexity

Skew binomial heap:

$insert$ in time $O(1)$
del_min and $merge$ still $O(\log n)$

Fibonacci heap (imperative!):

$insert$ and $merge$ in time $O(1)$
del_min still $O(\log n)$

Every operation in time $O(1)$?

Puzzle

Design a functional queue
with (worst case) constant time enq and deq functions

Chapter 10

Amortized Complexity

Example

n increments of a binary counter starting with 0

- WCC of one increment? $O(\log_2 n)$
- WCC of n increments? $O(n * \log_2 n)$
- $O(n * \log_2 n)$ is too pessimistic!
- Every second increment is cheap and compensates for the more expensive increments
- Fact: WCC of n increments is $O(n)$

WCC $=$ worst case complexity

The problem

WCC of individual operations
may lead to overestimation of
WCC of sequences of operations

Amortized analysis

Idea:

Try to determine the average cost of each operation
(in the worst case!)

Use cheap operations to pay for expensive ones

Method:

- Cheap operations pay extra (into a "bank account"), making them more expensive
- Expensive operations withdraw money from the account, making them cheaper

Bank account = *Potential*

- The potential ("credit") is implicitly "stored" in the data structure.
- Potential $\Phi :: \mathit{data\text{-}structure} \Rightarrow \mathit{non\text{-}neg.\ number}$ tells us how much credit is stored in a data structure
- Increase in potential = deposit to pay for *later* expensive operation
- Decrease in potential = withdrawal to pay for expensive operation

Back to example: counter

Increment:

- Actual cost: 1 for each bit flip
- Bank transaction:
    - pay in 1 for final $0 \to 1$ flip
    - take out 1 for each $1 \to 0$ flip

$\Longrightarrow$ increment has amortized cost 2 $=$ 1$+$1

Formalization via potential:

Φ $counter$ $=$ the number of 1's in $counter$

20 Amortized Complexity

Data structure

Given an implementation:

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

Needed for complexity analysis:

- Time/cost: $T\_f :: \tau \Rightarrow num$
  ($num =$ some numeric type
  nat may be inconvenient)
- Potential $\Phi :: \tau \Rightarrow num$ (creative spark!)

Need to prove: $\Phi\ s \geq 0$ and $\Phi\ init = 0$

Amortized and real cost

Sequence of operations: f_1, ..., f_n

Sequence of states:

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

Amortized cost := real cost + potential difference

$$A_{i+1} := T\_f_{i+1}\ s_i + \Phi\ s_{i+1} - \Phi\ s_i$$

$$\Longrightarrow$$

Sum of amortized costs $\geq$ sum of real costs

$$\begin{array}{rcl} \sum_{i=1}^{n} A_i & = & \sum_{i=1}^{n} (T\_f_i\ s_{i-1} + \Phi\ s_i - \Phi\ s_{i-1}) \\ & = & (\sum_{i=1}^{n} T\_f_i\ s_{i-1}) + \Phi\ s_n - \Phi\ init \\ & \geq & \sum_{i=1}^{n} T\_f_i\ s_{i-1} \end{array}$$

Verification of amortized cost

For each operation f:
provide an upper bound for its amortized cost

$$A\_f :: \tau \Rightarrow num$$

and prove

$$T\_f\ s + \Phi(f\ s) - \Phi\ s \leq A\_f\ s$$

Back to example: counter

$incr :: bool\ list \Rightarrow bool\ list$
$incr\ [] = [True]$
$incr\ (False\ \#\ bs) = True\ \#\ bs$
$incr\ (True\ \#\ bs) = False\ \#\ incr\ bs$

$init = []$

$\Phi\ bs = length\ (filter\ id\ bs)$

Lemma

$T\_incr\ bs + \Phi\ (incr\ bs) - \Phi\ bs = 2$

Proof by induction

Proof obligation summary

- $\Phi\ s \geq 0$
- $\Phi\ init = 0$
- For every operation $f :: \tau \Rightarrow \dots \Rightarrow \tau$:
  $T\_f\ s\ \overline{x} + \Phi(f\ s\ \overline{x}) - \Phi\ s \leq A\_f\ s\ \overline{x}$

If the data structure has an invariant $invar$:
assume precondition $invar\ s$

If f takes 2 arguments of type τ:
$T\_f\ s_1\ \ s_2\ \overline{x} + \Phi(f\ s_1\ \ s_2\ \overline{x}) - \Phi\ s_1 - \Phi\ s_2 \leq A\_f\ s_1\ \ s_2\ \overline{x}$

Warning: real time

Amortized analysis unsuitable for real time applications:

Real running time for individual calls
may be much worse than amortized time

Warning: single threaded

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

Single threaded = no value is used more than once

Otherwise:

```
let  $counter = 0;$
     $bad =$ increment $counter\ 2^n - 1$ times;
     $\_ = incr\ bad;$
     $\_ = incr\ bad;$
     $\_ = incr\ bad;$
     $\vdots$
```

Warning: observer functions

Observer function: does not modify data structure

$\Longrightarrow$ Potential difference $= 0$

$\Longrightarrow$ amortized cost $=$ real cost

$\Longrightarrow$ Must analyze WCC of observer functions

This makes sense because

Observer functions do not consume their arguments!

Legal: *let* bad $=$ create unbalanced data structure with high potential;

$\_ = observer\ bad$;

$\_ = observer\ bad$;

$\vdots$

Archive of Formal Proofs

https://www.isa-afp.org/entries/Amortized_Complexity.shtml

Fact

Can reverse $[x_1,\ldots,x_n]$ onto ys in n steps:

$$
\begin{array}{ll}
 & ([x_1,\ x_2,\ x_3,\ \ldots,\ x_n],\ ys) \\
\to & ([x_2,\ x_3,\ \ldots,\ x_n],\ x_1\ \#\ ys) \\
\to & ([x_3,\ \ldots,\ x_n],\ x_2\ \#\ x_1\ \#\ ys) \\
\vdots & \\
\to & ([],\ x_n\ \#\ \ldots\ \#\ x_1\ \#\ ys)
\end{array}
$$

The problem with $(front,\ rear)$ queues

- Only amortized linear complexity of enq and deq
- Problem: $([],\ rear)$ requires reversal of $rear$

Solution

- Do not wait for $([], \mathit{rear})$
- Compute new front $\mathit{front} \mathbin{@} \mathit{rev}\ \mathit{rear}$
  early and slowly
- In parallel with enq and deq calls
- Using a ‘copy’ of front and rear
  “shadow queue”

Solution

When to start? When $|front| = n$ and $|rear| = n{+}1$

Two phases:

$$\begin{tikzcd}
front \arrow[r, "n"] & rev\ front \arrow[dr, "n"] & \\
& & front\ @\ rev\ rear \\
rear \arrow[r, "n+1"] & rev\ rear \arrow[ur] &
\end{tikzcd}$$

Must finish before original $front$ is empty.
$\Rightarrow$ Must take two steps in every enq and deq call

Complication

Calls of deq remove elements from the original $front$

Cannot easily remove them from the modified copy of $front$

Solution:

- Remember how many elements have been removed
- Better: how many elements are still valid

Example

$$
\begin{array}{lll}
\mathit{enq}: & ([1..5],\ [11..6], & \mathit{Idle}) \\
\rightarrow & ([1..5],\ [], & R\ (0,\ [1..5],[],\ \ [11..6],[]) \\
\rightarrow^2 & & R\ (2,\ [3..5],[2..1],\ [9..6],[10..11]) \\
\mathit{deq}: & ([2..5],\ [], & R\ (1,\ [3..5],[2..1],\ [9..6],[10..11]) \\
\rightarrow^2 & & R\ (3,\ [5],[4..1],\ \ [7..6],[8..11]) \\
\mathit{enq}: & ([2..5],\ [12], & R\ (3,\ [5],[4..1],\ \ [7..6],[8..11]) \\
\rightarrow & & R\ (4,\ [],[5..1],\ \ [6],[7..11]) \\
\rightarrow & & A\ (4,\ [5..1],\ [6..11]) \\
\mathit{deq}: & ([3..5],\ [12], & A\ (3,\ [5..1],\ [6..11]) \\
\rightarrow^2 & & A\ (1,\ [3..1],\ [4..11]) \\
\mathit{deq}: & ([4..5],\ [12], & A\ (0,\ [3..1],\ [4..11]) \\
\rightarrow & & \mathit{Done}\ [4..11]) \\
\rightarrow & ([4..11],\ [12], & \mathit{Idle})
\end{array}
$$

The shadow queue

datatype $'a\ status =$
$\quad Idle\ |$
$\quad Rev\ (nat)\ ('a\ list)\ ('a\ list)\ ('a\ list)\ ('a\ list)\ |$
$\quad App\ (nat)\ ('a\ list)\ ('a\ list)\ |$
$\quad Done\ ('a\ list)$

Shadow step

$exec :: {}'a\ status \Rightarrow {}'a\ status$

$exec\ Idle = Idle$

$exec\ (Rev\ ok\ (x \# f)\ f'\ (y \# r)\ r')$
$= Rev\ (ok + 1)\ f\ (x \# f')\ r\ (y \# r')$

$exec\ (Rev\ ok\ []\ f'\ [y]\ r') = App\ ok\ f'\ (y \# r')$

$exec\ (App\ (ok + 1)\ (x \# f')\ r') = App\ ok\ f'\ (x \# r')$

$exec\ (App\ 0\ f'\ r') = Done\ r'$

$exec\ (Done\ v) = Done\ v$

Dequeue from shadow queue

$$invalidate :: {}'a\ status \Rightarrow {}'a\ status$$

$$invalidate\ Idle = Idle$$
$$invalidate\ (Rev\ ok\ f\ f'\ r\ r') = Rev\ (ok - 1)\ f\ f'\ r\ r'$$
$$invalidate\ (App\ (ok + 1)\ f'\ r') = App\ ok\ f'\ r'$$
$$invalidate\ (App\ 0\ f'\ (x \# r')) = Done\ r'$$
$$invalidate\ (Done\ v) = Done\ v$$

The whole queue

$$\textbf{record}\ 'a\ queue = \begin{array}{ll} front & :: 'a\ list \\ lenf & :: nat \\ rear & :: 'a\ list \\ lenr & :: nat \\ status & :: 'a\ status \end{array}$$

enq and deq

$$enq\ x\ q =$$
$$check\ (q\llparenthesis rear := x \# rear\ q,\ lenr := lenr\ q + 1\rrparenthesis)$$

$$deq\ q =$$
$$check$$
$$(q\llparenthesis lenf := lenf\ q - 1,\ front := tl\ (front\ q),$$
$$status := invalidate\ (status\ q)\rrparenthesis)$$

```
$check\ q =$
$($if $lenr\ q \le lenf\ q$ then $exec2\ q$
 else let $newstate =$
            $Rev\ 0\ (front\ q)\ []\ (rear\ q)\ []$
     in $exec2$
          $(q(\!| lenf := lenf\ q + lenr\ q,$
               $status := newstate,$
               $rear := [],\ lenr := 0 |\!)))$

$exec2\ q =\ (case\ exec\ (exec\ q)\ of$
               $Done\ fr \Rightarrow q(\!| status = Idle,\ front = fr |\!)\ |$
               $newstatus \Rightarrow q(\!| status = newstatus |\!))$
```

Correctness

The proof is

- easy because all functions are non-recursive
  ($\Longrightarrow$ constant running time!)
- tricky because of invariant

$status$ invariant

$$inv\_st\ (Rev\ ok\ f\ f'\ r\ r') =$$
$$(|f| + 1 = |r| \wedge |f'| = |r'| \wedge ok \leq |f'|)$$
$$inv\_st\ (App\ ok\ f'\ r') = (ok \leq |f'| \wedge |f'| < |r'|)$$
$$inv\_st\ Idle = True$$
$$inv\_st\ (Done\ \_) = True$$

Queue invariant

$$
\begin{array}{l}
\mathit{invar}\ q = \\
(\mathit{lenf}\ q = |\mathit{front\_list}\ q| \wedge \\
\ \mathit{lenr}\ q = |\mathit{rev}\ (\mathit{rear}\ q)| \wedge \\
\ \mathit{lenr}\ q \le \mathit{lenf}\ q \wedge \\
\ (\textsf{case}\ \mathit{status}\ q\ \textsf{of} \\
\qquad \mathit{Rev}\ \mathit{ok}\ f\ f'\ r\ r' \Rightarrow \\
\qquad\quad 2 * \mathit{lenr}\ q \le |f'| \wedge \\
\qquad\quad \mathit{ok} \ne 0 \wedge 2 * |f| + \mathit{ok} + 2 \le 2 * |\mathit{front}\ q| \\
\quad |\ \mathit{App}\ \mathit{ok}\ f\ r \Rightarrow \\
\qquad\quad 2 * \mathit{lenr}\ q \le |r| \wedge \mathit{ok} + 1 \le 2 * |\mathit{front}\ q| \\
\quad |\ \_ \Rightarrow \mathit{True}) \wedge \\
\ (\exists\, \mathit{rest}.\ \mathit{front\_list}\ q = \mathit{front}\ q\ @\ \mathit{rest}) \wedge \\
\ (\nexists\, \mathit{fr}.\ \mathit{status}\ q = \mathit{Done}\ \mathit{fr}) \wedge \mathit{inv\_st}\ (\mathit{status}\ q))
\end{array}
$$

Queue invariant

$$
\begin{array}{l}
\textit{front\_list}\ q = \\
(\textsf{case}\ \textit{status}\ q\ \textsf{of} \\
\quad \textit{Idle} \Rightarrow \textit{front}\ q \\
\ \mid \textit{Rev}\ \textit{ok}\ f\ f'\ r\ r' \Rightarrow \textit{rev}\ (\textit{take}\ \textit{ok}\ f')\ @\ f\ @\ \textit{rev}\ r\ @\ r' \\
\ \mid \textit{App}\ \textit{ok}\ f'\ x \Rightarrow \textit{rev}\ (\textit{take}\ \textit{ok}\ f')\ @\ x \\
\ \mid \textit{Done}\ f \Rightarrow f)
\end{array}
$$

The inventors

Robert Hood and Robert Melville.
Real-Time Queue Operation in Pure LISP.
Information Processing Letters, 1981.

Archive of Formal Proofs

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

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

Functions $insert$, $merge$ and del_min
have amortized logarithmic complexity.

Functions $insert$ and del_min are defined via $merge$

Implementation type

Ordinary binary trees

Invariant: $heap$

$merge$

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

Swap subtrees when descending:

$merge\ (\langle l_1,\ a_1,\ r_1\rangle =: t_1)\ (\langle l_2,\ a_2,\ r_2\rangle =: t_2) =$
$(\textsf{if}\ a_1 \le a_2\ \textsf{then}\ \langle merge\ t_2\ r_1,\ a_1,\ l_1\rangle$
$\ \textsf{else}\ \langle merge\ t_1\ r_2,\ a_2,\ l_2\rangle)$

Function $merge$ terminates because ...?

merge

Very similar to leftist heap but

- subtrees are *always* swapped
- no size information needed

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

Right heavy:
$rh\ l\ r = (\text{if } |l| < |r| \text{ then } 1 \text{ else } 0)$

Number of right heavy nodes on left spine:

$lrh\ \langle\rangle = 0$
$lrh\ \langle l,\ \_,\ r\rangle = rh\ l\ r + lrh\ l$

Lemma

$2^{lrh\ t} \leq |t| + 1$

Corollary

$lrh\ t \leq \log_2\ |t|_1$

Towards the proof

Right heavy:
$rh\ l\ r = (\textsf{if}\ |l| < |r|\ \textsf{then}\ 1\ \textsf{else}\ 0)$

Number of not right heavy nodes on right spine:

$rlh\ \langle\rangle = 0$
$rlh\ \langle l,\ \_,\ r\rangle = 1 - rh\ l\ r + rlh\ r$

Lemma

$2^{rlh\ t} \leq |t| + 1$

Corollary

$rlh\ t \leq \log_2\ |t|_1$

Potential

The potential is the number of right heavy nodes:

$$\Phi\ \langle\rangle = 0$$
$$\Phi\ \langle l,\ \_,\ r\rangle = \Phi\ l + \Phi\ r + rh\ l\ r$$

Lemma

$$T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$$
$$\le lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1$$

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

Node-Node case

Let $t_1 = \langle l_1,\ a_1,\ r_1\rangle$, $t_2 = \langle l_2,\ a_2,\ r_2\rangle$.
Case $a_1 \le a_2$. Let $m = merge\ t_2\ r_1$

$T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$
$= T\_merge\ t_2\ r_1 + 1 + \Phi\ m + \Phi\ l_1 + rh\ m\ l_1$
$\quad - \Phi\ t_1 - \Phi\ t_2$
$= T\_merge\ t_2\ r_1 + 1 + \Phi\ m + rh\ m\ l_1$
$\quad - \Phi\ r_1 - rh\ l_1\ r_1 - \Phi\ t_2$
$\le lrh\ m + rlh\ t_2 + rlh\ r_1 + rh\ m\ l_1 + 2 - rh\ l_1\ r_1$
by IH
$= lrh\ m + rlh\ t_2 + rlh\ t_1 + rh\ m\ l_1 + 1$
$= lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1$

Main proof

$$
\begin{aligned}
& T\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\
\le\ & lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1 \\
\le\ & \log_2 |merge\ t_1\ t_2|_1 + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1 \\
=\ & \log_2 (|t_1|_1 + |t_2|_1 - 1) + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1 \\
\le\ & \log_2 (|t_1|_1 + |t_2|_1) + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1 \\
\le\ & \log_2 (|t_1|_1 + |t_2|_1) + 2 * \log_2 (|t_1|_1 + |t_2|_1) + 1
\end{aligned}
$$

because $\log_2 x + \log_2 y \le 2 * \log_2 (x + y)$ if $x,y > 0$

$$= 3 * \log_2 (|t_1|_1 + |t_2|_1) + 1$$

$insert$ and del_min

Easy consequences:

Lemma
$T\_insert\ a\ t + \Phi\ (insert\ a\ t) - \Phi\ t$
$\leq 3 * \log_2 (|t|_1 + 2) + 2$

Lemma
$T\_del\_min\ t + \Phi\ (del\_min\ t) - \Phi\ t$
$\leq 3 * \log_2 (|t|_1 + 2) + 2$

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.

Archive of Formal Proofs

https://www.isa-afp.org/entries/Splay_Tree.shtml

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

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

Definition (splay)

Become wider or more separated.

Example

The river splayed out into a delta.

23 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 $isin/insert/delete\ a$:

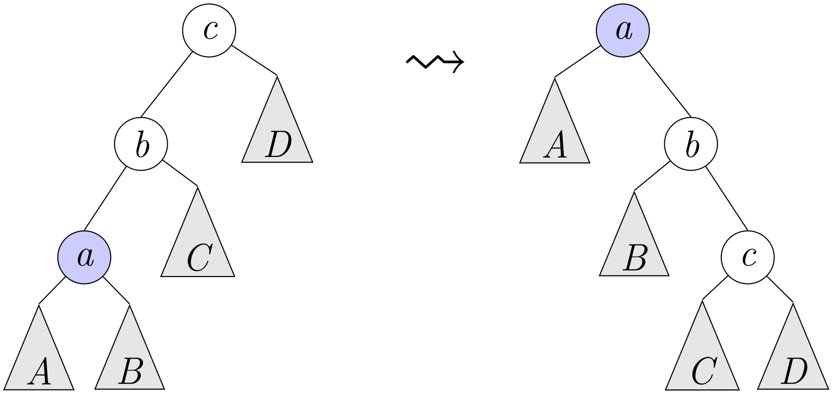
1. $splay\ a$
2. Perform $isin/insert/delete$ action

Key ideas

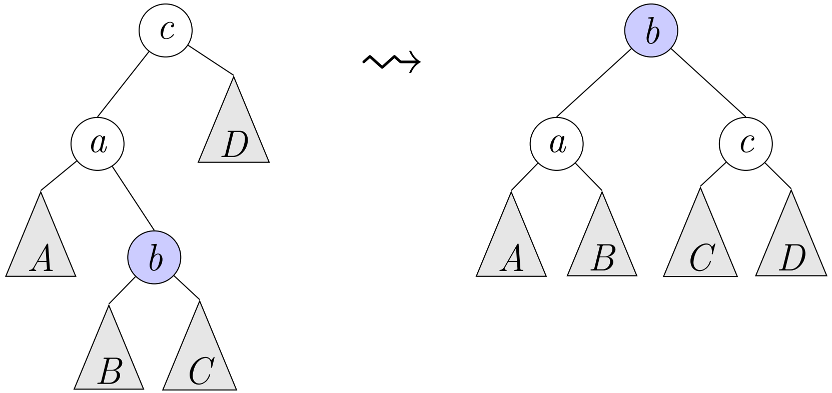
Move to root

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

$$\llbracket x < b;\ x < c;\ AB \neq \langle\rangle \rrbracket$$
$$\Longrightarrow \mathit{splay}\ x\ \langle\langle AB,\ b,\ C\rangle,\ c,\ D\rangle =$$
$$(\textsf{case}\ \mathit{splay}\ x\ AB\ \textsf{of}$$
$$\langle A,\ a,\ B\rangle \Rightarrow \langle A,\ a,\ \langle B,\ b,\ \langle C,\ c,\ D\rangle\rangle\rangle)$$

$$\llbracket x < c;\ c < a;\ BC \neq \langle\rangle \rrbracket$$
$$\Longrightarrow \mathit{splay}\ c\ \langle\langle A,\ x,\ BC\rangle,\ a,\ D\rangle =$$
$$(\textsf{case}\ \mathit{splay}\ c\ BC\ \textsf{of}$$
$$\langle B,\ b,\ C\rangle \Rightarrow \langle\langle A,\ x,\ B\rangle,\ b,\ \langle C,\ a,\ D\rangle\rangle)$$

Some base cases

$$x < b \Longrightarrow splay\ x\ \langle\langle A,\ x,\ B\rangle,\ b,\ C\rangle = \langle A,\ x,\ \langle B,\ b,\ C\rangle\rangle$$

$$x < a \Longrightarrow$$
$$splay\ x\ \langle\langle\langle\rangle,\ a,\ A\rangle,\ b,\ B\rangle = \langle\langle\rangle,\ a,\ \langle A,\ b,\ B\rangle\rangle$$

Functional correctness proofs

Automatic

23 Splay Tree

Algorithm

Amortized Analysis

Archive of Formal Proofs

https://www.isa-afp.org/entries/Amortized_
Complexity.shtml

Potential

Sum of logarithms of the size of all nodes:

$\Phi\ \langle\rangle = 0$

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

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

$[\![bst\ t;\ \langle l,\ a,\ r\rangle \in\ subtrees\ t]\!]$
$\Longrightarrow A\_splay\ a\ t \le 3 * (\varphi\ t - \varphi\ \langle l,\ a,\ r\rangle) + 1$

Corollary

$[\![bst\ t;\ x \in\ set\_tree\ t]\!]$
$\Longrightarrow A\_splay\ x\ t \le 3 * (\varphi\ t - 1) + 1$

Corollary

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

Lemma

$[\![t \neq \langle\rangle;\ bst\ t]\!]$
$\Longrightarrow \exists\, x' \in set\_tree\ t.$
$\quad splay\ x'\ t = splay\ x\ t \wedge T\_splay\ x'\ t = T\_splay$

$insert$

Definition

```
$insert\ x\ t =$
(if $t = \langle\rangle$ then $\langle\langle\rangle,\ x,\ \langle\rangle\rangle$
 else case $splay\ x\ t$ of
        $\langle l,\ a,\ r\rangle \Rightarrow$ case $cmp\ x\ a$ of
                        $LT \Rightarrow \langle l,\ x,\ \langle\langle\rangle,\ a,\ r\rangle\rangle$
                      | $EQ \Rightarrow \langle l,\ a,\ r\rangle$
                      | $GT \Rightarrow \langle\langle l,\ a,\ \langle\rangle\rangle,\ x,\ r\rangle)$
```

Counting only the cost of $splay$:

Lemma

$bst\ t \Longrightarrow$
$T\_insert\ x\ t + \Phi\ (insert\ x\ t) - \Phi\ t \le 4 * \varphi\ t + 3$

$delete$

Definition

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

Lemma

$bst\ t \Longrightarrow$

$T\_delete\ a\ t + \Phi\ (delete\ a\ t) - \Phi\ t \leq 6 * \varphi\ t + 3$

Remember

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

Otherwise:

```
let  $counter = 0$;
     $bad =$ increment $counter\ 2^n - 1$ times;
     $\_ = incr\ bad$;
     $\_ = incr\ bad$;
     $\_ = incr\ bad$;
     ⋮
```

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

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

Otherwise:

```
let  bad = build unbalanced splay tree;
     _ = isin bad a;
     _ = isin bad a;
     _ = isin bad a;
     ⋮
```

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

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

Client uses $splay$ before calling is_root:

Definition

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

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

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.

Archive of Formal Proofs

https://www.isa-afp.org/entries/Pairing_Heap.shtml

Implementation type

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

Heap invariant:

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

Also: $Empty$ must only occur at the root

$insert$

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

$merge\ h\ Empty = h$
$merge\ Empty\ h = h$
$merge\ (Hp\ x\ hsx =: hx)\ (Hp\ y\ hsy =: hy) =$
$(\text{if } x < y \text{ then } Hp\ x\ (hy \# hsx) \text{ else } Hp\ y\ (hx \# hsy))$

Like function $link$ for binomial heaps

del_min

$del\_min\ Empty = Empty$
$del\_min\ (Hp\ x\ hs) = pass_2\ (pass_1\ hs)$

$pass_1\ (h_1\ \#\ h_2\ \#\ hs) = merge\ h_1\ h_2\ \#\ pass_1\ hs$
$pass_1\ hs = hs$

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

Fusing $pass_2 \circ pass_1$

$$
\begin{aligned}
&merge\_pairs\ [] = Empty\\
&merge\_pairs\ [h] = h\\
&merge\_pairs\ (h_1\ \#\ h_2\ \#\ hs) =\\
&merge\ (merge\ h_1\ h_2)\ (merge\_pairs\ hs)
\end{aligned}
$$

Lemma

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

Functional correctness proofs

Straightforward

24 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.

Archive of Formal Proofs

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

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.

More trees

Huffman Trees

Finger Trees

B Trees

k-d Trees

Optimal BSTs

Priority Search Trees

Treaps

...

Graph algorithms

Floyd-Warshall

Dijkstra Dijkstra

Maximum Network Flow

Strongly Connected Components

Kruskal Kruskal

Prim Prim

Algorithms

Knuth-Morris-Pratt

Median of Medians

Approximation Algorithms

FFT

Gauss-Jordan

Simplex

QR-Decomposition

Smith Normal Form

Probabilistic Primality Testing

...

Dynamic programming

- Start with recursive function
- Automatic translation to memoized version incl. correctness theorem
- Applications
  - Optimal binary search tree
  - Minimum edit distance
  - Bellman-Ford (SSSP)
  - CYK
  - ...

Infrastructure

Refinement Frameworks by Lammich:

Abstract specification
$\rightsquigarrow$ functional program
$\rightsquigarrow$ imperative program

using a library of collection types

Model Checkers

- SPIN-like LTL Model Checker:
  Esparza, Lammich, Neumann, Nipkow, Schimpf, Smaus 2013
- SAT Certificate Checker:
  Lammich 2017; beats unverified standard tool

Mostly in the Archive of Formal Proofs