

Functional Data Structures

with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2020-7-20

Part II

Functional Data Structures

Chapter 6

Sorting

① Correctness

② Insertion Sort

③ Time

④ Merge Sort

① Correctness

② Insertion Sort

③ Time

④ Merge Sort

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

$sorted [] = True$

$sorted (x \# ys) = ((\forall y \in set\ ys. x \leq y) \wedge sorted\ ys)$

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:

$$\begin{aligned}\{\#\} &:: 'a \text{ multiset} \\ \text{add_mset} &:: 'a \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \\ + &:: 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \\ \text{mset} &:: 'a \text{ list} \Rightarrow 'a \text{ multiset} \\ \text{set_mset} &:: 'a \text{ multiset} \Rightarrow 'a \text{ set}\end{aligned}$$

Import *HOL–Library.Multiset*

① Correctness

② Insertion Sort

③ Time

④ Merge Sort

HOL/Data_Structures/Sorting.thy

Insertion Sort Correctness

① Correctness

② Insertion Sort

③ Time

④ Merge Sort

Principle: Count function calls

For every function $f :: \tau_1 \Rightarrow \dots \Rightarrow \tau_n \Rightarrow \tau$
define a *timing function* $t_f :: \tau_1 \Rightarrow \dots \Rightarrow \tau_n \Rightarrow \text{nat}$:
Translation of defining equations:

$$\frac{e \rightsquigarrow e'}{f\ p_1 \dots p_n = e \rightsquigarrow t_f\ p_1 \dots p_n = e' + 1}$$

Translation of expressions:

$$\frac{s_1 \rightsquigarrow t_1 \quad \dots \quad s_k \rightsquigarrow t_k}{g\ s_1 \dots s_k \rightsquigarrow t_1 + \dots + t_k + t_g\ s_1 \dots s_k}$$

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

Example

$$\mathit{app} [] \ ys = \ ys$$

\rightsquigarrow

$$\mathit{t_app} [] \ ys = 1 + 1$$

$$\mathit{app} (x\#xs) \ ys = x \# \mathit{app} \ xs \ ys$$

\rightsquigarrow

$$\mathit{t_app} (x\#xs) \ ys = 1 + (1 + 1 + \mathit{t_app} \ xs \ ys) + 1 + 1$$

A compact formulation of

$$e \rightsquigarrow t$$

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

If g is

- a variable, a constant, a constructor or
- a predefined function on *bool* or numbers

then $t_g \dots = 1$.

if and case

So far we model a call-by-value semantics

Conditionals and case expressions are evaluated **lazily**.

Translation:

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

Similarly for case

$O(.)$ is enough

\implies Reduce all additive constants to 1

Example

$t_app\ (x\#xs)\ ys = t_app\ xs\ ys + 1$

\implies Count only

- the defined functions via t_f and
- 1 for the function call.

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 $O(\cdot)$) would require a formal model of (at least) the compiler and the hardware.

HOL/Data_Structures/Sorting.thy

Insertion sort complexity

① Correctness

② Insertion Sort

③ Time

④ Merge Sort

④ Merge Sort

Top-Down

Bottom-Up

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

$merge []\ ys = ys$

$merge\ xs\ [] = xs$

$merge\ (x \# xs)\ (y \# ys) =$
 $(\text{if } x \leq y \text{ then } x \# merge\ xs\ (y \# ys)$
 $\text{else } y \# merge\ (x \# xs)\ ys)$

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

$msort\ xs =$

$(\text{let } n = length\ xs$

$\text{in if } n \leq 1 \text{ then } xs$

$\text{else } merge\ (msort\ (take\ (n\ div\ 2)\ xs))$
 $\quad (msort\ (drop\ (n\ div\ 2)\ xs)))$

Number of comparisons

$c_merge :: 'a\ list \Rightarrow 'a\ list \Rightarrow nat$

$c_msort :: 'a\ list \Rightarrow nat$

Lemma

$c_merge\ xs\ ys$

Theorem

$length\ xs = 2^k \implies c_msort\ xs \leq k * 2^k$

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

$length\ xs = 2^k \implies c_msort_bu\ xs \leq k * 2^k$

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

⑤ Binary Trees

⑥ Basic Functions

⑦ Complete and Balanced Trees

⑤ Binary Trees

⑥ Basic Functions

⑦ Complete and Balanced Trees

HOL/Library/Tree.thy

Binary trees

datatype *'a tree* = *Leaf* | *Node ('a tree) 'a ('a tree)*

Abbreviations:

$$\langle \rangle \equiv \textit{Leaf}$$
$$\langle l, a, r \rangle \equiv \textit{Node } l \ a \ r$$

Most of the time: *tree* = *binary tree*

⑤ Binary Trees

⑥ Basic Functions

⑦ Complete and Balanced Trees

Tree traversal

inorder :: 'a tree \Rightarrow 'a list

inorder $\langle \rangle$ = []

inorder $\langle l, x, r \rangle$ = *inorder* $l @ [x] @ \textit{inorder } r$

preorder :: 'a tree \Rightarrow 'a list

preorder $\langle \rangle$ = []

preorder $\langle l, x, r \rangle$ = $x \# \textit{preorder } l @ \textit{preorder } r$

postorder :: 'a tree \Rightarrow 'a list

postorder $\langle \rangle$ = []

postorder $\langle l, x, r \rangle$ = *postorder* $l @ \textit{postorder } r @ [x]$

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: $|\cdot|$ and $|\cdot|_1$ 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: $h(\cdot)$ 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 $2^{mh(t)} \leq |t|_1$

⑤ Binary Trees

⑥ Basic Functions

⑦ Complete and Balanced Trees

Complete tree

$complete :: 'a \text{ tree} \Rightarrow bool$

$complete \langle \rangle = True$

$complete \langle l, _, r \rangle =$

$(complete\ l \wedge complete\ r \wedge h(l) = h(r))$

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

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

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

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

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

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

Balanced tree

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

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

Balanced trees have optimal height:

Lemma If $balanced\ t$ and $|t| \leq |t'|$ then $h(t) \leq h(t')$.

Warning

- The terms *complete* and *balanced* are not defined uniquely in the literature.
- For example, Knuth calls *complete* what we call *balanced*.

Chapter 8

Search Trees

- ⑧ Unbalanced BST
- ⑨ Abstract Data Types
- ⑩ 2-3 Trees
- ⑪ Red-Black Trees
- ⑫ More Search Trees
- ⑬ Union, Intersection, Difference on BSTs
- ⑭ Tries and Patricia Tries

Most of the material focuses on
BSTs = binary search trees

BSTs represent sets

Any tree represents a set:

$$\text{set_tree} :: 'a \text{ tree} \Rightarrow 'a \text{ set}$$
$$\text{set_tree } \langle \rangle = \{\}$$
$$\text{set_tree } \langle l, x, r \rangle = \text{set_tree } l \cup \{x\} \cup \text{set_tree } r$$

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

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

$bst\ \langle \rangle = True$

$bst\ \langle l, a, r \rangle =$

$(bst\ l \wedge$

$bst\ r \wedge$

$(\forall x \in set_tree\ l. x < a) \wedge (\forall x \in set_tree\ r. a < x))$

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 $'s$
- $empty :: 's$
- $insert :: 'a \Rightarrow 's \Rightarrow 's$
- $delete :: 'a \Rightarrow 's \Rightarrow 's$
- $isin :: 's \Rightarrow 'a \Rightarrow bool$

Map interface

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

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

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)

- ⑧ Unbalanced BST
- ⑨ Abstract Data Types
- ⑩ 2-3 Trees
- ⑪ Red-Black Trees
- ⑫ More Search Trees
- ⑬ Union, Intersection, Difference on BSTs
- ⑭ Tries and Patricia Tries

⑧ Unbalanced BST

Implementation

Correctness

Correctness Proof Method Based on Sorted Lists

Implementation type: *'a tree*

empty = Leaf

insert x <> = <<>, x, <>>

insert x <l, a, r> = (case cmp x a of
 LT ⇒ <insert x l, a, r>
 | EQ ⇒ <l, a, r>
 | GT ⇒ <l, a, insert x r>)

$isin \langle \rangle x = False$

$isin \langle l, a, r \rangle x = (\text{case } cmp\ x\ a\ \text{of}$
 $LT \Rightarrow isin\ l\ x$
 $| EQ \Rightarrow True$
 $| GT \Rightarrow isin\ r\ x)$

```

delete x ⟨⟩ = ⟨⟩
delete x ⟨l, a, r⟩ =
(case cmp x a of
  LT ⇒ ⟨delete x l, a, r⟩
| EQ ⇒ if r = ⟨⟩ then l
        else let (a', r') = split_min r in ⟨l, a', r'⟩
| GT ⇒ ⟨l, a, delete x r⟩)

```

```

split_min ⟨l, a, r⟩ =
(if l = ⟨⟩ then (a, r)
 else let (x, l') = split_min l in (x, ⟨l', a, r⟩))

```


⑧ Unbalanced BST

Implementation

Correctness

Correctness Proof Method Based on Sorted Lists

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 = $(x \in \textit{set_tree } t)$

Under the assumption *bst t*

Also: *bst* must be invariant

bst empty

bst t \implies bst (insert x t)

bst t \implies bst (delete x t)

⑧ Unbalanced BST

Implementation

Correctness

Correctness Proof Method Based on Sorted Lists

Key idea

Local definition:

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

No duplicates!

\Rightarrow *bst* t can be expressed as *sorted*(*inorder* t)

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

- ⑧ Unbalanced BST
- ⑨ Abstract Data Types
- ⑩ 2-3 Trees
- ⑪ Red-Black Trees
- ⑫ More Search Trees
- ⑬ Union, Intersection, Difference on BSTs
- ⑭ Tries and Patricia Tries

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)

$empty :: 's$

$insert :: 'a \Rightarrow 's \Rightarrow 's$

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

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 :: $'s \Rightarrow 'a \text{ set}$ (name arbitrary)

invar :: $'s \Rightarrow \text{bool}$ (name arbitrary)

$\text{set } \text{empty} = \{\}$

$\text{invar } s \implies \text{set}(\text{insert } x \ s) = \text{set } s \cup \{x\}$

$\text{invar } s \implies \text{isin } s \ x = (x \in \text{set } s)$

$\text{invar } \text{empty}$

$\text{invar } s \implies \text{invar}(\text{insert } x \ s)$

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* = ($x \in \text{set } s$)

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):

$$\begin{aligned} invar\ t_1 \wedge \dots \wedge invar\ t_n &\implies \\ \alpha(f\ t_1 \dots t_n) &= f_A(\alpha\ t_1) \dots (\alpha\ t_n) \\ (\alpha \text{ is a homomorphism}) \end{aligned}$$

- f must preserve the invariant:

$$invar\ t_1 \wedge \dots \wedge invar\ t_n \implies invar(f\ t_1 \dots t_n)$$

⑨ Abstract Data Types

Defining ADTs

Using ADTs

Implementing ADTs

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: 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 empty* = $\{\}$ $\langle proof \rangle$

next

fix *s* **assume** *bst s*

show *set_tree (insert_tree 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 . . .

- ⑧ Unbalanced BST
- ⑨ Abstract Data Types
- ⑩ 2-3 Trees
- ⑪ Red-Black Trees
- ⑫ More Search Trees
- ⑬ Union, Intersection, Difference on BSTs
- ⑭ Tries and Patricia Tries

HOL/Data_Structures/
Tree23_Set.thy

2-3 Trees

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

Abbreviations:

$$\langle l, a, r \rangle \equiv \text{Node2 } l \ a \ r$$
$$\langle l, a, m, b, r \rangle \equiv \text{Node3 } l \ a \ m \ b \ r$$

isin

```
isin ⟨l, a, m, b, r⟩ x =  
(case cmp x a of  
  LT ⇒ isin l x  
| EQ ⇒ True  
| GT ⇒ case cmp x b of  
    LT ⇒ isin m x  
  | EQ ⇒ True  
  | GT ⇒ isin r x)
```

Assumes the usual ordering invariant

Structural invariant *complete*

All leaves are at the same level:

$$\textit{complete} \langle \rangle = \textit{True}$$

$$\begin{aligned} \textit{complete} \langle l, _, r \rangle = \\ (h(l) = h(r) \wedge \textit{complete} \, l \wedge \textit{complete} \, r) \end{aligned}$$

$$\begin{aligned} \textit{complete} \langle l, _, m, _, r \rangle = \\ (h(l) = h(m) \wedge h(m) = h(r) \wedge \\ \textit{complete} \, l \wedge \textit{complete} \, m \wedge \textit{complete} \, r) \end{aligned}$$

Lemma

$$\textit{complete} \, t \implies 2^{h(t)} \leq |t| + 1$$

Insertion

The idea:

$Leaf \rightsquigarrow Node2$
 $Node2 \rightsquigarrow Node3$
 $Node3 \rightsquigarrow \text{overflow}$, pass 1 element back up

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

$ins\ x\ \langle \rangle = OF\ \langle \rangle\ x\ \langle \rangle$

$ins\ x\ \langle l, a, r \rangle =$

$case\ cmp\ x\ a\ of$

$LT \Rightarrow case\ ins\ x\ l\ of$

$TI\ l' \Rightarrow TI\ \langle l', a, r \rangle$

$| OF\ l_1\ b\ l_2 \Rightarrow TI\ \langle l_1, b, l_2, a, r \rangle$

$| EQ \Rightarrow TI\ \langle l, x, r \rangle$

$| GT \Rightarrow case\ ins\ x\ r\ of$

$TI\ r' \Rightarrow TI\ \langle l, a, r' \rangle$

$| OF\ r_1\ b\ r_2 \Rightarrow TI\ \langle l, a, r_1, b, r_2 \rangle$

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 $t \implies$

complete $(\text{treeI } (\text{ins } a \ t)) \wedge h(\text{ins } a \ t) = h(t)$

where $h :: 'a \ \text{upI} \Rightarrow \text{nat}$

$h(\text{TI } t) = h(t)$

$h(\text{OF } l \ a \ r) = h(l)$

Proof by induction on t . Base and step automatic.

Corollary

complete $t \implies \text{complete } (\text{insert } a \ t)$

Deletion

The idea:

Node3 \rightsquigarrow *Node2*

Node2 \rightsquigarrow **underflow**, height decreases by 1

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) | 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

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

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

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

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

```

del x ⟨l, a, r⟩ =
(case cmp x a of
  LT ⇒ node21 (del x l) a r
  | EQ ⇒ let (a', t) = split_min r in node22 l a' t
  | GT ⇒ node22 l a (del x r))

```

```

node21 (TD t1) a t2 = TD ⟨t1, a, t2⟩
node21 (UF t1) a ⟨t2, b, t3⟩ = UF ⟨t1, a, t2, b, t3⟩
node21 (UF t1) a ⟨t2, b, t3, c, t4⟩ =
TD ⟨⟨t1, a, t2⟩, b, ⟨t3, c, t4⟩⟩

```

Analogous: *node22*

Deletion preserves *complete*

After 13 simple lemmas:

Lemma

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

Corollary

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

Beyond 2-3 trees

```
datatype 'a tree234 =  
  Leaf | Node2 ... | Node3 ... | Node4 ...
```

Like 2-3 trees, but with many more cases

The general case:

B-trees and (a, b) -trees

- 8 Unbalanced BST
- 9 Abstract Data Types
- 10 2-3 Trees
- 11 Red-Black Trees**
- 12 More Search Trees
- 13 Union, Intersection, Difference on BSTs
- 14 Tries and Patricia Tries

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

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

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* | *Black*

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

Abbreviations:

R l a r \equiv *Node l (a, Red) r*

B l a r \equiv *Node l (a, Black) r*

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

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

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

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

$invc\ \langle \rangle = True$

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

$(invc\ l \wedge$

$invc\ r \wedge$

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

Structural invariants

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

$invh\ \langle \rangle = True$

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

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

$bh(\langle \rangle) = 0$

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

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

Logarithmic height

Lemma

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

Insertion

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

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

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

$ins\ x\ \langle \rangle = R\ \langle \rangle\ x\ \langle \rangle$

$ins\ x\ (B\ l\ a\ r) = (\text{case } cmp\ x\ a\ \text{of}$
 $LT \Rightarrow baliL\ (ins\ x\ l)\ a\ r$
 $| EQ \Rightarrow B\ l\ a\ r$
 $| GT \Rightarrow baliR\ l\ a\ (ins\ x\ r))$

$ins\ x\ (R\ l\ a\ r) = (\text{case } cmp\ x\ a\ \text{of}$
 $LT \Rightarrow R\ (ins\ x\ l)\ a\ r$
 $| EQ \Rightarrow R\ l\ a\ r$
 $| GT \Rightarrow R\ l\ a\ (ins\ x\ r))$

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

$$\begin{aligned} balil\ (R\ (R\ t_1\ a_1\ t_2)\ a_2\ t_3)\ a_3\ t_4 \\ &= R\ (B\ t_1\ a_1\ t_2)\ a_2\ (B\ t_3\ a_3\ t_4) \\ balil\ (R\ t_1\ a_1\ (R\ t_2\ a_2\ t_3))\ a_3\ t_4 \\ &= R\ (B\ t_1\ a_1\ t_2)\ a_2\ (B\ t_3\ a_3\ t_4) \end{aligned}$$

- 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:

- Node z is red.
- If $z \neq p$ is the root, then z is black.
- If the tree violates any of the red-black properties, then z is a violation at most one of them, and the violation is of 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 $z.p$ is black when we reference it in lines 2, 3, 7, 8, 13, and 14.

Recall that we now 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 in the tree **RB-INSERT-FIXUP** is called.

- When **RB-INSERT-FIXUP** is called, z is the red node that was added.
- If $z \neq p$ is the root, then $z.p$ started out black and did not change prior to the call of **RB-INSERT-FIXUP**.
- We have already seen that properties 1, 3, and 5 hold when **RB-INSERT-FIXUP** is called.

If the true violation property 2, then the 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 true violation 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

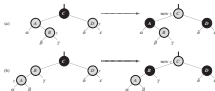


Figure 13.6 Case 1 of the procedure **RB-INSERT-FIXUP**. Property 4 is violated, since z is its parent (z can be both red). We take the case where either part (a) is a right child or (b) is a left child. For (a), the children of z are both red, and each has a black node. For (b), the children of z are both red, and each has a black node. The color of z changes the color of its children, guaranteeing property 3. If z is a right child, the path from z to its left child has the same number of black nodes. The **while** loop continues with node z (z is a grandchild of $z.p$ or its own child). If z is a left child, the path from z to its right child has the same number of black nodes. The **while** loop continues with node z (z is a grandchild of $z.p$ or its own child).

If node z' is the root at the start of the next iteration, then case 1 connected the true 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 true violation of property 4 that existed at the start of this iteration. It then made z' red and left z' alone. If $z' \neq p$ was black, then is no violation of property 4. If $z' \neq p$ was red, coloring z' red created one violation of property 4 (because z' and z' are red).

- 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 consider 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

violation must be because both z and $z.p$ are red. Moreover, the true violation is no other red-black properties.

Termination: When the loop terminates, it does so because z is black. If z is the root, then z is the sentinel T (which is black). Thus, the tree does not violate property 4. If z is not the root, then z is a left child of $z.p$. We have shown the code only for the situation in which z is a left child. The node $z.p$ exists, since by part (b) of the loop invariant, if z is the root, then z is black. Since we enter a loop iteration only if z is red, we know that $z.p$ cannot be the root. Hence, $z.p$ exists.

Maintainance: We actually need to consider six cases in the **while** loop that are then symmetric to the other three, depending on whether line 2 descends to z (parent z is a left child or a right child of z' 's grandparent $z.p$) or z is a right child of $z.p$. We have given the code only for the situation in which z is a left child. The node $z.p$ exists, since by part (b) of the loop invariant, if z is the root, then z is black. Since we enter a loop iteration only if z is red, we know that $z.p$ cannot be the root. Hence, $z.p$ 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 y , $z.p$ is black, and line 4 sets y to z '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 $z.p$ 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 and y are red. Because $z.p$ is black, we can color both z and y black, thereby fixing the problem of z and $z.p$ both being red, and we can color $z.p$ red, thereby maintaining property 5. We then repeat the **while** loop with $z.p$ 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 $z' = z.p$ to denote the node that will be called node z at the start in line 1 upon the next iteration.

- Because this iteration colors z and $z.p$ red, node z' is red at the start of the next iteration.
- The node $z' \neq p$ is $z.p$ 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.
- We have already argued that case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.



Figure 13.6 Case 2 and 3 of the procedure **RB-INSERT-FIXUP**. As in case 1, property 4 is violated in either case (Case 2 because z and its parent $z.p$ are both red. Each of the children of z is a left child of $z.p$ and z is a right child of $z.p$). We transform case 2 into case 3 by a left rotation, which preserves property 4. All descendant node paths from z to z have the same number of black nodes. Case 1 moves some color changes and a right rotation, which also preserve property 4. The **while** loop terminates, because property 4 is violated. 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 1 directly or through case 3, z 's uncle y is black, since otherwise we would have executed case 1. Additionally, the node z 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 in line 11, the identity of z 's p remains unchanged. In case 3, we execute some color changes and a right rotation, which preserves property 5, and thus, since we no longer have two red nodes in a row, we are done. The **while** loop does not leave another node, since z is no more black.

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

- Case 2 makes z point to $z.p$, which is red. No further changes to z or its color occurs in cases 2 and 3.
- Case 3 makes z point to $z.p$, which is red. No further changes to z or its color occurs in cases 2 and 3.
- As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.

Since node z is now 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 true violation of property 4, and they do not introduce another violation.

Deletion

delete x $t = \text{paint } \text{Black} (\text{del } x \ t)$

del $_ \langle \rangle = \langle \rangle$

del $x \langle l, (a, _), r \rangle =$

(*case cmp* x a of

$LT \Rightarrow$

if $l \neq \langle \rangle \wedge \text{color } l = \text{Black}$

then *baldL* $(\text{del } x \ l) \ a \ r$ else $R (\text{del } x \ l) \ a \ r$

| $EQ \Rightarrow \text{app } l \ r$

| $GT \Rightarrow$

if $r \neq \langle \rangle \wedge \text{color } r = \text{Black}$

then *baldR* $l \ a \ (\text{del } x \ r)$ else $R \ l \ a \ (\text{del } x \ r)$)

Deletion

Tricky functions: *baldL*, *baldR*, *app*

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

$$\begin{aligned} & \llbracket \text{invh } t; \text{ invc } t \rrbracket \\ \implies & \text{invh } (\text{del } x \ t) \wedge \\ & (\text{color } t = \text{Red} \wedge \\ & \quad \text{bh}(\text{del } x \ t) = \text{bh}(t) \wedge \text{invc } (\text{del } x \ t) \vee \\ & \quad \text{color } t = \text{Black} \wedge \\ & \quad \text{bh}(\text{del } x \ t) = \text{bh}(t) - 1 \wedge \text{invc2 } (\text{del } x \ t)) \end{aligned}$$

Theorem

$$\text{rbt } t \implies \text{rbt } (\text{delete } x \ t)$$

Code and proof in

114

Source of code

Insertion:

Okasaki's *Purely Functional Data Structures*

Deletion:

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

- ⑧ Unbalanced BST
- ⑨ Abstract Data Types
- ⑩ 2-3 Trees
- ⑪ Red-Black Trees
- ⑫ More Search Trees
- ⑬ Union, Intersection, Difference on BSTs
- ⑭ Tries and Patricia Tries

12 More Search Trees

AVL Trees

Weight-Balanced Trees

AA Trees

Scapegoat 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

AVL Trees

Weight-Balanced Trees

AA Trees

Scapegoat 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*.

12 More Search Trees

AVL Trees

Weight-Balanced Trees

AA Trees

Scapegoat Trees

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

12 More Search Trees

AVL Trees

Weight-Balanced Trees

AA Trees

Scapegoat Trees

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*.

- ⑧ Unbalanced BST
- ⑨ Abstract Data Types
- ⑩ 2-3 Trees
- ⑪ Red-Black Trees
- ⑫ More Search Trees
- ⑬ Union, Intersection, Difference on BSTs
- ⑭ Tries and Patricia Tries

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:

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

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

If $c(x) = \log_2 x \implies$

$$\text{Cost} = O(m * \log_2(n + m)) = O(m * \log_2 n)$$

Similar for intersection and difference.

- We can do better than $O(m * \log_2 n)$
- Flatten trees to lists, merge, build balanced tree takes time $O(m + n)$ — better than $O(m * \log_2 n)$ if $m \approx n$
- This chapter:
 - 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*

13 Union, Intersection, Difference on BSTs

Implementation

Correctness

Join for Red-Black Trees

Uniform *tree* type

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

('a × 'b) tree

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

In this chapter: *tree* abbreviates *('a × 'b) tree*

Just *join*

Can synthesize all BST interface functions from just one function:

$$\textit{join } l \ a \ r \approx \langle l, (a, -), r \rangle + \text{rebalance}$$

Thus *join* determines the balancing schema

Just *join*

Given $\text{join} :: \text{tree} \Rightarrow 'a \Rightarrow \text{tree} \Rightarrow \text{tree}$
we implement

$\text{insert} :: 'a \Rightarrow \text{tree} \Rightarrow \text{tree}$

$\text{delete} :: 'a \Rightarrow \text{tree} \Rightarrow \text{tree}$

$\text{union} :: \text{tree} \Rightarrow \text{tree} \Rightarrow \text{tree}$

$\text{inter} :: \text{tree} \Rightarrow \text{tree} \Rightarrow \text{tree}$

$\text{diff} :: \text{tree} \Rightarrow \text{tree} \Rightarrow \text{tree}$

split :: *tree* \Rightarrow 'a \Rightarrow *tree* \times *bool* \times *tree*

split $\langle \rangle$ *x* = ($\langle \rangle$, *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))

insert :: 'a \Rightarrow *tree* \Rightarrow *tree*

insert x t = (*let* (*l*, -, *r*) = *split t x in join l x r*)

split_min :: *tree* \Rightarrow *'a* \times *tree*

split_min $\langle l, (a, -), r \rangle =$
(if $l = \langle \rangle$ then (a, r) else
let $(m, l') = \textit{split_min } l$ in $(m, \textit{join } l' a r)$)

join2 :: *tree* \Rightarrow *tree* \Rightarrow *tree*

join2 $l r =$
(if $r = \langle \rangle$ then l
else let $(m, r') = \textit{split_min } r$ in $\textit{join } l m r'$)

delete :: *'a* \Rightarrow *tree* \Rightarrow *tree*

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

union :: *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

$\langle l_1, (a, -), r_1 \rangle \Rightarrow$

let $(l_2, -, r_2) = \text{split } t_2 \ a;$

$l' = \text{union } l_1 \ l_2;$

$r' = \text{union } r_1 \ r_2$

in *join* $l' \ a \ r')$

inter :: *tree* \Rightarrow *tree* \Rightarrow *tree*

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

$\langle l_1, (a, -), r_1 \rangle \Rightarrow$

let (*l*₂, *ain*, *r*₂) = *split* *t*₂ *a*;

l' = *inter* *l*₁ *l*₂;

r' = *inter* *r*₁ *r*₂

in if *ain* then *join* *l'* *a* *r'* else *join2* *l'* *r'*)

diff :: *tree* \Rightarrow *tree* \Rightarrow *tree*

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

$\langle l_2, (a, -), r_2 \rangle \Rightarrow$

let (*l*₁, -, *r*₁) = *split* *t*₁ *a*;

l' = *diff* *l*₁ *l*₂;

r' = *diff* *r*₁ *r*₂

in *join2* *l'* *r'*)

Why this way around: *t*₁/*t*₂?

13 Union, Intersection, Difference on BSTs

Implementation

Correctness

Join for Red-Black Trees

Specification of *join* and *inv*

- $set_tree (join\ l\ a\ r) = set_tree\ l \cup \{a\} \cup set_tree\ r$
- $bst\ \langle l, (a, b), r \rangle \implies bst\ (join\ l\ a\ r)$

Also required: structural invariant *inv*:

- $inv\ \langle \rangle$
- $inv\ \langle l, (a, b), r \rangle \implies inv\ l \wedge inv\ r$
- $\llbracket inv\ l; inv\ r \rrbracket \implies inv\ (join\ l\ a\ r)$

Locale context for def of *union* etc

Specification of *union*, *inter*, *diff*

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

- $\text{union}, \text{inter}, \text{diff} :: 's \Rightarrow 's \Rightarrow 's$
- $\llbracket \text{invar } s_1; \text{invar } s_2 \rrbracket \implies \text{set } (\text{union } s_1 \ s_2) = \text{set } s_1 \cup \text{set } s_2$
- $\llbracket \text{invar } s_1; \text{invar } s_2 \rrbracket \implies \text{invar } (\text{union } s_1 \ s_2)$
- $\dots \text{inter} \dots$
- $\dots \text{diff} \dots$

We focus on *union*.

See `HOL/Data_Structures/Set_Specs.thy`

Correctness lemmas for *union* etc code

In the context of *join* specification:

- $bst\ t_2 \implies$
 $set_tree\ (union\ t_1\ t_2) = set_tree\ t_1 \cup set_tree\ t_2$
- $\llbracket bst\ t_1; bst\ t_2 \rrbracket \implies bst\ (union\ t_1\ t_2)$
- $\llbracket inv\ t_1; inv\ t_2 \rrbracket \implies inv\ (union\ t_1\ t_2)$

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

Implementation of locale *Set2*:

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

HOL/Data_Structures/
Set2_Join.thy

13 Union, Intersection, Difference on BSTs

Implementation

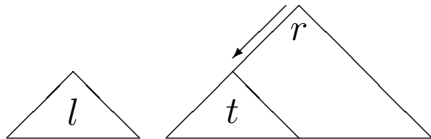
Correctness

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)

Need to store black height in each node
for logarithmic complexity

HOL/Data_Structures/
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.

- ⑧ Unbalanced BST
- ⑨ Abstract Data Types
- ⑩ 2-3 Trees
- ⑪ Red-Black Trees
- ⑫ More Search Trees
- ⑬ Union, Intersection, Difference on BSTs
- ⑭ Tries and Patricia Tries

Trie

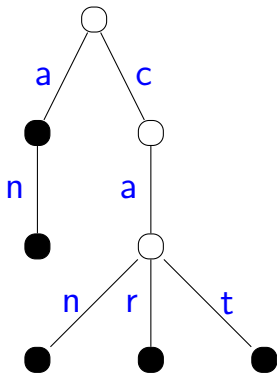
[Fredkin, CACM 1960]

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

Thys/Trie_Fun

Trie

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

Function update notation:

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

$$f(a \mapsto b) = f(a := \text{Some } b)$$

Next: Implementation of ADT *Set*

empty

empty = Nd False (λ_. None)

isin

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

$isin\ (Nd\ b\ m)\ (k\ \# \ xs) = (\text{case } m\ k\ \text{of}$
 $None \Rightarrow False$
 $| \ Some\ t \Rightarrow isin\ t\ xs)$

insert

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

$insert (x \# xs) (Nd\ b\ m) =$

$Nd\ b\ (m(x \mapsto insert\ xs\ (\mathbf{case}\ m\ x\ \mathbf{of}$

$\quad None \Rightarrow empty$

$\quad | \ Some\ t \Rightarrow t)))$

delete

$$\begin{aligned} \text{delete } [] \ (Nd \ b \ m) &= Nd \ False \ m \\ \text{delete } (x \# \ xs) \ (Nd \ b \ m) &= \\ Nd \ b \ (\text{case } m \ x \text{ of} & \\ \quad None \Rightarrow m & \\ \quad | \ Some \ t \Rightarrow m(x \mapsto \text{delete } xs \ t)) & \end{aligned}$$

Does not shrink trie — exercise!

Correctness: Abstraction function

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

$set\ (Nd\ b\ m) =$
 $(\text{if } b \text{ then } \{\} \text{ else } \{\}) \cup$
 $(\bigcup_a \text{ case } m\ a \text{ of}$
 $None \Rightarrow \{\}$
 $| Some\ t \Rightarrow (\#)\ a\ 'set\ t)$

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

14 Tries and Patricia Tries

Tries via Functions

Binary Tries and Patricia Tries

Thys/Tries_Binary

Trie

datatype *trie* = *Lf* | *Nd bool (trie × trie)*

Auxiliary functions on pairs:

sel2 :: *bool* ⇒ *'a* × *'a* ⇒ *'a*

sel2 *b* (*a*₁, *a*₂) = (if *b* then *a*₂ else *a*₁)

mod2 :: (*'a* ⇒ *'a*) ⇒ *bool* ⇒ *'a* × *'a* ⇒ *'a* × *'a*

mod2 *f* *b* (*a*₁, *a*₂) = (if *b* then (*a*₁, *f* *a*₂) else (*f* *a*₁, *a*₂))

empty

$$\textit{empty} = Lf$$

isin

$isin\ Lf\ ks = False$

$isin\ (Nd\ b\ lr)\ ks = (\text{case } ks \text{ of}$
 $\square \Rightarrow b$
 $| k \# x \Rightarrow isin\ (sel2\ k\ lr)\ x)$

insert

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

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

$$\begin{aligned} \text{insert } (k \# ks) \text{ } Lf = \\ Nd \text{ False } (\text{mod2 } (\text{insert } ks) \text{ } k \text{ } (Lf, Lf)) \end{aligned}$$

$$\begin{aligned} \text{insert } (k \# ks) \text{ } (Nd \text{ } b \text{ } lr) = \\ Nd \text{ } b \text{ } (\text{mod2 } (\text{insert } ks) \text{ } k \text{ } lr) \end{aligned}$$

delete

delete ks Lf = Lf

delete ks (Nd b lr) =

case ks of

[] ⇒ node False lr

| k # ks' ⇒ 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:

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

- $isin\ (insert\ xs\ t)\ ys = (xs = ys \vee isin\ t\ ys)$
 $\implies set_trie\ (insert\ xs\ t) = set_trie\ t \cup \{xs\}$
- $isin\ (delete\ xs\ t)\ ys = (xs \neq ys \wedge isin\ t\ ys)$
 $\implies set_trie\ (delete\ xs\ t) = set_trie\ t - \{xs\}$

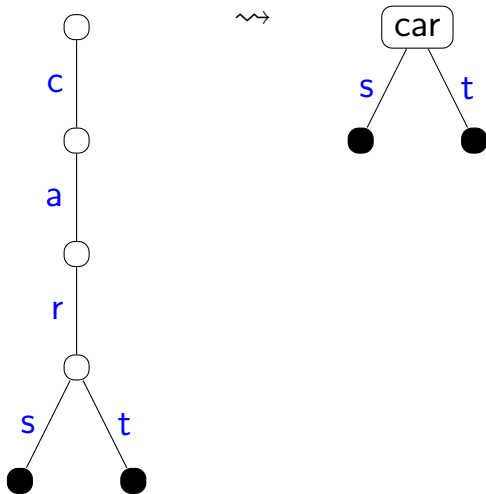
Abstraction function via *isin*

$$\text{set_trie } t = \{xs. \text{isin } t \text{ } xs\}$$

- Trivial definition
- Reusing code (*isin*) may complicate proofs.
- Separate abstract mathematical definition can simplify proofs (see tries with functions)

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

From tries to Patricia tries



Patricia trie

datatype *trieP* = *LfP*
| *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

[] ⇒ b

| k # ks' ⇒ 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*))

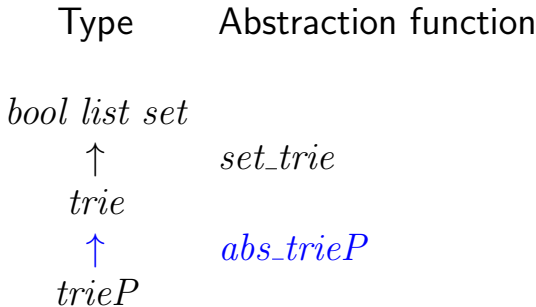
deleteP

$$\text{deleteP } ks \text{ LfP} = \text{LfP}$$

$$\begin{aligned} \text{deleteP } ks \text{ (NdP } ps \text{ b } lr) = \\ (\text{case split } ks \text{ ps of} \\ & (qs, ks', p\#ps') \Rightarrow \text{NdP } ps \text{ b } lr \mid \\ & (qs, k\#ks', []) \Rightarrow \\ & \quad \text{nodeP } ps \text{ b (mod2 (deleteP } ks') \text{ k } lr) \mid \\ & (qs, [], []) \Rightarrow \text{nodeP } ps \text{ False } lr) \end{aligned}$$

Stepwise data refinement

View *trieP* as an implementation (“refinement”) of *trie*



⇒ 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*

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

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

Correctness of *trieP* w.r.t. *bool list set*

Define $set_trieP = set_trie \circ abs_trieP$

\implies 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

$$\begin{aligned} abs_trieP (insertP\ ks\ t) &= insert\ ks\ (abs_trieP\ t) \\ set_trie (insert\ xs\ t) &= set_trie\ t \cup \{xs\} \end{aligned}$$

Chapter 9

Priority Queues

- 15 Priority Queues
- 16 Leftist Heap
- 17 Priority Queue via Braun Tree
- 18 Binomial Heap
- 19 Skew Binomial Heap

15 Priority Queues

16 Leftist Heap

17 Priority Queue via Braun Tree

18 Binomial Heap

19 Skew Binomial Heap

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:

$\text{element} = \text{priority}$

Priority queues are multisets

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



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: $mset :: 'q \Rightarrow 'a \text{ multiset}$

Invariant: $invar :: 'q \Rightarrow bool$

Correctness of implementation

Must prove $\text{invar } q \implies$

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

$$\text{is_empty } q = (\text{mset } q = \{\#\})$$

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

$$\text{mset } q \neq \{\#\} \implies \text{get_min } q = \text{Min_mset } (\text{mset } q)$$

$$\text{mset } q \neq \{\#\} \implies$$

$$\text{mset } (\text{del_min } q) = \text{mset } q - \{\#\text{get_min } q\# \}$$

$$\text{invar empty}$$

$$\text{invar } (\text{insert } x \ q)$$

$$\text{invar } (\text{del_min } q)$$

Terminology

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

$$\text{heap } \langle \rangle = \text{True}$$

$$\text{heap } \langle l, m, r \rangle =$$

$$(\text{heap } l \wedge \text{heap } r \wedge (\forall x \in \text{set_tree } l \cup \text{set_tree } r. m \leq x))$$

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:

$$\begin{aligned} \text{merge } t_1 \ t_2 = & (\text{case } (t_1, t_2) \text{ of} \\ & (\langle \rangle, -) \Rightarrow t_2 \mid \\ & (-, \langle \rangle) \Rightarrow t_1 \mid \\ & (\langle l_1, a_1, r_1 \rangle, \langle l_2, a_2, r_2 \rangle) \Rightarrow \\ & \quad \text{if } a_1 \leq a_2 \text{ then } \langle \text{merge } l_1 \ r_1, a_1, t_2 \rangle \\ & \quad \text{else } \langle t_1, a_2, \text{merge } l_2 \ r_2 \rangle \end{aligned}$$

Challenge: how to maintain some kind of balance

15 Priority Queues

16 Leftist Heap

17 Priority Queue via Braun Tree

18 Binomial Heap

19 Skew Binomial Heap

HOL/Data_Structures/
Leftist_Heap.thy

Leftist tree informally

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

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

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

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

Implementation type

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

Abstraction function:

mset_tree :: *'a lheap* \Rightarrow *'a multiset*

mset_tree $\langle \rangle$ = $\{\#\}$

mset_tree $\langle l, (a, -), r \rangle$ =
 $\{\#a\# \} + \textit{mset_tree } l + \textit{mset_tree } r$

Leftist tree

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

$rank \langle \rangle = 0$

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

Node $\langle l, (a, n), r \rangle$: $n = \text{rank of node}$

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

$ltree \langle \rangle = True$

$ltree \langle l, (-, n), r \rangle =$

$(n = rank\ r + 1 \wedge rank\ r \leq rank\ l \wedge ltree\ l \wedge ltree\ r)$

Leftist heap invariant

$$\textit{invar } h = (\textit{heap } h \wedge \textit{ltree } h)$$

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 $rl = rk$ l ; $rr = rk$ r
in if $rr \leq rl$ then $\langle l, (a, rr + 1), r \rangle$
else $\langle r, (a, rl + 1), l \rangle$)

where $r k \langle -, (-, n), - \rangle = n$

merge

Functional correctness proofs

including preservation of *invar*

Straightforward

Logarithmic complexity

Correlation of rank and size:

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

Complexity measures t_merge , t_insert t_del_min :
count calls of *merge*.

Lemma $t_merge\ l\ r \leq rank\ l + rank\ r + 1$

Corollary $\llbracket ltree\ l; ltree\ r \rrbracket$
 $\implies t_merge\ l\ r \leq \log_2 |l|_1 + \log_2 |r|_1 + 1$

Corollary

$ltree\ t \implies t_insert\ x\ t \leq \log_2 |t|_1 + 2$

Corollary

$ltree\ t \implies t_del_min\ t \leq 2 * \log_2 |t|_1 + 1$

Can we avoid the rank info in each node?

- 15 Priority Queues
- 16 Leftist Heap
- 17 Priority Queue via Braun Tree
- 18 Binomial Heap
- 19 Skew Binomial Heap

Archive of Formal Proofs

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

What is a Braun tree?

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

$braun \langle \rangle = True$

$braun \langle l, x, r \rangle =$

$((|l| = |r| \vee |l| = |r| + 1) \wedge braun\ l \wedge braun\ r)$

1

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

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 $braun\ t'$: $|l| \leq |r'| \wedge |r'| \leq |l| + 1$ □

Delete element: from l , then swap subtrees: $t' = \langle r, x, l' \rangle$

To prove $braun\ t'$: $|l'| \leq |r| \wedge |r| \leq |l'| + 1$ □

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

(if $a < x$ then $\langle \text{insert } x \ r, a, l \rangle$ else $\langle \text{insert } a \ r, x, l \rangle$)

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

(let (y, l') = *del_left* l 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 $\langle l, x, r \rangle =$

(let $(y, l') = \textit{del_left } l$ 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 $\langle \rangle$ =

(if $a \leq x$ then $\langle \langle \langle \rangle$, x, $\langle \rangle$ \rangle , a, $\langle \rangle$ \rangle

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) =

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

else if $x_1 \leq x_2$ then \langle *sift_down* l_1 a r_1 , x_1 , $t_2 \rangle$

else $\langle t_1$, x_2 , *sift_down* l_2 a $r_2 \rangle$

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* $t \implies 2^{h(t)} \leq 2 * |t| + 1$

\implies

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

$pq [] = empty$

$pq (x \# xs) = insert\ x\ (pq\ xs)$

$mins\ q =$

$(if\ is_empty\ q\ then\ [])$

$\quad else\ get_min\ h\ \# mins\ (del_min\ h))$

$sort_pq = mins \circ pq$

Complexity of $sort$: $O(n \log n)$

if all priority queue functions have complexity $O(\log n)$

- 15 Priority Queues
- 16 Leftist Heap
- 17 Priority Queue via Braun Tree
- 18 Binomial Heap**
- 19 Skew Binomial Heap

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 =
Node (rank: nat) (root: 'a) ('a tree list)

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

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

Lemma

$invar_btree\ t \implies |t| = 2^{\text{rank } t}$

Combining two trees

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

$link (Node\ r\ x_1\ ts_1 =: t_1) (Node\ r'\ x_2\ ts_2 =: t_2) =$
(if $x_1 \leq x_2$ then $Node\ (r + 1)\ x_1\ (t_2 \# ts_1)$
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_bheap ts =
(($\forall t \in \text{set } ts. \text{invar_btrees } t$) \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

$ins_tree\ t\ [] = [t]$
 $ins_tree\ t_1\ (t_2 \# ts) =$
 $(if\ rank\ t_1 < rank\ t_2\ then\ t_1 \# t_2 \# ts$
 $\quad else\ ins_tree\ (link\ t_1\ t_2)\ ts)$

merge

```
merge ts1 [] = ts1
merge [] ts2 = ts2
merge (t1 # ts1 =: h1) (t2 # ts2 =: h2) =
  (if rank t1 < rank t2 then t1 # merge ts1 h2
   else if rank t2 < rank t1 then t2 # merge h1 ts2
   else ins_tree (link t1 t2) (merge ts1 ts2))
```

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:

$$ts \neq [] \implies \text{get_min } ts = \text{Min } (\text{set } (\text{map } \text{root } ts))$$

Similar:

$$\text{get_min_rest} :: 'a \text{ tree list} \Rightarrow 'a \text{ tree} \times 'a \text{ tree list}$$

Returns tree with minimal root, and remaining trees

$$\begin{aligned} \text{del_min } ts = \\ (\text{case } \text{get_min_rest } ts \text{ of} \\ \quad (\text{Node } r \ x \ ts_1, \ ts_2) \Rightarrow \text{merge } (\text{rev } ts_1) \ ts_2) \end{aligned}$$

Why *rev*? Rank decreasing in ts_1 but increasing in ts_2

Complexity

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

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

Complexity of operations: linear in length of heap
i.e., logarithmic in number of elements

Proofs: straightforward?

Complexity of *merge*

$\text{merge } (t_1 \# ts_1 =: h_1) (t_2 \# ts_2 =: h_2) =$
(if $\text{rank } t_1 < \text{rank } t_2$ then $t_1 \# \text{merge } ts_1 h_2$
else if $\text{rank } t_2 < \text{rank } t_1$ then $t_2 \# \text{merge } h_1 ts_2$
else $\text{ins_tree } (\text{link } t_1 t_2) (\text{merge } ts_1 ts_2)$)

Complexity of *ins_tree*: $t_ins_tree\ t\ ts \leq \text{length } ts + 1$

A call $\text{merge } t_1\ t_2$ (where $\text{length } ts_1 = \text{length } ts_2 = n$)
can lead to calls of *ins_tree* on lists of length $1, \dots, n$.

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

Complexity of *merge*

$\text{merge } (t_1 \# ts_1 =: h_1) (t_2 \# ts_2 =: h_2) =$
(if $\text{rank } t_1 < \text{rank } t_2$ then $t_1 \# \text{merge } ts_1 h_2$
else if $\text{rank } t_2 < \text{rank } t_1$ then $t_2 \# \text{merge } h_1 ts_2$
else $\text{ins_tree } (\text{link } t_1 t_2) (\text{merge } ts_1 ts_2)$)

Relate time and length of input/output:

$$\begin{aligned} t_{\text{ins_tree } t \text{ } ts} + \text{length } (\text{ins_tree } t \text{ } ts) &= 2 + \text{length } ts \\ \text{length } (\text{merge } ts_1 \text{ } ts_2) + t_{\text{merge } ts_1 \text{ } ts_2} \\ &\leq 2 * (\text{length } ts_1 + \text{length } ts_2) + 1 \end{aligned}$$

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.

- 15 Priority Queues
- 16 Leftist Heap
- 17 Priority Queue via Braun Tree
- 18 Binomial Heap
- 19 Skew Binomial Heap

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

- 20 Amortized Complexity
- 21 Skew Heap
- 22 Splay Tree
- 23 Pairing Heap
- 24 More Verified Data Structures and Algorithms
(in Isabelle/HOL)

20 Amortized Complexity

21 Skew Heap

22 Splay Tree

23 Pairing Heap

24 More Verified Data Structures and Algorithms
(in Isabelle/HOL)

20 Amortized Complexity

Motivation

Formalization

Simple Classical Examples

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 $\Phi :: \text{data-structure} \Rightarrow \text{non-neg. number}$ 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 $0 \rightarrow 1$ flip
 - take out 1 for each $1 \rightarrow 0$ flip

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

Formalization via potential:

$\Phi \text{ counter} = \text{the number of 1's in counter}$

20 Amortized Complexity

Motivation

Formalization

Simple Classical Examples

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, \dots, f_n

Sequence of states:

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

Amortized cost := real cost + potential difference

$$a_{i+1} := t_{f_{i+1}} s_i + \Phi s_{i+1} - \Phi s_i$$

\implies

Sum of amortized costs \geq sum of real costs

$$\begin{aligned} \sum_{i=1}^n a_i &= \sum_{i=1}^n (t_{f_i} s_{i-1} + \Phi s_i - \Phi s_{i-1}) \\ &= \left(\sum_{i=1}^n t_{f_i} s_{i-1} \right) + \Phi s_n - \Phi \text{init} \\ &\geq \sum_{i=1}^n t_{f_i} s_{i-1} \end{aligned}$$

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

$incr :: \text{bool list} \Rightarrow \text{bool list}$

$incr [] = [True]$

$incr (False \# bs) = True \# bs$

$incr (True \# bs) = False \# incr bs$

$init = []$

$\Phi bs = \text{length} (\text{filter id } bs)$

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 $f :: \tau \Rightarrow \dots \Rightarrow \tau$:
$$t_f \ s \ \bar{x} + \Phi(f \ s \ \bar{x}) - \Phi \ s \leq a_f \ s \ \bar{x}$$

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

If *f* takes 2 arguments of type τ :

$$t_f \ s_1 \ s_2 \ \bar{x} + \Phi(f \ s_1 \ s_2 \ \bar{x}) - \Phi \ s_1 - \Phi \ s_2 \leq a_f \ s_1 \ s_2 \ \bar{x}$$

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;  
⋮
```

Warning: observer functions

Observer function: does not modify data structure

$$\Rightarrow \text{Potential difference} = 0$$
$$\Rightarrow \text{amortized cost} = \text{real cost}$$

⇒ 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;  
⋮
```

20 Amortized Complexity

Motivation

Formalization

Simple Classical Examples

Archive of Formal Proofs

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

20 Amortized Complexity

21 Skew Heap

22 Splay Tree

23 Pairing Heap

24 More Verified Data Structures and Algorithms
(in Isabelle/HOL)

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 $\langle \rangle$ $h = h$

merge h $\langle \rangle = h$

Swap subtrees when descending:

merge $(\langle l_1, a_1, r_1 \rangle =: h_1) (\langle l_2, a_2, r_2 \rangle =: h_2) =$
(if $a_1 \leq a_2$ then $\langle \textit{merge } h_2 \ r_1, a_1, l_1 \rangle$
else $\langle \textit{merge } h_1 \ r_2, a_2, l_2 \rangle$)

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

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

Corollary

$$lrh\ h \leq \log_2 |h|_1$$

Towards the proof

Right heavy:

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

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

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

Corollary

$$rlh\ h \leq \log_2 |h|_1$$

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

$$\begin{aligned} & t_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\ & \leq lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1 \end{aligned}$$

`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 $a_1 \leq a_2$. Let $m = \text{merge } t_2 \ r_1$

$$\begin{aligned}
 & t_merge \ t_1 \ t_2 + \Phi \ (\text{merge } t_1 \ t_2) - \Phi \ t_1 - \Phi \ t_2 \\
 &= t_merge \ t_2 \ r_1 + 1 + \Phi \ m + \Phi \ l_1 + rh \ m \ l_1 \\
 &\quad - \Phi \ t_1 - \Phi \ t_2 \\
 &= t_merge \ t_2 \ r_1 + 1 + \Phi \ m + rh \ m \ l_1 \\
 &\quad - \Phi \ r_1 - rh \ l_1 \ r_1 - \Phi \ t_2 \\
 &\leq lrh \ m + rlh \ t_2 + rlh \ r_1 + rh \ m \ l_1 + 2 - rh \ l_1 \ r_1 \\
 &\quad \text{by IH} \\
 &= lrh \ m + rlh \ t_2 + rlh \ t_1 + rh \ m \ l_1 + 1 \\
 &= lrh \ (\text{merge } t_1 \ t_2) + rlh \ t_1 + rlh \ t_2 + 1
 \end{aligned}$$

Main proof

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

insert and del_min

Easy consequences:

Lemma

$$t_{insert} a h + \Phi (insert a h) - \Phi h \\ \leq 3 * \log_2 (|h|_1 + 2) + 2$$

Lemma

$$t_{del_min} h + \Phi (del_min h) - \Phi h \\ \leq 3 * \log_2 (|h|_1 + 2) + 2$$

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.

20 Amortized Complexity

21 Skew Heap

22 Splay Tree

23 Pairing Heap

24 More Verified Data Structures and Algorithms
(in Isabelle/HOL)

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.

22 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 :

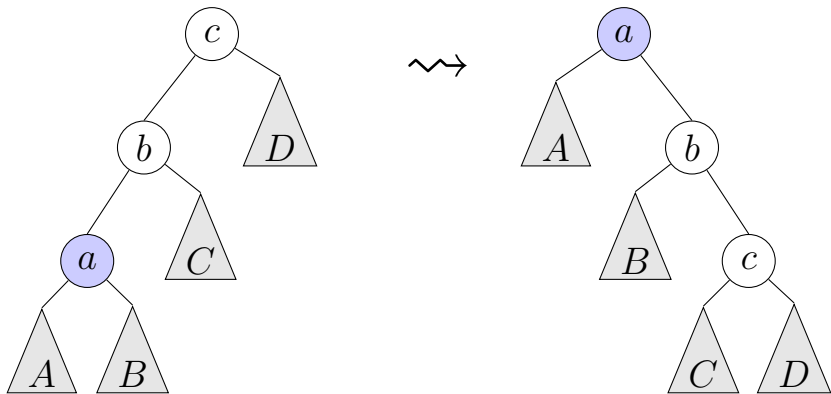
- ① *splay* a
- ② Perform *isin/insert/delete* action

Key ideas

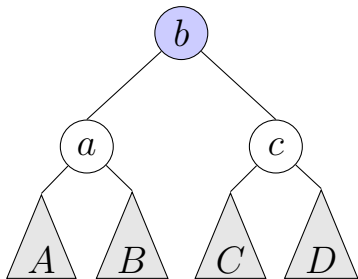
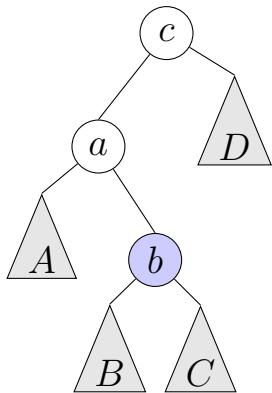
Move to root

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

$$\begin{aligned} & \llbracket x < b; x < c; AB \neq \langle \rangle \rrbracket \\ \implies & \text{splay } x \langle \langle AB, b, C \rangle, c, D \rangle = \\ & (\text{case splay } x AB \text{ of} \\ & \quad \langle A, a, B \rangle \Rightarrow \langle A, a, \langle B, b, \langle C, c, D \rangle \rangle \rangle) \end{aligned}$$

$$\begin{aligned} & \llbracket x < c; c < a; BC \neq \langle \rangle \rrbracket \\ \implies & \text{splay } c \langle \langle A, x, BC \rangle, a, D \rangle = \\ & (\text{case splay } c BC \text{ of} \\ & \quad \langle B, b, C \rangle \Rightarrow \langle \langle A, x, B \rangle, b, \langle C, a, D \rangle \rangle) \end{aligned}$$

Some base cases

$$x < b \implies \text{splay } x \langle \langle A, x, B \rangle, b, C \rangle = \langle A, x, \langle B, b, C \rangle \rangle$$

$$x < a \implies \\ \text{splay } x \langle \langle \langle \rangle, a, A \rangle, b, B \rangle = \langle \langle \rangle, a, \langle A, b, B \rangle \rangle$$

Functional correctness proofs

Automatic

22 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:

$$\Phi \langle \rangle = 0$$

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

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

Amortized complexity of *splay*:

$$a_splay\ a\ t = t_splay\ a\ t + \Phi (splay\ a\ t) - \Phi t$$

Analysis of *splay*

Theorem

$$\begin{aligned} & \llbracket bst\ t; \langle l, a, r \rangle \in subtrees\ t \rrbracket \\ & \implies a_splay\ a\ t \leq 3 * (\varphi\ t - \varphi\ \langle l, a, r \rangle) + 1 \end{aligned}$$

Corollary

$$\begin{aligned} & \llbracket bst\ t; a \in set_tree\ t \rrbracket \\ & \implies a_splay\ a\ t \leq 3 * (\varphi\ t - 1) + 1 \end{aligned}$$

Corollary

$$bst\ t \implies a_splay\ a\ t \leq 3 * \varphi\ t + 1$$

Lemma

$$\begin{aligned} & \llbracket t \neq \langle \rangle; bst\ t \rrbracket \\ & \implies \exists a' \in set_tree\ t. \end{aligned}$$

$$splay\ a'\ t = splay\ a\ t \wedge t_splay\ a'\ t = t_splay\ a\ t_{285}$$

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

$bst\ t \implies$
 $t_splay\ a\ t + \Phi\ (insert\ a\ t) - \Phi\ t \leq 4 * \varphi\ t + 2$

delete

Definition

delete x $t =$
(if $t = \langle \rangle$ then $\langle \rangle$
else case *splay* x t of
 $\langle l, a, r \rangle \Rightarrow$
 if $x \neq a$ 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

$bst\ t \implies$
 $t_delete\ a\ t + \Phi\ (delete\ a\ t) - \Phi\ t \leq 6 * \varphi\ t + 2$

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 \Rightarrow *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

$$\begin{aligned} isin \ t \ a = \\ (\text{let } t' = splay \ a \ t \text{ in } (\text{case } t' \text{ of} \\ \quad \langle \rangle \Rightarrow False \\ \quad | \langle l, x, r \rangle \Rightarrow a = x, \\ \quad t')) \end{aligned}$$

Solution 2:

isin = *splay*; *is_root*

Client uses *splay* before calling *is_root*:

Definition

is_root :: 'a \Rightarrow 'a tree \Rightarrow bool

is_root *x* *t* = (case *t* of
 $\langle \rangle \Rightarrow$ False
 | $\langle l, a, r \rangle \Rightarrow x = a$)

May call *is_root* _ *t* multiple times (with the same *t*!)
because *is_root* takes constant time

\Rightarrow *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.

20 Amortized Complexity

21 Skew Heap

22 Splay Tree

23 Pairing Heap

24 More Verified Data Structures and Algorithms
(in Isabelle/HOL)

Archive of Formal Proofs

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

Implementation type

datatype *'a heap* = *Empty* | *Hp 'a ('a heap list)*

Heap invariant:

pheap Empty = *True*

pheap (Hp x hs) =

$(\forall h \in \text{set } hs. (\forall y \in \#mset_heap\ h. x \leq y) \wedge pheap\ h)$

Also: *Empty* must only occur at the root

insert

insert x $h = \text{merge } (\text{Hp } x \ \square) \ h$

merge $h \ \text{Empty} = h$

merge $\text{Empty } h = h$

merge $(\text{Hp } x \ lx =: hx) \ (\text{Hp } y \ ly =: hy) =$
 $(\text{if } x < y \text{ then } \text{Hp } x \ (hy \ \# \ lx) \text{ else } \text{Hp } y \ (hx \ \# \ ly))$

Like function *link* for binomial heaps

del_min

$$\text{del_min } \text{Empty} = \text{Empty}$$

$$\text{del_min } (\text{Hp } x \text{ } hs) = \text{pass}_2 (\text{pass}_1 \text{ } hs)$$

$$\text{pass}_1 [] = []$$

$$\text{pass}_1 [h] = [h]$$

$$\text{pass}_1 (h_1 \# h_2 \# hs) = \text{merge } h_1 \text{ } h_2 \# \text{pass}_1 \text{ } hs$$

$$\text{pass}_2 [] = \text{Empty}$$

$$\text{pass}_2 (h \# hs) = \text{merge } h (\text{pass}_2 \text{ } hs)$$

Fusing $pass_2 \circ pass_1$

$merge_pairs [] = Empty$

$merge_pairs [h] = h$

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

Lemma

$pass_2 (pass_1 hs) = merge_pairs hs$

Functional correctness proofs

Straightforward

23 Pairing Heap

Amortized Analysis

Analysis

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

$homs :: 'a \text{ heap list} \Rightarrow 'a \text{ tree}$

$homs [] = \langle \rangle$

$homs (Hp\ x\ hs_1\ \# \ hs_2) = \langle homs\ hs_1, x, homs\ hs_2 \rangle$

$hom :: 'a \text{ heap} \Rightarrow 'a \text{ tree}$

$hom\ Empty = \langle \rangle$

$hom (Hp\ x\ hs) = \langle homs\ hs, x, \langle \rangle \rangle$

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.

20 Amortized Complexity

21 Skew Heap

22 Splay Tree

23 Pairing Heap

24 More Verified Data Structures and Algorithms
(in Isabelle/HOL)

More trees

- Huffman Trees:
Huffman 1952 / Blanchette 2008
- Finger Trees:
Hinze and Paterson 2006 /
Nordhoff, Körner and Lammich 2010

Graph algorithms

- Floyd-Warshall:
Floyd 1962, Warshall 1962 /
Wimmer and Lammich 2017
- Shortest Path:
Dijkstra 1956 / Nordhoff and Lammich 2012
- Maximum Flow:
Ford-Fulkerson 1955 / Lammich and Sefidgar 2016
- Strongly Connected Components:
Tarjan 1972 / Schimpf 2015
Gabow 2000 / Lammich 2014
- Minimum spanning tree:
Kruskal 1956, Prim 1957 /
Guttmann 2018, Lammich *et al.* 2019

Model Checkers

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

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

~> functional program

~> imperative program

using a library of collection types

Mostly in the [Archive of Formal Proofs](#)