

# Functional Data Structures

with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik  
Technische Universität München

2021-6-14

# 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

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

$sorted [] = True$

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

# 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:

$\{\#\}$  :: *'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*

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:

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

Translation of expressions:

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

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**.

$$\begin{aligned} & \mathcal{T}[\text{if } b \text{ then } e_1 \text{ else } e_2] \\ &= \mathcal{T}[b] + (\text{if } b \text{ then } \mathcal{T}[e_1] \text{ else } \mathcal{T}[e_2]) \end{aligned}$$

$$\begin{aligned} & \mathcal{T}[\text{case } e \text{ of } p_1 \Rightarrow e_1 \mid \dots \mid p_k \Rightarrow e_k] \\ &= \mathcal{T}[e] + (\text{case } e \text{ of } p_1 \Rightarrow \mathcal{T}[e_1] \mid \dots \mid p_k \Rightarrow \mathcal{T}[e_k]) \end{aligned}$$

Also special: let  $x = t_1$  in  $t_2$

## $O(\cdot)$ is enough

$\implies$  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  $O(\cdot)$ ) 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

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

$merge [] ys = ys$

$merge xs [] = xs$

$merge (x \# xs) (y \# ys) =$   
 $(if\ x \leq y\ then\ x \# merge\ xs\ (y \# ys)$   
 $\quad else\ y \# merge\ (x \# xs)\ ys)$

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

$msort\ xs =$

$(let\ n = length\ xs$

$\quad in\ if\ n \leq 1\ then\ xs$

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

# 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

$m\text{sort\_bu} :: 'a \text{ list} \Rightarrow 'a \text{ list}$

$m\text{sort\_bu } xs = \text{merge\_all } (\text{map } (\lambda x. [x]) xs)$

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

$\text{merge\_all } [] = []$

$\text{merge\_all } [xs] = xs$

$\text{merge\_all } xss = \text{merge\_all } (\text{merge\_adj } xss)$

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

$\text{merge\_adj } [] = []$

$\text{merge\_adj } [xs] = [xs]$

$\text{merge\_adj } (xs \# ys \# zss) =$

$\text{merge } xs \text{ } ys \# \text{merge\_adj } zss$



# 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 (Almost) Complete Trees

5 Binary Trees

6 Basic Functions

7 (Almost) Complete 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 (Almost) Complete Trees

# Tree traversal

*inorder* :: 'a tree  $\Rightarrow$  'a list

*inorder*  $\langle \rangle = []$

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

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

# 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:**  $mh(\cdot)$  only on slides

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

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

5 Binary Trees

6 Basic Functions

7 (Almost) Complete Trees

# Complete tree

*complete* :: 'a tree  $\Rightarrow$  bool

*complete*  $\langle \rangle = \text{True}$

*complete*  $\langle l, -, r \rangle =$

$(h(l) = h(r) \wedge \text{complete } l \wedge \text{complete } r)$

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

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

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

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

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

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



# 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

- 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:

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

$set\_tree\ \langle \rangle = \{\}$

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

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

*bst*  $\langle \rangle$  = True

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

$((\forall x \in \text{set\_tree } l. x < a) \wedge$

$(\forall x \in \text{set\_tree } r. a < x) \wedge \text{bst } l \wedge \text{bst } r)$

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 ⇒ 'b ⇒ 'm ⇒ 'm*
- *delete :: 'a ⇒ 'm ⇒ 'm*
- *lookup :: 'm ⇒ 'a ⇒ '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  $GT$ )

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

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

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

## ⑧ Unbalanced BST

Implementation

**Correctness**

Correctness Proof Method Based on Sorted Lists



# Why is this implementation correct?

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

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

$$\text{set\_tree } (\text{insert } x \ t) = \text{set\_tree } t \cup \{x\}$$

$$\text{set\_tree } (\text{delete } x \ t) = \text{set\_tree } t - \{x\}$$

$$\text{isin } t \ x = (x \in \text{set\_tree } t)$$

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!

$\implies$  *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 See 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

ADT = *interface* + *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*  $invar :: T \Rightarrow bool$

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

## Example (Set ADT)

*empty* :: ...

*insert* :: ...

*isin* :: ...

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

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

*set empty* = {}

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

*invar s*  $\implies$  *isin s x* = ( $x \in$  *set s*)

*invar empty*

*invar s*  $\implies$  *invar(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* = ( $x \in$  *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  $\alpha$  and *invar* (conceptually):  
 $\alpha$  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  $A$ ):

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

- $f$  must preserve the invariant:

$$\textit{invar } t_1 \wedge \dots \wedge \textit{invar } t_n \implies \textit{invar}(f t_1 \dots t_n)$$

## 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 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 ...

- 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 = ⟨  
  | 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*  $\langle l, a, m, b, r \rangle x =$   
(*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:

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

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

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

## Lemma

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

# Insertion

The idea:

*Leaf*  $\rightsquigarrow$  *Node2*  
*Node2*  $\rightsquigarrow$  *Node3*  
*Node3*  $\rightsquigarrow$  **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 ⇒ 'a tree23 ⇒ 'a tree23

*insert* x t = treeI (ins x t)

*ins* :: 'a ⇒ 'a tree23 ⇒ '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*  $t \implies$

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

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

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

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

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

$treeD\ (TD\ t) = t$

$treeD\ (UF\ t) = t$

# Deletion

*delete* :: 'a ⇒ 'a tree23 ⇒ 'a tree23

*delete* x t = treeD (del x t)

*del* :: 'a ⇒ 'a tree23 ⇒ 'a upD

# Deletion

$$\mathit{del} x \langle \rangle = \mathit{TD} \langle \rangle$$

$$\mathit{del} x \langle \langle \rangle, a, \langle \rangle \rangle =$$

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

$$\mathit{del} x \langle \langle \rangle, a, \langle \rangle, b, \langle \rangle \rangle = \dots$$

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

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

Corollary

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



## Beyond 2-3 trees

**datatype** *'a tree*234 =

*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* × *color*) *tree*

Abbreviations:

*R l a r* ≡ *Node l (a, Red) r*

*B l a r* ≡ *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

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

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

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

$invc\ \langle \rangle = True$

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

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



# Structural invariants

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

$invh \langle \rangle = True$

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

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

$bh(\langle \rangle) = 0$

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

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

# Logarithmic height

## Lemma

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

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

# Insertion

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

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

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

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

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

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

## 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  $l/r$

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

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

- 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

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

# 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:

- $N_{\text{node}}$  is red.
- If  $\gamma$  is the root, then  $\gamma$  is black.
- 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  $\gamma$  is red and is not  $N_{\text{node}}$ . If the tree violates property 4, it is because both  $\gamma$  and  $\gamma$ 's  $\text{parent}$  are red.

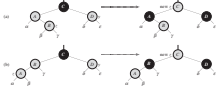
Part (c), which deals with violations of properties, is more central to showing that RB-INSERT-FIXUP restores the red-black properties (see parts (a) and (b), which we see along the way to understanding situations in the code. Because we'll be focusing on node  $c$  and nodes near it in the tree, it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

Recall that we need to show that a loop iteration is *vis* 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 assertions. Thus, at the first iteration, 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  $c$  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 violation, and we added  $n$  red nodes  $\gamma$ . We show that each part of the invariant holds at the time RB-INSERT-FIXUP is called:

- When RB-INSERT-FIXUP is called,  $c$  is the red node that was added.
- If  $\gamma$  is the root, then  $\gamma$ 's  $\text{parent}$  does not exist 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 tree violates property 2, then the red root must be the newly added node  $c$ , which is the only internal node in the tree. Because the parent and both children of  $c$  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  $c$  are black sentinels and the tree had no other violation prior to  $\gamma$  being added to the tree,



**Figure 13.6** Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since  $c$  and its parent ( $a$ ) are both red. We take the entire subtree rooted at  $a$  (a right child of  $b$ ) as a left child of  $b$  and rotate  $b$  around  $a$  to the new black height. The code for line 11 changes the colors of some nodes, preserving property 5. All descendant simple paths in  $a$  and  $b$  had the same number of blacks. The whole tree has the same number of blacks. Any violation of property 4 now occurs only between node  $c$ , which is red, and its parent, if it is not red.

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

If node  $c$ ' 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 tree violation of property 4 that existed at the start of this iteration. It has made  $c$ ' red and left  $c$ ''s  $\text{parent}$  black. If  $c$ ''s  $\text{parent}$  was black, then there is no violation of property 4.

If  $c$ ''s  $\text{parent}$  was red, rotating  $c$ ' created one violation of property 4 between  $c$ ' and  $c$ ''s  $\text{parent}$ .

**Case 2:**  $\gamma$ 's uncle  $y$  is black and  $\gamma$ ' is a right child

**Case 3:**  $\gamma$ 's uncle  $y$  is black and  $\gamma$ ' is a left child

In case 2 and 3, the color of  $\gamma$ 's uncle  $y$  is black. We distinguish the two cases according to whether  $\gamma$  is a right or left child of  $\gamma$ 's parent. Lines 10–11 construct case 2, which is shown in Figure 13.6 together with case 3. In case 2, node  $c$  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  $c$  is a left child. Because

rotation must be because both  $\gamma$  and  $\gamma$ 's  $\text{parent}$  are red. Moreover, the tree violates no other red-black properties.

**Termination:** When the loop terminates, it does so because  $\gamma$  is black. If  $\gamma$  is the root, then  $\gamma$  is the sentinel  $\text{nil}$ , which is black. Thus, the tree does not violate property 4 or 5. If  $\gamma$  is not the root, then  $\gamma$  is a right child of  $\gamma$ 's grandparent  $\gamma$ ''.  $\gamma$  is the only property that might fail to hold in property 2. Line 10 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 distinguishes  $\gamma$ 's parent  $\gamma$ ' as a left child of  $\gamma$ 's grandparent  $\gamma$ '', or  $\gamma$ ' as a right child of  $\gamma$ 's grandparent  $\gamma$ ''. We have given the code only for the situation in which  $\gamma$ ' is a left child. The node  $\gamma$ 's  $\text{parent}$  exists, since by part (b) of the loop invariant, if  $\gamma$ ' is in the root, then  $\gamma$ ' is black. Since we create a loop iteration only if  $\gamma$ ' is red, we know that  $\gamma$ ' cannot be the root. Hence,  $\gamma$ ''s  $\text{parent}$  exists.

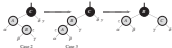
We distinguish case 1 from cases 2 and 3 by the color of  $\gamma$ ''s  $\text{parent}$ 's sibling, or “uncle.” Line 7 makes  $y$  point to  $\gamma$ 's uncle  $y$ ,  $\gamma$ 's right, and then  $\text{temp}$ 's color. If  $\text{temp}$  is red, then we execute case 1. Otherwise, control passes to cases 2 and 3. In all three cases,  $\gamma$ 's grandparent  $\gamma$ ' is black, since its parent  $\gamma$ ' is red, and property 4 is violated only between  $\gamma$ ' and  $\gamma$ '.

**Case 1:**  $\gamma$ 's uncle  $y$  is red

Figure 13.5 shows the situation for case 1 (lines 5–6), which occurs when both  $\gamma$ ' and  $y$  are red. Because  $\gamma$ ' is red, we can color both  $\gamma$ ' and  $y$  black, thereby fixing the problem of  $\gamma$ ' and  $\gamma$ ''s both being red, and we can color  $\gamma$ ''s  $\text{parent}$  red, thereby maintaining property 5. We then repeat the while loop with  $\gamma$ ' as the new node  $c$ '. The pointer  $c$  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  $c$ ' to denote node  $c$  at the current iteration, and  $c$ '' to denote the node that will be called node  $c$ ' at the start of line 1 upon the next iteration.

- Because this iteration colors  $\gamma$ ' and  $y$  black, node  $c$ ' is at the start of the next iteration.
- The node  $c$ ''s  $\text{parent}$  is  $\gamma$ ''s  $\text{parent}$  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 property 1 or 3.



**Figure 13.6** Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated in either case because  $\gamma$ ' and its parent ( $a$ ) are both black. The subtree rooted at  $a$ ,  $b$ , and  $c$  has a black height of 2, and  $\gamma$ ' and  $\gamma$ ''s  $\text{parent}$  are both red. We observe otherwise we would not be in case 1, and we can't see the same black height. We maintain case 2 into case 3 by a left rotation, which preserves property 5. All descendant simple paths from node  $c$  to leaf have the same number of blacks. Case 3 restores some color changes and a right rotation, which also preserves property 5. The while loop terminates because property 4 is satisfied from now on for leaf and red nodes.

both  $\gamma$ ' and  $\gamma$ ''s  $\text{parent}$  are red, the rotation affects neither the black-height of nodes nor property 5. Whether we enter case 3 directly or through case 2,  $\gamma$ 's uncle  $y$  is black, since otherwise we would have executed case 1. Additionally, the node  $\gamma$ 's  $\text{parent}$  exists, since we have argued that this node existed at the time that lines 2 and 3 were executed, and after rotating, its one level in line 10 and then down one level in line 11 (the identity of  $\gamma$ ''s  $\text{parent}$  does not change). In case 3, we execute some color changes and a right rotation, which preserves property 5. We then, since we no longer have two red nodes in a row, we are done. The while loop does not iterate another time, since  $\gamma$ ' is now black.

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

- Case 2 makes  $\gamma$  point to  $\gamma$ 's uncle  $y$ , which is red. No further change to  $\gamma$ ' or its color occurs in cases 2 and 3.
- Case 3 makes  $\gamma$ ' black, so that  $\gamma$ ' is the root at the start of the next iteration. It is black.
- In case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.
- Since node  $c$ ' 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 node that is made red becomes a child of a black node by the rotation in case 3.
- Case 2 and 3 correct the tree violation of property 4, and they do not introduce another violation.

## Deletion code

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

*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*  $\Rightarrow$

if  $r = \langle \rangle$  then  $l$

else let  $(a', r') = \text{split\_min } r$

in if  $\text{color } r = \text{Black}$  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  $(x, l') = \textit{split\_min } l$   
    in  $(x, \text{if } \textit{color } l = \textit{Black} \text{ then } \textit{baldL } l' \ a \ r$   
      else  $\textit{R } l' \ a \ r))$ 
```

```
 $\textit{baldL } (R \ t_1 \ a \ t_2) \ b \ t_3 = R \ (B \ t_1 \ a \ t_2) \ b \ t_3$   
 $\textit{baldL } t_1 \ a \ (B \ t_2 \ b \ t_3) = \textit{baldR } t_1 \ a \ (R \ t_2 \ b \ t_3)$   
 $\textit{baldL } t_1 \ a \ (R \ (B \ t_2 \ b \ t_3) \ c \ t_4) =$   
 $R \ (B \ t_1 \ a \ t_2) \ b \ (\textit{baldR } t_3 \ c \ (\textit{paint } \textit{Red } t_4))$   
 $\textit{baldL } t_1 \ a \ t_2 = R \ t_1 \ a \ t_2$ 
```



# Deletion proof

After a number of lemmas:

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

# Code proof in CDS

## 13.4 Deletion

List the other basic operations on an *node* and *black* tree, deletion of a node takes time  $O(\log n)$ . 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 (Exercise 23.2). First, we need to customize the TRANSPLANT subroutine that TREE-DELETE calls so that it applies to red-black trees:

```
RB-TRANSPLANT( $w, c$ )
1  $w.p \leftarrow T.w$ 
2  $T.w \leftarrow w$ 
3 if  $w.c \neq c$  do  $w.p \leftarrow w$ 
4  $w.p.c \leftarrow c$ 
5 else  $w.p.c \leftarrow \bar{c}$ 
6  $w.p \leftarrow w$ 
```

The procedure RB-TRANSPLANT differs from TRANSPLANT in two ways. First, line 1 reinserts the normal  $T.w$  instead of  $T$ . Second, the assignment to  $w.p$  in line 6 occurs unconditionally: we can assign  $w.p$  to  $w$  even if  $w$  points to the normal.

In fact, we shall exploit the ability to assign to  $w.p$  when  $w \leftarrow T.w$ . The procedure RB-DELETE in line 16 uses 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  $y$ , we know that  $y$  has fewer than two children, then  $y$  is removed from the tree, and we want  $y$  to be  $w$ . When  $y$  has two children, then  $y$  should be  $w$ 's successor, and  $y$  moves into  $w$ '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  $y$ , RB-DELETE calls an auxiliary procedure RB-DELETE-FIXUP, which changes colors and performs rotations to restore the red-black properties.

## 13.4 Deletion

*node*  $x$  is either “doubly black” or “red-black,” and is combined either 2 or 1, respectively, in the color of black nodes (one singly plus combining 2). The color attribute of  $w$  will still be 0 if it is red-black or BLACK if  $w$  is doubly black. In other words, the extra black on a node is reflected in  $w$ 's pointing to the node rather than to the color.

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

```
RB-DELETE-FIXUP( $T$ )
1 while  $w.c \neq \text{BLACK}$  do  $w.c \leftarrow \text{BLACK}$ 
2 if  $w \leftarrow w.p \neq \text{nil}$ 
3  $w.p.c \leftarrow \text{RIGHT}$ 
4  $w.p.c \leftarrow \text{RED}$ 
5  $w \leftarrow w.p.c$ 
6 if  $w.p \neq \text{nil}$ 
7  $w.p.c \leftarrow \text{LEFT}$ 
8  $w.p.c \leftarrow \text{RED}$ 
9 if  $w.p.c \leftarrow \text{BLACK}$  and  $w.p.c \leftarrow \text{right color} \neq \text{BLACK}$ 
10  $w.c \leftarrow \text{nil}$ 
11  $w \leftarrow w.p$ 
12 else  $w.p.c \leftarrow \text{BLACK}$ 
13  $w.c \leftarrow \text{nil}$ 
14  $w \leftarrow w.p$ 
15 return RB-DELETE( $T, w$ )
16  $w \leftarrow w.p$ 
17  $w.c \leftarrow \text{nil}$ 
18  $w \leftarrow w.p$ 
19  $w.c \leftarrow \text{nil}$ 
20  $w \leftarrow w.p$ 
21  $w.c \leftarrow \text{nil}$ 
22  $w \leftarrow w.p$ 
23 else do same as line 16 with “right” and “left” exchanged
```

The procedure RB-DELETE-FIXUP restores properties 1, 2, 4, and Exercise 13.4-1 and 13.4-2 and asks you to show that the procedure restores properties 2 and 4, and to 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  $w$  points to a red-black node, in which case we color  $w$  (singly) black in line 23;
```

```
RB-DELETE( $T, z$ )
1  $y \leftarrow z$ 
2  $\text{void}(\text{color}(y) \leftarrow \text{color}(z))$ 
3  $z \leftarrow z.p$ 
4  $z \leftarrow z.p$ 
5 RB-TRANSPLANT( $T, z, \text{color}(z)$ )
6 if  $z \leftarrow z.p \neq \text{nil}$ 
7  $z \leftarrow z.p$ 
8 RB-TRANSPLANT( $T, z, \text{color}(z)$ )
9 else  $y \leftarrow \text{Tree-Minimum}(T, \text{right}(z))$ 
10  $\text{void}(\text{color}(y) \leftarrow \text{color}(z))$ 
11  $z \leftarrow y.p$ 
12  $y \leftarrow z$ 
13 if  $y.p \neq \text{nil}$ 
14 else RB-TRANSPLANT( $T, y, \text{color}(y)$ )
15  $y.p \leftarrow z$ 
16  $y.p.c \leftarrow \text{right}(z)$ 
17 RB-TRANSPLANT( $T, y$ )
18  $y.p \leftarrow z$ 
19  $y.p.c \leftarrow \text{nil}$ 
20  $y \leftarrow z$ 
21 if original-color  $\neq$  BLACK
22 RB-DELETE-FIXUP( $T$ )
```

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  $\text{nil}$  by  $T.w$  and replacing calls to TRANSPLANT by calls to RB-TRANSPLANT), situated 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  $z$ , which  $z$  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  $\text{original-color}$  stores  $y$ 's color before any changes occur. Lines 11 and 19 set the variable remembering that its assignments to  $w$ . When  $z$  has two children, then  $y$  will move into node  $z$ 's original position in the red-black tree; line 20 gives  $y$  the color color  $\text{nil}$ . We need to save  $y$ 's original color in order to test it at the

## 13.4 Deletion

Within the **while** loop,  $w$  always points to a normal doubly black node. We determine in line 1 whether  $w$  is left child or right child of its parent  $p$ . We have given the code for the situation in which  $w$  is a left child; the situation in which  $w$  is a right child—line 22—is symmetric. We maintain a pointer to the sibling of  $w$ . Since node  $x$  is doubly black, node  $x$  cannot be  $T.w$ , because otherwise, the number of blacks on the simple path from  $p$  to  $x$  (the singly black) leaf  $x$  would exceed the number of blacks on the simple path from  $p$  to  $w$ .

The first case in the code appears in Figure 13.17. 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  $w$ 's extra black) from and including the node on the simple path from  $p$  to  $w$ ,  $p, \dots, w$ . Thus, if property 5 holds prior to the transformation, it continues to hold afterward. For example, in Figure 13.17(a), which illustrates case 1, the number of black nodes from the root to either subtree  $w$  or  $p$ 's left child before and after the transformation (again, remember that node  $x$  adds an extra black). Similarly, the number of black nodes from the root to  $w$ 's right child,  $w$ , is 2, both before and after the transformation. In Figure 13.17(b), the counting must involve the value  $x$  of the color attribute of the root of the subtree shown, which can be either RED or BLACK. If we define consistently  $\text{color}(w)$  to be 1 when the number of black nodes from the root to  $w$  is  $\text{count}(w)$ , both before and after the transformation. In this case, after the transformation, the new node  $x$  has color attribute  $\text{nil}$ , but this node is really either red-black ( $\text{color}(x) = \text{RED}$ ) or doubly black ( $\text{color}(x) = \text{BLACK}$ ). You can verify the other cases similarly (see Exercise 13.4-5).

### Case 1: $w$ 's sibling $x$ is red

Case 1 (lines 5–8 of RB-DELETE-FIXUP and Figure 13.17(a)) occurs when node  $w$ , the sibling of node  $x$ , is red. Since  $w$  must have black children, we can convert the colors of  $w$  and  $p$  and still perform a left rotation on  $w$ , so  $w$  is no longer violating any of the red-black properties. The new sibling of  $w$ , 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.

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

13.4 RB-DELETE-FIXUP: the cases for RB-DELETE-FIXUP are not mutually exclusive.

and 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 node  $y$  that moves into node  $z$ 's original position. The assignment to  $w.p$  in lines 6, 7, and 11 set  $w$  to point to either  $y$ 's left child or  $y$  if  $y$  has no children, the normal  $T.w$ . (Recall from Section 12.3 that  $y$  has no left child.)
- Since node  $x$  moves into node  $y$ 's original position, the attribute  $x.c$  is always set to point to the original position in the tree of  $y$ 's parent, even if  $x$  is, in fact, the normal  $T.w$ . Unlike  $y$ 's color, which is always only when  $w$  has two children and its successor  $x$  is  $w$ 's right child, the assignment to  $x.c$  takes place in line 8 of RB-TRANSPLANT. (Observe that when RB-TRANSPLANT is called in lines 5, 7, or 14, the second parameter passed is the same as  $x.c$ .) When  $y$ 's original parent  $z$  is, however, we do not want  $x.c$  to point to  $z$ 's original parent, since we are removing that node from the tree. Because node  $y$ 's 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  $x \leftarrow T.w$ .

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:

- No black heights in the tree have changed.
- No red nodes have been made adjacent. Because  $y$  takes  $x$ 's place in the tree, along with  $x$ 's color, we cannot have two adjacent red nodes in  $y$ 's new position in the tree. In addition, if  $w$  becomes  $y$ 's right child, then  $w$ '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.
- Since  $x$  could not have been the root of  $w$  if  $w$  was red, the root remains black.

If node  $y$  was black, these problems may arise, which the call of RB-DELETE-FIXUP will remedy. First, if  $y$  had been the root and  $x$  if  $y$  becomes the new root, we have violated property 2. Second, if both  $x$  and  $y$  are red, then we have violated property 4. Third, moving  $x$  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  $w$  is the sibling of the black node  $x$ , then  $w$ 's simple path that contains  $x$ , then under this interpretation, property 5 holds. When we restore or move the black node  $w$ , we “use” its blackness on node  $x$ . The problem is that the new node  $x$  is neither red nor black, thereby violating property 1. Instead,

## 13.4 Deletion

Case 2:  $w$ 's sibling  $x$  is black, and both of  $w$ 's children are black  
In case 2 (lines 10–11 of RB-DELETE-FIXUP and Figure 13.17(b)), both of  $w$ 's children are black. Since  $w$  is also black, we take one extra black off  $w$  and, leaving  $w$  with only one black and leaving  $w$  red. By compensating for removing one black from  $w$  and  $p$ , we would like to add an extra black to  $p$ , which was originally either red or black. We do so by replacing the white leaf with  $x$  as the new node  $w$ . (Observe that the original  $w$  was black in line 10;  $w$  is red-black, since the original  $x$  was red. However, the value of the color attribute of the new node  $w$  is RED, and the loop terminates when it sees the loop condition. We then color the new node  $w$  black in line 11.)

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

Case 3 (lines 12–16 of Figure 13.17) 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  $x$  and still perform a left rotation, since the rotation restores  $w$  to the red-black property. The new sibling of  $w$  is now a black node with a red right child, and thus we have transformed case 3 into case 4.

### Case 4: $w$ and $w$ 's right child are red

Case 4 (lines 17–21 of Figure 13.17(d)) occurs when node  $w$ 's sibling  $x$  is black and  $w$ 's right child is red. By making one color change and performing a left rotation on  $p$ , we can remove the extra black on  $w$ , making it singly black, without violating any of the red-black properties. Note that  $w$  is to be the root causes the **while** loop to terminate when it sees the loop condition.

## Analysis

What is the running time of RB-DELETE? Since the height of a red-black tree of  $n$  nodes is  $O(\log n)$ , the total cost of the procedure without the call to RB-DELETE-FIXUP takes  $O(\log n)$ . Within RB-DELETE-FIXUP, each of cases 1, 3, 4, and 5 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 does not terminate, and then the pointer  $w$  moves on the tree at most  $O(\log n)$  times, performing no rotations. Thus, the procedure RB-DELETE-FIXUP takes  $O(\log n)$  time and performs at most three rotations, and the overall time for RB-DELETE is therefore also  $O(\log n)$ .

# 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.

- 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  $O(m * \log_2 n)$

- 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:

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

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 t1 t2 =
  (if t1 = ⟨⟩ then t2
   else if t2 = ⟨⟩ then t1
   else case t1 of
     ⟨l1, (a, b), r1⟩ ⇒
       let (l2, x, r2) = split t2 a;
           l' = union l1 l2;
           r' = union r1 r2
       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 t1 t2 =
  (if t1 = ⟨ ⟩ then ⟨ ⟩
   else if t2 = ⟨ ⟩ then ⟨ ⟩
   else case t1 of
     ⟨ l1, (a, x), r1 ⟩ ⇒
       let (l2, b, r2) = split t2 a;
           l' = inter l1 l2;
           r' = inter r1 r2
       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 t1 t2 =
  (if t1 = ⟨⟩ then ⟨⟩
   else if t2 = ⟨⟩ then t1
    else case t2 of
      ⟨l2, (a, b), r2⟩ ⇒
        let (l1, x, r1) = split t1 a;
            l' = diff l1 l2;
            r' = diff r1 r2
        in join2 l' r')

```



## *insert and delete*

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

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

## 13 Union, Intersection, Difference on BSTs

Correctness

Join for Red-Black Trees

## Specification of *join* and *inv*

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

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

- $union, inter, diff :: 's \Rightarrow 's \Rightarrow 's$
- $\llbracket invar\ s_1; invar\ s_2 \rrbracket \implies set\ (union\ s_1\ s_2) = set\ s_1 \cup set\ s_2$
- $\llbracket invar\ s_1; invar\ s_2 \rrbracket \implies invar\ (union\ s_1\ s_2)$
- ...*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:

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

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

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.

- 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

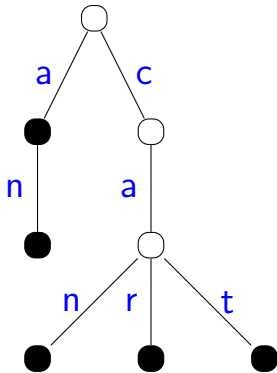
[Fredkin, CACM 1960]

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

HOL/Data\_Structures/  
Trie\_Fun.thy



# Trie

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

Function update notation:

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

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

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*) =

let *s* = case *m x* of

*None* ⇒ *empty*

    | *Some t* ⇒ *t*

in *Nd b* (*m(x ↦ insert xs s)*)

# *delete*

*delete* [] (Nd b m) = Nd False m

*delete* (x # xs) (Nd b m) =

Nd b (case m x of

    None  $\Rightarrow$  m

    | Some t  $\Rightarrow$  m(x  $\mapsto$  *delete* xs t))

**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*

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

- 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:

$\text{lookup} :: 't \Rightarrow ('a \Rightarrow 'b \text{ option})$



## 14 Tries and Patricia Tries

Tries via Functions

Binary Tries and Patricia Tries

HOL/Data\_Structures/  
Tries\_Binary.thy

# Trie

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

Auxiliary functions on pairs:

*sel2* :: *bool* ⇒ 'a × 'a ⇒ 'a

*sel2* *b* (*a*<sub>1</sub>, *a*<sub>2</sub>) = (if *b* then *a*<sub>2</sub> else *a*<sub>1</sub>)

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

*mod2* *f* *b* (*a*<sub>1</sub>, *a*<sub>2</sub>) = (if *b* then (*a*<sub>1</sub>, *f* *a*<sub>2</sub>) else (*f* *a*<sub>1</sub>, *a*<sub>2</sub>))

*empty*

*empty = Lf*

# *isin*

*isin Lf ks = False*

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

## *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*

*[] ⇒ node False lr*

*| k # ks' ⇒ node b (mod2 (delete ks') k lr)*

**Shrink trie if possible:**

*node b lr = (if ¬ b ∧ lr = (Lf, Lf) then Lf else Nd b lr)*

# Correctness of implementation

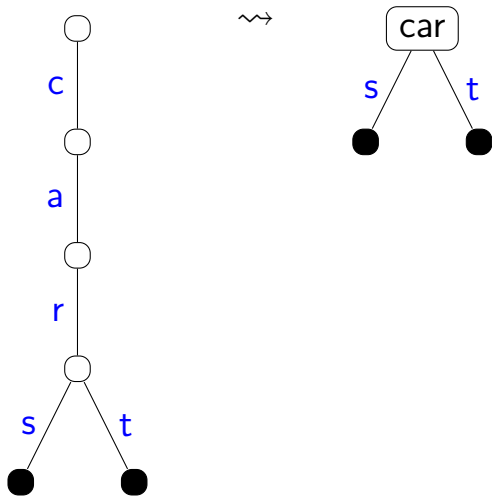
Abstraction function:

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

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



# From tries to Patricia tries



# Patricia trie

**datatype** *trieP* = *LfP*  
| *NdP* (*bool list*) *bool* (*trieP* × *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, [], []) ⇒ NdP ps True lr*

*| (qs, [], p # ps') ⇒*

*let t = NdP ps' b lr*

*in NdP qs True (if p then (LfP, t) else (t, LfP))*

*| (qs, k # ks', []) ⇒ NdP ps b (mod2 (insertP ks') k lr)*

*| (qs, k # ks', p # ps') ⇒*

*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') ⇒ NdP ps b lr |*

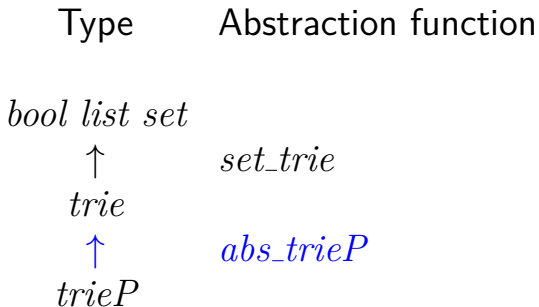
*(qs, k#ks', []) ⇒*

*nodeP ps b (mod2 (deleteP ks') k lr) |*

*(qs, [], []) ⇒ nodeP ps False lr)*

# Stepwise data refinement

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



$\implies$  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*

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

$isin\ (prefix\_trie\ ps\ t)\ ks =$

$(ps = take\ (length\ ps)\ ks \wedge isin\ t\ (drop\ (length\ ps)\ ks))$

$prefix\_trie\ ks\ (Nd\ True\ (Lf,\ Lf)) = insert\ ks\ Lf$

$insert\ ps\ (prefix\_trie\ ps\ (Nd\ b\ lr)) = prefix\_trie\ ps\ (Nd\ True\ lr)$

$insert\ (ks\ @\ ks')\ (prefix\_trie\ ks\ t) = prefix\_trie\ ks\ (insert\ ks'\ t)$

$prefix\_trie\ (ps\ @\ qs)\ t = prefix\_trie\ ps\ (prefix\_trie\ qs\ t)$

$split\ ks\ ps = (qs,\ ks',\ ps') \implies$

$ks = qs\ @\ ks' \wedge ps = qs\ @\ ps' \wedge (ks' \neq [] \wedge ps' \neq [] \implies hd\ ks' \neq hd\ ps')$

$(prefix\_trie\ xs\ t = Lf) = (xs = [] \wedge t = Lf)$

$(abs\_trieP\ t = Lf) = (t = LfP)$

$delete\ xs\ (prefix\_trie\ xs\ (Nd\ b\ (l,\ r))) =$

$(if\ (l,\ r) = (Lf,\ Lf)\ then\ Lf\ else\ prefix\_trie\ xs\ (Nd\ False\ (l,\ r)))$

$delete\ (xs\ @\ ys)\ (prefix\_trie\ xs\ t) =$

$(if\ delete\ ys\ t = Lf\ then\ Lf\ else\ prefix\_trie\ xs\ (delete\ ys\ t))$

## 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:

element = 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  $invar\ q \implies$

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

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

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

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

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

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

$invar\ empty$

$invar\ (insert\ x\ q)$

$invar\ (del\_min\ q)$

# Terminology

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

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

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

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

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

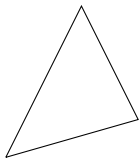
HOL/Data\_Structures/  
Leftist\_Heap.thy



## Leftist tree informally

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

$\implies$  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* × *nat*) *tree*

Abstraction function:

*mset\_tree* :: *'a lheap* ⇒ *'a multiset*

*mset\_tree* ⟨⟩ = {#}

*mset\_tree* ⟨*l*, (*a*, -), *r*⟩ =

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

# Leftist tree

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

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

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

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

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

*where*  $mht$   $\langle -, (-, n), - \rangle = n$

## *merge*

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

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**  $\llbracket ltree\ l; ltree\ r \rrbracket$

$$\implies T\_merge\ l\ r \leq mh(l) + mh(r) + 1$$

**Corollary**  $\llbracket ltree\ l; ltree\ r \rrbracket$

$$\implies T\_merge\ l\ r \leq \log_2 |l|_1 + \log_2 |r|_1 + 1$$

**Corollary**

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

**Corollary**

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



Can we avoid the height 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 ⇒ 'a tree ⇒ 'a tree

*insert* a ⟨⟩ = ⟨⟨⟩, a, ⟨⟩⟩

*insert* a ⟨l, x, r⟩ =

(if  $a < x$  then ⟨*insert* x r, a, l⟩ else ⟨*insert* a r, x, l⟩)

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  $(y, l') = \text{del\_left } l$  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  $\langle \textit{sift\_down } l_1 \textit{ a } r_1, x_1, t_2 \rangle$

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

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  $[]$

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) =*

*(( $\forall t \in \text{set } ts. \text{invar\_btree } t$ )  $\wedge$  map rank ts = 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 ts =*  
*(( $\forall t \in \text{set } ts. \text{invar\_tree } t$ )  $\wedge$*   
*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* ( $t_1 \# ts_1 =: h_1$ ) ( $t_2 \# ts_2 =: h_2$ ) =

(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*  $ts_1$   $ts_2$ ))

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

$$\text{del\_min } ts =$$

$$(\text{case } \text{get\_min\_rest } ts \text{ of}$$

$$(\text{Node } r \ x \ ts_1, \ ts_2) \Rightarrow \text{merge } (\text{rev } ts_1) \ ts_2)$$

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

# Complexity

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

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

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

Proofs: straightforward?



## Complexity of *merge*

*merge* ( $t_1 \# ts_1 =: h_1$ ) ( $t_2 \# ts_2 =: h_2$ ) =  
(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:

$$\begin{aligned} T_{ins\_tree\ t\ ts} + length (ins\_tree\ t\ ts) &= 2 + length\ ts \\ length (merge\ ts_1\ ts_2) + T_{merge\ ts_1\ ts_2} \\ &\leq 2 * (length\ ts_1 + length\ ts_2) + 1 \end{aligned}$$

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 Hood Melville Queues
- 22 Skew Heap
- 23 Splay Tree
- 24 Pairing Heap
- 25 More Verified Data Structures and Algorithms  
(in Isabelle/HOL)

- 20 Amortized Complexity
- 21 Hood Melville Queues
- 22 Skew Heap
- 23 Splay Tree
- 24 Pairing Heap
- 25 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:

$\Phi$  *counter* = the number of 1's in *counter*

## 20 Amortized Complexity

Motivation

**Formalization**

Simple Classical Examples

# Data structure

Given an implementation:

- Type  $\tau$
- 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

$$a_f :: \tau \Rightarrow \text{num}$$

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

- $\Phi s \geq 0$
- $\Phi \text{ init} = 0$
- 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  $\tau$ :

$$t\_f s_1 s_2 \bar{x} + \Phi(f s_1 s_2 \bar{x}) - \Phi s_1 - \Phi s_2 \leq a\_f s_1 s_2 \bar{x}$$

## 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

⇒ Potential difference = 0

⇒ amortized cost = 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 Hood Melville Queues**
- 22 Skew Heap
- 23 Splay Tree
- 24 Pairing Heap
- 25 More Verified Data Structures and Algorithms  
(in Isabelle/HOL)

# Fact

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

$([x_1, x_2, x_3, \dots, x_n], ys)$   
 $\rightarrow ([x_2, x_3, \dots, x_n], x_1 \# ys)$   
 $\rightarrow ([x_3, \dots, x_n], x_2 \# x_1 \# ys)$   
 $\vdots$   
 $\rightarrow ([], x_n \# \dots \# x_1 \# ys)$

## 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 ( $front$ ,  $rear$ )
- Compute new front  $front @ rev rear$   
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:

$front \xrightarrow{n} rev\ front$

$\searrow^n$

$front @ rev\ rear$

$\nearrow$

$rear \xrightarrow{n+1} rev\ rear$

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

*enq*: ([1..5], [11..6], *Idle*)  
→ ([1..5], [], *R* (0, [1..5], [], [11..6], []))  
→<sup>2</sup> *R* (2, [3..5], [2..1], [9..6], [10..11])  
*deq*: ([2..5], [], *R* (1, [3..5], [2..1], [9..6], [10..11]))  
→<sup>2</sup> *R* (3, [5], [4..1], [7..6], [8..11])  
*enq*: ([2..5], [12], *R* (3, [5], [4..1], [7..6], [8..11]))  
→ *R* (4, [], [5..1], [6], [7..11])  
→ *A* (4, [5..1], [6..11])  
*deq*: ([3..5], [12], *A* (3, [5..1], [6..11]))  
→<sup>2</sup> *A* (1, [3..1], [4..11])  
*deq*: ([4..5], [12], *A* (0, [3..1], [4..11]))  
→ *Done* [4..11])  
→ ([4..11], [12], *Idle*)

# The shadow queue

```
datatype 'a status =  
  Idle |  
  Rev ('a list) ('a list) ('a list) ('a list) |  
  App ('a list) ('a list) |  
  Done ('a list)
```

## Shadow step

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

$exec \text{ Idle} = \text{Idle}$

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

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

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

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

$exec (\text{Done } v) = \text{Done } v$

## 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

```
record 'a queue = front  :: 'a list  
                  lenf   :: nat  
                  rear   :: 'a list  
                  lenr   :: nat  
                  status :: 'a status
```

## *enq and deq*

*enq*  $x$   $q =$

*check* ( $q \parallel \text{rear} := x \# \text{rear } q, \text{lenr} := \text{lenr } q + 1 \parallel$ )

*deq*  $q =$

*check*

( $q \parallel \text{lenf} := \text{lenf } q - 1, \text{front} := \text{tl} (\text{front } q),$   
 $\text{status} := \text{invalidate} (\text{status } q) \parallel$ )



```

check q =
(if lenr q ≤ 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 ⇒ q(status = Idle, front = fr) |
  newstatus ⇒ q(status = newstatus))

```

# Correctness

The proof is

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

## *status* invariant

$$\text{inv\_st} (\text{Rev } ok\ f\ f'\ r\ r') =$$

$$(|f| + 1 = |r| \wedge |f'| = |r'| \wedge ok \leq |f'|)$$

$$\text{inv\_st} (\text{App } ok\ f'\ r') = (ok \leq |f'| \wedge |f'| < |r'|)$$

$$\text{inv\_st } \text{Idle} = \text{True}$$

$$\text{inv\_st} (\text{Done } \_) = \text{True}$$

## Queue invariant

$$\begin{aligned} \text{invar } q = & \\ & (\text{lenf } q = |\text{front\_list } q| \wedge \\ & \text{lenr } q = |\text{rev } (\text{rear } q)| \wedge \\ & \text{lenr } q \leq \text{lenf } q \wedge \\ & (\text{case status } q \text{ of} \\ & \quad \text{Rev } ok \ f \ f' \ r \ r' \Rightarrow \\ & \quad \quad 2 * \text{lenr } q \leq |f'| \wedge \\ & \quad \quad ok \neq 0 \wedge 2 * |f| + ok + 2 \leq 2 * |\text{front } q| \\ & \quad | \text{App } ok \ f \ r \Rightarrow \\ & \quad \quad 2 * \text{lenr } q \leq |r| \wedge ok + 1 \leq 2 * |\text{front } q| \\ & \quad | \_ \Rightarrow \text{True}) \wedge \\ & (\exists \text{rest. front\_list } q = \text{front } q \ @ \ \text{rest}) \wedge \\ & (\nexists \text{fr. status } q = \text{Done } \text{fr}) \wedge \text{inv\_st } (\text{status } q)) \end{aligned}$$

# Queue invariant

*front\_list*  $q =$

(*case status*  $q$  of

*Idle*  $\Rightarrow$  *front*  $q$

| *Rev*  $ok\ f\ f'\ r\ r' \Rightarrow rev\ (take\ ok\ f')\ @\ f\ @\ rev\ r\ @\ r'$

| *App*  $ok\ f'\ x \Rightarrow rev\ (take\ ok\ f')\ @\ x$

| *Done*  $f \Rightarrow f$ )

# The inventors

Robert Hood and Robert Melville.

Real-Time Queue Operation in Pure LISP.

*Information Processing Letters*, 1981.

- 20 Amortized Complexity
- 21 Hood Melville Queues
- 22 Skew Heap**
- 23 Splay Tree
- 24 Pairing Heap
- 25 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*  $\langle \rangle t = t$

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

Swap subtrees when descending:

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

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

$$T_{\text{merge}}(t_1, t_2) + \Phi(\text{merge}(t_1, t_2)) - \Phi(t_1) - \Phi(t_2) \leq 3 * \log_2(|t_1|_1 + |t_2|_1) + 1$$

# 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:

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

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

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

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

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

`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

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

## *insert and del\_min*

Easy consequences:

Lemma

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

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.

- 20 Amortized Complexity
- 21 Hood Melville Queues
- 22 Skew Heap
- 23 Splay Tree**
- 24 Pairing Heap
- 25 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.

## 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* :

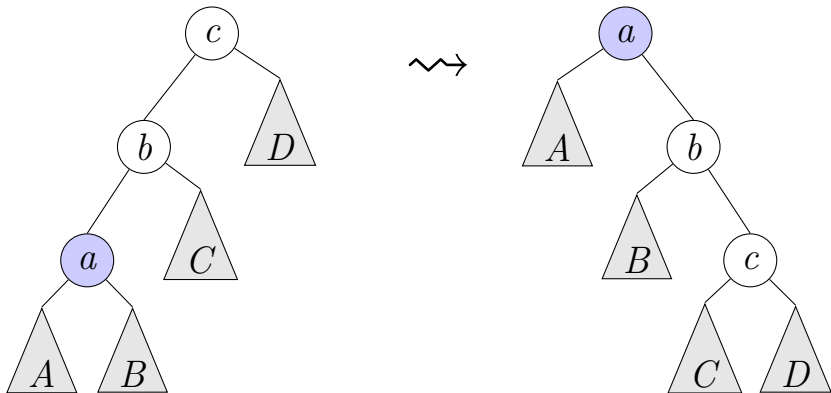
- ① *splay a*
- ② Perform *isin/insert/delete* action

# Key ideas

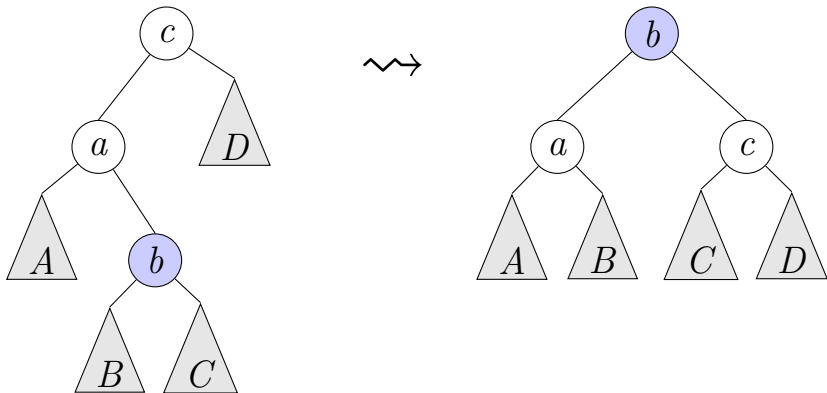
Move to root

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 ⇒ 'a tree ⇒ '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

## 23 Splay Tree

Algorithm

Amortized Analysis

# Archive of Formal Proofs

[https://www.isa-afp.org/entries/Amortized\\_  
Complexity.shtml](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

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

## Corollary

$$\begin{aligned} & \llbracket bst\ t; x \in set\_tree\ t \rrbracket \\ & \implies A\_splay\ x\ t \leq 3 * (\varphi\ t - 1) + 1 \end{aligned}$$

## Corollary

$$bst\ t \implies A\_splay\ x\ t \leq 3 * \varphi\ t + 1$$

## Lemma

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

$$splay\ x'\ t = splay\ x\ t \wedge T\_splay\ x'\ t = T\_splay\ 303$$

## *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*  $t \implies$

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



# *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*  $t \implies$

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

# 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  $\Rightarrow$  *isin t a* eats up *t*

Otherwise:

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

## Solution 1:

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

Observer function returns new data structure:

### Definition

$isin \ t \ a =$

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

## Solution 2:

*isin = splay; is\_root*

Client uses *splay* before calling *is\_root*:

### Definition

*is\_root* :: 'a ⇒ 'a tree ⇒ bool

*is\_root* x t = (case t of  
    ⟨ ⟩ ⇒ False  
    | ⟨l, a, r⟩ ⇒ x = a)

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

⇒ *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 Hood Melville Queues
- 22 Skew Heap
- 23 Splay Tree
- 24 Pairing Heap**
- 25 More Verified Data Structures and Algorithms  
(in Isabelle/HOL)



# Archive of Formal Proofs

[https://www.isa-afp.org/entries/Pairing\\_Heap.shtml](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*

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

$merge\ h\ Empty = h$

$merge\ Empty\ h = h$

$merge\ (Hp\ x\ h_{sx} =: hx)\ (Hp\ y\ h_{sy} =: hy) =$   
 $(if\ x < y\ then\ Hp\ x\ (hy\ \# \ h_{sx})\ else\ Hp\ y\ (hx\ \# \ h_{sy}))$

Like function *link* for binomial heaps

## *del\_min*

*del\_min Empty = Empty*

*del\_min (Hp x hs) = pass<sub>2</sub> (pass<sub>1</sub> hs)*

*pass<sub>1</sub> (h<sub>1</sub> # h<sub>2</sub> # hs) = merge h<sub>1</sub> h<sub>2</sub> # pass<sub>1</sub> hs*

*pass<sub>1</sub> [] = []*

*pass<sub>1</sub> [v] = [v]*

*pass<sub>2</sub> [] = Empty*

*pass<sub>2</sub> (h # hs) = merge h (pass<sub>2</sub> 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

## 24 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. Sedgwick, 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 Hood Melville Queues
- 22 Skew Heap
- 23 Splay Tree
- 24 Pairing Heap
- 25 More Verified Data Structures and Algorithms  
(in Isabelle/HOL)**

# 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

↪ functional program

↪ 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](#)