Verified Analysis of Functional Data Structures with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2017-8-6
Lecture 1  Introduction
Lecture 2  Search Trees
Lecture 3  Priority Queues
Lecture 4  Amortized Complexity
Lecture 1

Introduction
1 Motivation
2 Time
3 Binary Trees
1 Motivation

2 Time

3 Binary Trees
General aims

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

Complexity analysis should be as algebraic as proving

\[ \binom{r}{m} \binom{m}{k} = \binom{r}{k} \binom{r-k}{m-k} \]

Part of a project on Verified Algorithm Analysis
This course

- An introduction to the basic methods
- A whirlwind tour of verified functional data structures (and some proofs)
- Specifically: “Classical” search trees and priority queues
- First functional versions:
- Now: verified algebraic proofs
  See src/HOL/Data_Structures in the Isabelle distribution or online
1 Motivation

2 Time

3 Binary Trees
2 Time

A Lightweight Approach
A Monadic Approach
Amortized Analysis
Principle: Count function calls

For every function \( f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau \)

define a **timing function** \( t_f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \text{nat} \):

Translation of defining equations:

\[
\begin{align*}
    e & \leadsto e' \\
    f \ p_1 \ldots p_n = e & \leadsto t_f \ p_1 \ldots p_n = e' + 1
\end{align*}
\]

Translation of expressions:

\[
\begin{align*}
    s_1 & \leadsto t_1 \quad \ldots \quad s_k & \leadsto t_k \\
    g \ s_1 \ldots s_k & \leadsto t_1 + \cdots + t_k + t_g \ s_1 \ldots s_k
\end{align*}
\]

- Variable \( \leadsto 0 \), Constant \( \leadsto 0 \)
- Constructor calls and primitive operations on \textit{bool} and numbers cost 1
Example

\[ app \; [] \; ys = ys \]
\[ \leadsto \]
\[ t\_app \; [] \; ys = 0 + 1 \]

\[ app \; (x\#xs) \; ys = x \# \; app \; xs \; ys \]
\[ \leadsto \]
\[ t\_app \; (x\#xs) \; ys = 0 + (0 + 0 + t\_app \; xs \; ys) + 1 + 1 \]
A compact formulation of $e \rightsquigarrow t$

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

If $g$ is

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

then $t_g \ldots = 1$. 
if and case

So far we model a call-by-value semantics

Conditionals and case expressions are evaluated lazily.

Translation:

\[
\begin{align*}
& b \Rightarrow t \quad s_1 \Rightarrow t_1 \quad s_2 \Rightarrow t_2 \\
\text{if } b \text{ then } s_1 \text{ else } s_2 \Rightarrow t + (\text{if } b \text{ then } t_1 \text{ else } t_2)
\end{align*}
\]

Similarly for case
\( O(.) \) is enough

\( \longrightarrow \) Reduce all additive constants to 1

Example

\[
t_{app} (x \# xs) \ ys = t_{app} \ xs \ ys + 1
\]
2 Time

A Lightweight Approach
A Monadic Approach
Amortized Analysis
• Start with the definition of a function $f_{tm}$ that computes both the result and the running time simultaneously
• Define $f_{tm}$ via a monad that counts function calls behind the scene
• Derive $f$ and $t_f$ from $f_{tm}$
The time monad

**datatype** `′a tm = TM ′a nat`

```ocaml
val (TM v n) = v
time (TM v n) = n
```

The programmer must only use:

**Bind:**

```ocaml
TM u m >>= f =
(let TM v n = f u in TM v (m+n))
```

```ocaml
return v = TM v 0
```

Equality plus 1 clock tick: `⇒`
Simple example

Using do-notation rather than >>=:

\[
app\_tm :: \text{'a list} \Rightarrow \text{'a list} \Rightarrow \text{'a list tm}
\]

\[
\text{app}\_tm [\text{ys}] \Rightarrow \text{return ys}
\]

\[
\text{app}\_tm (x \# xs) \text{ ys} \Rightarrow \text{do }
\]

\[
\text{zs} \leftarrow \text{app}_tm \text{ xs } y\text{s};
\]

\[
\text{return (x \# zs)}
\]

\}
Simple example

Define

\[ app \; xs \; ys = \text{val} \; (app\_tm \; xs \; ys) \]
\[ t\_app \; xs \; ys = \text{time} \; (app\_tm \; xs \; ys) \]

Prove automatically:

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

Prove with (some) insight:

\[ t\_app \; xs \; ys = \text{length} \; xs + 1 \]
2 Time

A Lightweight Approach
A Monadic Approach
Amortized Analysis
Recall

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

**Example**

Consider a $k$-bit binary counter. An individual increment has complexity $O(k)$. In a sequence of increments starting from 0, each increment has amortized constant complexity.
Implementation: Data structure of type $\tau$ with

- Operation(s) $f :: \tau \Rightarrow \tau$
  (may have additional parameters)
- Initial value: $\text{init} :: \tau$
  (function “empty”)

Claim: Operation $f$ has a certain amortized complexity of at most $a_f :: \tau \Rightarrow \text{num}$.
(Typically $a_f$ is constant or logarithmic)
Potential method

Find a *potential function* \( \Phi : \tau \Rightarrow \text{num} \) and prove

- \( \Phi \text{ init} = 0 \)
- \( \Phi \ s \geq 0 \)
- for each operation \( f \):
  \[ t_f \ s + \Phi(f \ s) - \Phi \ s \leq a_f \ s \]

What does this imply?
Amortized and real cost

A sequence of operations \( f_1, \ldots, f_n \) induces a sequence of states:

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

The real cost of performing \( f_1, \ldots, f_n \) is upper-bounded by amortized costs:

\[
\sum_{i=1}^{n} t_{-f_i} s_{i-1} \leq \sum_{i=1}^{n} a_{-f_i} s_{i-1}
\]
Warning

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

Single threaded = no value is used more than once

Otherwise:

\[
\text{let } \quad \text{counter} = 0; \\
\text{bad} = \text{increment } \text{counter} 2^n - 1 \text{ times}; \\
\_ = \text{incr bad}; \\
\_ = \text{incr bad}; \\
\_ = \text{incr bad}; \\
\vdots
\]
Okasaki’s insight

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

Not covered in this course
1 Motivation

2 Time

3 Binary Trees
Binary trees

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

Abbreviations:
⟨⟩ ≡ Leaf
⟨l, a, r⟩ ≡ Node l a r

Most of the time: tree = binary tree
Tree traversal

\[
\begin{align*}
inorder :: 'a \text{ tree} & \Rightarrow 'a \text{ list} \\
inorder \langle \rangle & = [] \\
inorder \langle l, x, r \rangle & = inorder l @ [x] @ inorder r
\end{align*}
\]
Number of nodes:

size :: 'a tree ⇒ nat

|⟨⟩| = 0

|⟨l, _, r⟩| = |l| + |r| + 1

Number of leaves:

size₁ :: 'a tree ⇒ nat

|t|₁ = |t| + 1
height :: 'a tree ⇒ nat

\[ h(\langle \rangle) = 0 \]
\[ h(\langle l, -, r \rangle) = \max (h(l)) (h(r)) + 1 \]

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

Search Trees
4 Correctness

5 2-3 Trees

6 Red-Black Trees
We assume that the elements in the search trees are linearly ordered

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

Instead of using < and ≤ directly:

\textbf{datatype} \ cmp\_val = \ LT \mid \ EQ \mid \ GT

\textit{cmp} :: \ 'a \Rightarrow \ 'a \Rightarrow \ cmp\_val

\textit{cmp} \ x \ y =
(if \ x < \ y \ then \ LT \ else \ if \ x = \ y \ then \ EQ \ else \ GT)
4 Correctness

5 2-3 Trees

6 Red-Black Trees
Search trees represent sets
Specification for sets?
Set interface

An implementation of sets of elements of type \( 'a \) must provide

- An implementation type \( 't \)
- \textit{empty} :: \( 't \)
- \textit{insert} :: \( 'a \Rightarrow 't \Rightarrow 't \)
- \textit{delete} :: \( 'a \Rightarrow 't \Rightarrow 't \)
- \textit{isin} :: \( 't \Rightarrow 'a \Rightarrow \text{bool} \)
4 Correctness

The Standard Specification

Correctness via Sorted Lists
A correct implementation must also provide

- an **abstraction function** `set :: 't ⇒ 'a set`
- and an **invariant** `invar :: 't ⇒ bool`

such that

\[
\begin{align*}
\text{set empty} &= \{\} \\
\text{set (insert } x \text{ t)} &= \text{set } t \cup \{x\} \\
\text{set (delete } x \text{ t)} &= \text{set } t - \{x\} \\
\text{isin } t \ x &= (x \in \text{set } t)
\end{align*}
\]

under the assumption `invar t`
Also: \( \textit{invar} \) must be invariant

\[
\begin{align*}
\text{invar empty} \\
\text{invar } t &\implies \text{invar } (\text{insert } x \; t) \\
\text{invar } t &\implies \text{invar } (\text{delete } x \; t)
\end{align*}
\]
Example: type 'a tree

Abstraction function: \( \text{set\_tree} :: 'a \text{ tree} \Rightarrow 'a \text{ set} \)

\[
\text{set\_tree} \langle \rangle = \{\}\n\]
\[
\text{set\_tree} \langle l, x, r \rangle = \text{set\_tree} l \cup \{x\} \cup \text{set\_tree} r
\]

Invariant: \( \text{bst} :: 'a \text{ tree} \Rightarrow \text{bool} \)

\[
\text{bst} \langle \rangle = \text{True}
\]
\[
\text{bst} \langle l, a, r \rangle =
\]
\[
(\text{bst} l \land
\]
\[
\text{bst} r \land
\]
\[
(\forall x \in \text{set\_tree} l. \ x < a) \land
\]
\[
(\forall x \in \text{set\_tree} r. \ a < x)
\)
This is the *standard approach*.

Empirical evidence: *proofs are hard(er) to automate*
4 Correctness

The Standard Specification

Correctness via Sorted Lists
sorted :: 'a list ⇒ bool

sorted [] = True
sorted [x] = True
sorted (x # y # zs) = (x < y ∧ sorted (y # zs))

No duplicates!
We assume that there is some structural invariant on the search tree:

\[ \text{inv} : t \Rightarrow \text{bool} \]

e.g. some balance criterion.
Correctness of \( \text{insert} \)

\[
\text{inv } t \land \text{sorted } (\text{inorder } t) \implies \\
\text{inorder } (\text{insert } x \ t) = \text{ins\_list } x (\text{inorder } t)
\]

where

\[
\text{ins\_list} :: 'a \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list}
\]

inserts an element into a sorted list.

Also covers preservation of \( \text{bst} \)
Correctness of \textit{delete}

\[\text{inv } t \land \text{sorted } (\text{inorder } t) \implies \text{inorder } (\text{delete } x \ t) = \text{del\_list } x \ (\text{inorder } t)\]

where

\[\text{del\_list :: } 'a \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list}\]

deletes an element from a sorted list.

\textbf{Also covers preservation of bst}
Correctness of \textit{isin}

\[\text{inv } t \land \text{sorted } (\text{inorder } t) \implies \text{isin } t \ x = (x \in \text{elems } (\text{inorder } t))\]

where

\texttt{elems :: 'a list } \Rightarrow \texttt{ 'a set}

converts a list into a set.
Correctness w.r.t. sorted lists implies the standard set specification and can be automated for all search trees in this course. Except for the structural invariants. Therefore we concentrate on the latter.

For details of the automation see
4 Correctness

5 2-3 Trees

6 Red-Black Trees
2-3 Trees

**datatype** 'a tree23 = ⟨⟩

| Node2 ('a tree23) 'a ('a tree23) |
| Node3 ('a tree23) 'a ('a tree23) 'a ('a tree23) |

Abbreviations:

⟨l, a, r⟩ ≡ Node2 l a r

⟨l, a, m, b, r⟩ ≡ Node3 l a m b r
Invariant \( \text{bal} \)

All leaves are at the same level:

\[
\text{bal} \langle \rangle = \text{True}
\]

\[
\text{bal} \langle l, \_ , r \rangle = (\text{bal} \ l \land \text{bal} \ r \land h(l) = h(r))
\]

\[
\text{bal} \langle l, \_ , m, \_ , r \rangle = \\
(\text{bal} \ l \land \text{bal} \ m \land \text{bal} \ r \land h(l) = h(m) \land h(m) = h(r))
\]

Lemma

\[
\text{bal} \ t \implies 2^{h(t)} \leq |t| + 1
\]
The idea:

\[ \text{Leaf} \xrightarrow{\sim} \text{Node2} \]
\[ \text{Node2} \xrightarrow{\sim} \text{Node3} \]
\[ \text{Node3} \xrightarrow{\sim} \text{overflow, pass 1 element back up} \]
Insertion

Two possible return values:

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

**datatype**

\[
'a \ up_i = T_i ('a \ tree23) \\
\mid Up_i ('a \ tree23) 'a ('a \ tree23)
\]

\[
tree_i :: 'a \ up_i \Rightarrow 'a \ tree23 \\
tree_i (T_i \ t) = t \\
tree_i (Up_i \ l \ a \ r) = \langle l, a, r \rangle
\]
Insertion

\[
\text{insert} :: \ 'a \Rightarrow \ 'a \ \text{tree23} \Rightarrow \ 'a \ \text{tree23} \\
\text{insert} \ x \ t = \ \text{tree}_i \ (\text{ins} \ x \ t) \\
\text{ins} :: \ 'a \Rightarrow \ 'a \ \text{tree23} \Rightarrow \ 'a \ \text{up}_i
\]
\[
\text{Insertion}
\]

\[
\text{ins } x \langle l, a, m, b, r \rangle =
\]
\[
\text{case } \text{cmp } x \ a \text{ of }
\]
\[
\text{LT } \Rightarrow \text{ case } \text{ins } x \ l \text{ of }
\]
\[
T \ i \ l' \Rightarrow T \ i \ \langle l', a, m, b, r \rangle
\]
\[
| \quad \text{Up} \ i \ l_1 \ c \ l_2 \Rightarrow \text{Up} \ i \ \langle l_1, c, l_2 \rangle \ a \ \langle m, b, r \rangle
\]
\[
| \quad \text{EQ } \Rightarrow \ T \ i \ \langle l, a, m, b, r \rangle
\]
\[
| \quad \text{GT } \Rightarrow
\]
\[
\text{case } \text{cmp } x \ b \text{ of }
\]
\[
\text{LT } \Rightarrow
\]
\[
\text{case } \text{ins } x \ m \text{ of }
\]
\[
T \ i \ m' \Rightarrow T \ i \ \langle l, a, m', b, r \rangle
\]
\[
| \quad \text{Up} \ i \ m_1 \ c \ m_2 \Rightarrow \text{Up} \ i \ \langle l, a, m_1 \rangle \ c \ \langle m_2, b, r \rangle
\]
\[
| \quad \text{EQ } \Rightarrow \ T \ i \ \langle l, a, m, b, r \rangle
\]
\[
| \quad \text{GT } \Rightarrow
\]
Insertion preserves \( bal \)

**Lemma**

\[
\text{bal } t \iff \text{bal } (\text{tree}_i (\text{ins } a \ t)) \land h(\text{ins } a \ t) = h(t)
\]

where \( h :: \ 'a \ \text{up}_i \Rightarrow \text{nat} \)

\[
h(T_i \ t) = h(t)
\]

\[
h(\text{Up}_i \ l \ a \ r) = h(l)
\]

**Proof** by induction on \( t \). Step automatic.

**Corollary**

\[
\text{bal } t \iff \text{bal } (\text{insert } a \ t)
\]
Deletion

The idea:

\[ \text{Node3} \rightsquigarrow \text{Node2} \]
\[ \text{Node2} \rightsquigarrow \text{underflow, height -1} \]

Underflow: merge with siblings on the way up
Deletion

Two possible return values:

- height unchanged: \( T_d \ t \)
- height decreased by 1: \( Up_d \ t \)

**datatype** \('a up_d = T_d ('a tree23) | Up_d ('a tree23)\)

\[ \text{tree}_d (T_d \ t) = t \]
\[ \text{tree}_d (Up_d \ t) = t \]
Deletion

\[
\text{delete} :: 'a \Rightarrow 'a \text{ tree23} \Rightarrow 'a \text{ tree23}
\]
\[
\text{delete } x \ t = \text{tree}_d \ (\text{del } x \ t)
\]

\[
\text{del} :: 'a \Rightarrow 'a \text{ tree23} \Rightarrow 'a \text{ upd}_d
\]
\[ \text{del } x \langle l, a, r \rangle = \]
\[
(\text{case } \text{cmp } x \ a \ of \]
\[ LT \Rightarrow \text{node21 } (\text{del } x \ l) \ a \ r \\
| \ EQ \Rightarrow \text{let } (a', t) = \text{del}_\text{min} \ r \ in \ \text{node22 } l \ a' \ t \\
| \ GT \Rightarrow \text{node22 } l \ a \ (\text{del } x \ r)) \]

\[
\text{node21 } \left( T_d \ t_1 \right) \ a \ t_2 = T_d \left( t_1, a, t_2 \right) \\
\text{node21 } \left( U_p d \ t_1 \right) \ a \left( t_2, b, t_3 \right) = U_p d \left( t_1, a, t_2, b, t_3 \right) \\
\text{node21 } \left( U_p d \ t_1 \right) a \left( t_2, b, t_3, c, t_4 \right) = \\
T_d \left( \langle t_1, a, t_2 \rangle, b, \langle t_3, c, t_4 \rangle \right) \]
Deletion preserves $bal$

After 13 simple lemmas:

**Lemma**

\[ bal\, t \implies bal\, (tree_d\, (del\, x\, t)) \]

**Corollary**

\[ bal\, t \implies bal\, (delete\, x\, t) \]
Beyond 2-3 trees

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

Like 2-3 trees, but with many more cases

The general case:

B-trees and \((a, b)\)-trees
4 Correctness

5 2-3 Trees

6 Red-Black Trees
Relationship to 2-3-4 trees

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

\[ \langle \rangle \approx \langle \rangle \]
\[ \langle t_1, a, t_2 \rangle \approx \langle t_1, a, t_2 \rangle \]
\[ \langle t_1, a, t_2, b, t_3 \rangle \approx \langle \langle t_1, a, t_2 \rangle, b, t_3 \rangle \langle t_1, a, \langle t_2, b, t_3 \rangle \rangle \]
\[ \langle t_1, a, t_2, b, t_3, c, t_4 \rangle \approx \langle \langle t_1, a, t_2 \rangle, b, \langle t_3, c, t_4 \rangle \rangle \]

Red means “I am part of a bigger node”
Red-black trees

datatype \textit{color} = \textit{Red} | \textit{Black}

datatype

\begin{quote}
\texttt{'}a rbt = \textit{Leaf} | \textit{Node \textit{color} ('a tree) 'a ('a tree)}
\end{quote}

Abbreviations:

\begin{quote}
\begin{align*}
\langle \rangle & \equiv \textit{Leaf} \\
\langle c, l, a, r \rangle & \equiv \textit{Node} c l a r \\
R l a r & \equiv \textit{Node} \textit{Red} l a r \\
B l a r & \equiv \textit{Node} \textit{Black} l a r
\end{align*}
\end{quote}
\[
\text{color} :: \ 'a \ rbt \Rightarrow \ color \\
\text{color} \ \langle \rangle = \text{Black} \\
\text{color} \ \langle c, \ , \ , \ \rangle = \ c \\

\text{paint} :: \ \text{color} \Rightarrow \ 'a \ rbt \Rightarrow \ 'a \ rbt \\
\text{paint} \ c \ \langle \rangle = \ \langle \rangle \\
\text{paint} \ c \ \langle \ , \ l, \ a, \ r \rangle = \ \langle c, \ l, \ a, \ r \rangle
\]
Invariants

\[ rbt :: 'a rbt \Rightarrow bool \]
\[
rbt t = (invc t \land invh t \land color t = \textit{Black})
\]

\[ invc :: 'a rbt \Rightarrow bool \]
\[
invc \langle \rangle = \textit{True}
\]
\[
invc \langle c, l, \_ , r \rangle =
\]
\[
(\text{invc } l \land
\text{invc } r \land
(c = \textit{Red} \rightarrow color l = \textit{Black} \land color r = \textit{Black}))\]
Invariants

invh :: 'a rbt ⇒ bool
invh ⟨⟩ = True
invh ⟨_, l, _, r⟩ = (invh l ∧ invh r ∧ bh(l) = bh(r))

bheight :: 'a rbt ⇒ nat
bh(⟨⟩) = 0
bh(⟨c, l, _, _⟩) =
(if c = Black then bh(l) + 1 else bh(l))
Logarithmic height

Lemma

\[ rbt \ t \implies h(t) \leq 2 \times \log_2 |t|_1 \]
**Insertion**

\[
\text{insert :: } 'a \Rightarrow 'a \ rbt \Rightarrow 'a \ rbt \\
\text{insert } x \ t = \text{paint Black} \ (\text{ins } x \ t) \\
\text{ins :: } 'a \Rightarrow 'a \ rbt \Rightarrow 'a \ rbt \\
\text{ins } x \ \langle \rangle = R \ \langle \rangle \ x \ \langle \rangle \\
\text{ins } x \ (B \ l \ a \ r) = (\text{case } \text{cmp } x \ a \ \text{of} \\
    \text{LT } \Rightarrow \ \text{baliL} \ (\text{ins } x \ l) \ a \ r \\
    \text{EQ } \Rightarrow \ B \ l \ a \ r \\
    \text{GT } \Rightarrow \ \text{baliR} \ l \ a \ (\text{ins } x \ r)) \\
\text{ins } x \ (R \ l \ a \ r) = (\text{case } \text{cmp } x \ a \ \text{of} \\
    \text{LT } \Rightarrow \ R \ (\text{ins } x \ l) \ a \ r \\
    \text{EQ } \Rightarrow \ R \ l \ a \ r \\
    \text{GT } \Rightarrow \ R \ l \ a \ (\text{ins } x \ r))
\]
Adjusting colors

\[ \text{baliL, baliR :: } 'a \text{ rbt} \Rightarrow 'a \Rightarrow 'a \text{ rbt} \Rightarrow 'a \text{ rbt} \]

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

After 14 simple lemmas:

**Theorem**

\[ \text{rbt } t \iff \text{rbt} \left( \text{insert } x \ t \right) \]
The while loop in lines 1–15 maintains the following three-part invariant at the start of each iteration of the loop:

a. Node \( x \) is red.

b. If \( x \) is the root, then \( x \) is black.

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 3 or property 4. If the tree violates property 3, then \( x \) is the root and is red. If the tree violates property 4, then it is because both \( y \) and \( z \) are red.

Part (c), which deals with violations of red-black properties, is more central to showing that RB-INSERT-FIXUP restores the red-black properties than parts (a) and (b), which we have done along the way to understand situations in the code. Because we’re focusing on node \( x \) and nodes near it in the tree, it helps to know from past parts (a) and (b) that when we reference it in lines 2, 3, 8, 13, and 14.

Recall that we need to show that a loop invariant is true prior to the first iteration of the loop, that each iteration maintains the loop invariant, and that the loop invariant gives us a useful property at loop termination. Along the way, we shall also demonstrate that each iteration of the loop has two possible outcomes, either the pointer \( x \) moves up the tree, or we perform some rotations and then the loop terminates.

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

a. When RB-INSERT-FIXUP is called, \( x \) is the red node that was added.

b. If \( x \) is the root, then \( x \) started out black and did not change prior to the call of RB-INSERT-FIXUP.

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

If the tree violates property 2, then the root node 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 violations prior to \( c \) being added, the violation must be because both \( y \) and \( z \) are red.

**Termination:** When the loop terminates, it does so because \( z.p \) is black. (\( z \) is the root, then \( z \) is the sentinel \( T.o \), which is black.) Thus, the tree does not violate property 3 at termination. By the loop invariant, the only property violated by the tree is property 4 at the start of this iteration. By the loop invariant, the only property violated by the tree is property 4 at the start of this iteration. Hence, property 4 exists. We distinguish case 1 from cases 2 and 3 by the color of \( p \)’s parent’s sibling, or "uncle." Line 3 makes \( y \) point to \( c \)’s uncle, \( p \), and line 4 tests \( y \’s \) color.

If \( y \) is red, we execute case 1. Otherwise, control passes to case 2 and 3. In all three cases, \( y \’s \) grandparent \( z \) is black, since its parent \( p \) is red, and property 4 is violated only between \( c \) and \( z \).

**Case 1:** \( y \) is red and \( z \) is red at the start of the next iteration.

a. Because this iteration colors \( y \), \( z \), and \( c \) red, \( d \) is red at the start of the next iteration.

b. The node \( c \)’s parent, \( p \), is in this iteration, and the color of this node does not change. If this node is in the root, it was black prior to this iteration, and it remains black at the start of the next iteration.

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

delete x t = paint Black (del x t)

del _ ⟨⟩ = ⟨⟩
del x ⟨_, l, a, r⟩ =

(case cmp x a of
  LT ⇒
    if l ≠ ⟨⟩ ∧ color l = Black
    then baldL (del x l) a r else R (del x l) a r
  | EQ ⇒ combine l r
  | GT ⇒
    if r ≠ ⟨⟩ ∧ color r = Black
    then baldR l a (del x r) else R l a (del x r))
Deletion

Tricky functions: *baldL, baldR, combine*

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

\[
\left[ \text{invh } t; \text{invc } t \right] \\
\implies \text{invh } (\text{del } x \ t) \land \\
\left( \text{color } t = \text{Red} \land \\
\text{bh}(\text{del } x \ t) = \text{bh}(t) \land \text{invc } (\text{del } x \ t) \lor \\
\text{color } t = \text{Black} \land \\
\text{bh}(\text{del } x \ t) = \text{bh}(t) - 1 \land \text{invc} \!2 \!\! \!\! \!\! \!\! (\text{del } x \ t) \right)
\]

**Theorem**

\[
\text{rbt } t \implies \text{rbt } (\text{delete } k \ t)
\]
13.4 Deletion

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

The procedure for deleting a node from a red-black tree is based on the TREE-DELETE subroutine in Section 13.2. First, we need to understand the TRANSPLANT subroutine that TREE-DELETE calls so that it applies to a red-black tree:

**RB-TRANSPLANT(x, y)**
- If $x$ is red, we're done.
- Else if $x$ is black, we must either make $y$ the root and reorient $x$’s parent or make $y$’s parent $x$’s parent.
- Incidentally, $x$’s parent is $y$’s original parent.

**Algorithm RB-DELETE**

1. Let $x$ be the node to be deleted.
2. Let $y$ be found as the node to be used interchangeably with $x$.
3. If $x$ is red, we're done.
4. If $x$ is black:
   - If $x$'s left or right child is red, color $x$ black and make the child red.
   - Else $x$'s parent is black. If $x$'s parent has two red children, color $x$'s parent black, its left child $y$ red, and move $y$ into $x$'s place.
5. If $y$'s original color is red, we're done.
6. Else $(y$ is transformed to be the root).

**Case 1:** $x$ is a left child. (See Figure 13.7(a).) (If $x$ is right child, replace $x$ with $y$ in the tree, $x$'s original parent is $y$'s original parent, and $x$'s color is $y$'s original color; the new sibling of $x$ is $y$’s right child.)
- $p$ is black, so $p$'s left child $x$ was also black and $y$’s root is red.
  - $y$: color $= 0$
  - $x$: color = red from the root to any of the black nodes from the root to any of the red nodes.

**Case 2:** $x$ is a right child, the new sibling of $x$ is red, and $x$'s original parent is $y$’s original parent.
- $p$ is black, so $p$’s right child $x$ was also black and $y$’s root is red.
  - $y$: color $= 0$
  - $x$: color = red

**Case 3:** $x$'s original parent is $y$’s original parent, and $x$'s original right child is red.
- $p$ is black, so $p$’s right child $x$'s original parent is $y$’s original parent.
  - $y$: color $= 0$
  - $x$: color = red

**Case 4:** $x$’s original right child is red.
- $p$ is black, so $p$’s right child $x$’s original parent is $y$’s original parent.
  - $y$: color $= 0$
  - $x$: color = red

**Conclusion:**

- We know how to apply to a red-black tree.
- We can now see the procedure RB-DELETE in detail, let’s look more generally at how we can verify that the transformation holds afterward. For example, in Figure 13.7(a), which illustrates case 1, the number of black nodes from the root to any node is $O(\lg n)$, the height of the tree.

**Analysis:**

What is the running time of RB-DELETE? Since the height of a red-black tree is $O(\lg n)$, the total cost of the procedure without call to RB-TRANSPLANT is $O(\lg n)$ time. Within RB-DELETE, each of cases 1, 2, and 3 lead to at most one rotation and at most three rotations, and the overall running time for RB-DELETE is therefore $O(\lg n)$.
Source of code

Insertion:
Okasaki’s *Purely Functional Data Structures*

Deletion:
Lecture 3

Priority Queues
7 Priority Queues

8 Leftist Heap

9 Priority Queue via Braun Tree

10 Binomial Heap
Priority Queues

Leftist Heap

Priority Queue via Braun Tree

Binomial Heap
Priority queue informally

Collection of elements with priorities

We focus on the priorities:
\[ \text{element} = \text{priority} \]

The same element can be contained \textit{multiple times} in a priority queue

\[ \Rightarrow \]

The abstract view of a priority queue is a \textit{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 \( 'q \)
- \( \text{empty} :: 'q \)
- \( \text{is_empty} :: 'q \Rightarrow \text{bool} \)
- \( \text{insert} :: 'a \Rightarrow 'q \Rightarrow 'q \)
- \( \text{get_min} :: 'q \Rightarrow 'a \)
- \( \text{del_min} :: 'q \Rightarrow 'q \)
More operations

- $\text{merge} :: 'q \Rightarrow 'q \Rightarrow 'q$
  Often provided
- decrease key/priority
  Not easy in functional setting
Correctness of implementation

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

Abstraction function: \( mset :: 'q \Rightarrow 'a \text{ multiset} \)

Invariant: \( invar :: 'q \Rightarrow \text{ bool} \)
Correctness of implementation

Must prove \( \text{invar } q \implies \)

\[
\begin{align*}
\text{mset } (\text{insert } x \ q) &= \text{mset } q + \{#x#\} \\
\text{mset } q \neq \{#\} &\implies \\
\text{mset } (\text{del}_\text{min} \ q) &= \text{mset } q - \{#\text{Min}_\text{mset} (\text{mset } q)#\}
\end{align*}
\]

\( \text{invar } (\text{insert } x \ q) \)
\( \text{invar } (\text{del}_\text{min} \ q) \)
Terminology

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

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

- \( \text{get\_min} \langle _, a, _ \rangle = a \)
- Assume we have \( \text{merge} \)
- \( \text{insert} \ a \ t = \text{merge} \langle \langle \rangle, a, \langle \rangle \rangle \ t \)
- \( \text{del\_min} \langle l, a, r \rangle = \text{merge} \ l \ r \)
7 Priority Queues

8 Leftist Heap

9 Priority Queue via Braun Tree

10 Binomial Heap
Leftist tree informally

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

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

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

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

datatype

'\texttt{a lheap} = \texttt{Leaf} \mid \texttt{Node nat ('a tree) 'a ('a tree)}

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

Abstraction function:

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

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

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

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

\[ \text{rank :: 'a lheap} \Rightarrow \text{nat} \]
\[ \text{rank } \langle \rangle = 0 \]
\[ \text{rank } \langle -, -, - , r \rangle = \text{rank } r + 1 \]

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

\[ \text{ltree :: 'a lheap} \Rightarrow \text{bool} \]
\[ \text{ltree } \langle \rangle = \text{True} \]
\[ \text{ltree } \langle n, l, -, r \rangle = \]
\[ (n = \text{rank } r + 1 \land \text{rank } r \leq \text{rank } l \land \text{ltree } l \land \text{ltree } r) \]
Leftist heap invariant

\[ \text{invar } h = (\text{heap } h \land \text{ltree } h) \]
merge

Principle: descend on the right

\[
\text{merge} \langle \rangle \ t_2 = t_2 \\
\text{merge} \ t_1 \langle \rangle = t_1 \\
\text{merge} \langle n_1, l_1, a_1, r_1 \rangle \langle n_2, l_2, a_2, r_2 \rangle = \\
(\text{if } a_1 \leq a_2 \text{ then node } l_1 \ a_1 \ (\text{merge } r_1 \langle n_2, l_2, a_2, r_2 \rangle)) \\
\text{else node } l_2 \ a_2 \ (\text{merge } r_2 \langle n_1, l_1, a_1, r_1 \rangle))
\]

\[
\text{node} :: 'a \ \text{lheap} \Rightarrow 'a \Rightarrow 'a \ \text{lheap} \Rightarrow 'a \ \text{lheap}
\]

\[
\text{node} \ l \ a \ r = \\
(\text{let } rl = \text{rk} \ l; \ rr = \text{rk} \ r \\
\text{in if } rr \leq rl \text{ then } \langle rr + 1, l, a, r \rangle \text{ else } \langle rl + 1, r, a, l \rangle)
\]

where \[ \text{rk} \langle n, \_, \_, \_ \rangle = n \]
Functional correctness proofs

including preservation of *invar*

Straightforward
Logarithmic complexity

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

Corollary
\[ [ltree \ l; \ ltree \ r] \implies t_{\text{merge}} \ l \ r \leq \log_2 |l|_1 + \log_2 |r|_1 + 1 \]
Can we avoid the rank info in each node?
7 Priority Queues
8 Leftist Heap
9 Priority Queue via Braun Tree
10 Binomial Heap
Archive of Formal Proofs

https://www.isa-afp.org/entries/Priority_Queue_Braun.shtml
What is a Braun tree?

\[ \text{braun} :: 'a \text{ tree} \Rightarrow \text{bool} \]

\[ \text{braun} \langle \rangle = \text{True} \]

\[ \text{braun} \langle l, x, r \rangle = (|r| \leq |l| \land |l| \leq |r| + 1 \land \text{braun} l \land \text{braun} r) \]

**Lemma** \( \text{braun} \ t \iff 2^{h(t)} \leq 2 \times |t| + 1 \)
Idea of invariant maintenance

\( \text{braun} \langle \rangle = \text{True} \)

\( \text{braun} \langle l, x, r \rangle = \)

\( (|r| \leq |l| \land |l| \leq |r| + 1 \land \text{braun} \, l \land \text{braun} \, r) \)

Add element: to right subtree, then swap subtrees

Remove element: from left subtree, then swap subtrees
Priority queue implementation

Implementation type: 'a tree

Invariants: heap and braun

No merge — insert and del_min defined explicitly
insert :: '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.
**$\text{del}_\text{min}$**

\[
\text{del}_\text{min} :: 'a \text{ tree} \Rightarrow 'a \text{ tree}
\]

\[
\text{del}_\text{min} \langle \rangle = \langle \rangle
\]

\[
\text{del}_\text{min} \langle \langle \rangle, x, r \rangle = \langle \rangle
\]

\[
\text{del}_\text{min} \langle l, x, r \rangle =
\]

\[
(\text{let } (y, l') = \text{del}_\text{left} l \text{ in } \text{sift}_\text{down} r y l')
\]

1. Delete leftmost element $y$
2. Sift $y$ from the root down
$$\text{del\_left}$$

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

$$\text{del\_left}\ \langle\langle\rangle,\ x,\ \langle\rangle\rangle = (x,\ \langle\rangle)$$

$$\text{del\_left}\ \langle l,\ x,\ r\rangle =$$

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

sift_down (t₁ = (Node l₁ x₁ r₁)) a (t₂ = (Node l₂ x₂ r₂)) =
if a ≤ x₁ ∧ a ≤ x₂ then ⟨t₁, a, t₂⟩
else if x₁ ≤ x₂ then ⟨sift_down l₁ a r₁, x₁, t₂⟩
else ⟨t₁, x₂, sift_down l₂ a r₂⟩
Functional correctness proofs for \textit{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: \( braun \ t \iff 2^{h(t)} \leq 2 \times |t| + 1 \)

\[ \implies \]

Above running times logarithmic in size
Based on code from
based on code from Chris Okasaki.
7 Priority Queues
8 Leftist Heap
9 Priority Queue via Braun Tree
10 Binomial Heap
Numerical method

Idea: only use trees $t_i$ of size $2^i$

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

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

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

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

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

Lemma

\[
invar_btree t \rightarrow |t| = 2^{\text{rank } t}
\]
```
Combining two trees

How to combine two trees of rank \(i\)
into one tree of rank \(i+1\)

\[
\text{link } t_1 \ t_2 = \\
(\text{case } (t_1, t_2) \text{ of } \\
\quad (\text{Node } r \ x_1 \ c_1, \ \text{Node } x \ x_2 \ c_2) \Rightarrow \\
\quad \text{if } x_1 \leq x_2 \text{ then Node } (r + 1) \ x_1 \ (t_2 \not= c_1) \\
\quad \text{else Node } (r + 1) \ x_2 \ (t_1 \not= c_2))
\]
Binomial heap

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

\textbf{type_synonym} 'a heap = 'a tree list

Remember: \textit{tree} contains rank

Invariant:

\begin{verbatim}
invar_bheap ts =
((\forall t \in \text{set ts}. \ \text{invar_btree t}) \land
\text{strictly_ascending (map rank ts))}
\end{verbatim}
Inserting a tree

\[
\text{ins_tree } t \[\] = [t]
\]

\[
\text{ins_tree } t_1 \ (t_2 \ # \ ts) =
\]

(\text{if rank } t_1 < \text{rank } t_2 \ \text{then } t_1 \ # \ t_2 \ # \ ts

\text{else } \text{ins_tree} \ (\text{link} \ t_1 \ t_2) \ ts)

Intuition: Handle a carry

Precondition:
Rank of inserted tree \( \leq \) ranks of trees in heap
merge

merge \(ts_1\) [] = \(ts_1\)
merge [] \(ts_2\) = \(ts_2\)
merge \((t_1 \# ts_1\) \(t_2 \# ts_2) =
(if \text{rank} t_1 < \text{rank} t_2 \text{ then } t_1 \# \text{merge } ts_1 (t_2 \# ts_2)
 else if \text{rank} t_2 < \text{rank} t_1 \text{ then } t_2 \# \text{merge } (t_1 \# ts_1) ts_2
 else \text{ins_tree} (\text{link } t_1 t_2) (\text{merge } ts_1 ts_2))

Intuition: Addition of binary numbers
Note: Handling of carry after recursive call
Find/delete minimum element

All trees are min-heaps.
Smallest element may be any root node:

\[ ts \neq [] \implies \text{find_min } ts = \text{Min} \left( \text{set} \left( \text{map} \text{ root} \ ts \right) \right) \]

Similar: \( \text{get_min} :: \text{'a tree list} \Rightarrow \text{'a tree} \times \text{'a tree list} \)
Returns tree with minimal root, and remaining trees

\[ \text{delete_min } ts = \]
\[ \text{(case get_min } ts \text{ of} \]
\[ \quad \text{(Node} r x ts_1, ts_2) \Rightarrow \text{merge} \left( \text{rev} ts_1 \right) ts_2 \) \]
Recall: $|t| = 2^{\text{rank } t}$

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

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

Proofs: straightforward?
Complexity of $merge$

$merge \ (t_1 \ # \ ts_1) \ (t_2 \ # \ ts_2) =$

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

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

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

$\sum \in O(n^2)$
Complexity of \( \text{merge} \)

\[
\text{merge} \left( t_1 \# ts_1 \right) \left( t_2 \# ts_2 \right) = \\
\left( \text{if rank } t_1 < \text{rank } t_2 \text{ then } t_1 \# \text{merge } ts_1 \left( t_2 \# ts_2 \right) \right) \\
\left( \text{else if rank } t_2 < \text{rank } t_1 \text{ then } t_2 \# \text{merge } \left( t_1 \# ts_1 \right) ts_2 \right) \\
\left( \text{else ins_tree } \left( \text{link } t_1 t_2 \right) \left( \text{merge } ts_1 ts_2 \right) \right)
\]

Relate time and length of input/output:

\[
t_{\text{ins_tree}} t ts + \text{length } \left( \text{ins_tree } t ts \right) = 2 + \text{length } ts \\
\text{length } \left( \text{merge } ts_1 ts_2 \right) + t_{\text{merge}} ts_1 ts_2 \\
\leq 2 \times \left( \text{length } ts_1 + \text{length } ts_2 \right) + 1
\]

Yields desired linear bound!
Lecture 4

Amortized Complexity
11 Skew Heap

12 Splay Tree

13 Pairing Heap
Archive of Formal Proofs

https://www.isa-afp.org/entries/Amortized_Analysis.shtml
11 Skew Heap
12 Splay Tree
13 Pairing Heap
A *skew heap* is a self-adjusting heap (priority queue)

Almost like leftist heap, but *no size info needed*
\[
\text{merge } \langle \rangle \ h = h \\
\text{merge } h \langle \rangle = h
\]

Swap subtrees when descending:
\[
\text{merge } (h_1 = \langle l_1, a, r_1 \rangle) \ (h_2 = \langle l_2, b, r_2 \rangle) = \begin{cases} 
\text{if } a \leq b \ \text{then } \langle \text{merge } h_2 \ r_1, a, l_1 \rangle \\
\text{else } \langle \text{merge } h_1 \ r_2, b, l_2 \rangle
\end{cases}
\]
Functional correctness proofs

Straightforward
Theorem

$$t_{\text{merge}} t_1 t_2 + \Phi (\text{merge} t_1 t_2) - \Phi t_1 - \Phi t_2 \leq 3 \times \log_2 (|t_1|_1 + |t_2|_1) + 1$$
Towards the proof

\[ rheavy \langle l, -, r \rangle = (|l| < |r|) \]

\[ lpath \langle \rangle = [] \]

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

\[ G \ h = \text{length} (\text{filter} \ rheavy (lpath h)) \]

**Lemma**

\[ 2^{G \ h} \leq |h| + 1 \]

**Corollary**

\[ G \ h \leq \log_2 |h|_1 \]
Towards the proof

$r_{\text{heavy}} \langle l, -, r \rangle = (|l| < |r|)$

$r_{\text{path}} \langle \rangle = []$

$r_{\text{path}} \langle l, a, r \rangle = \langle l, a, r \rangle \neq r_{\text{path}} r$

$D \ h = \text{length}(\text{filter}(\lambda p. \neg r_{\text{heavy}} p)(r_{\text{path}} h))$

**Lemma**

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

**Corollary**

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

The potential is the number of \textit{rheavy} nodes:

\[
\Phi \langle \rangle = 0 \\
\Phi \langle l, -, r \rangle = \Phi l + \Phi r + (\text{if } |l| < |r| \text{ then } 1 \text{ else } 0)
\]

Lemma

\[
t_{\text{merge}} t_1 t_2 + \Phi (\text{merge } t_1 t_2) - \Phi t_1 - \Phi t_2 \\
\leq G (\text{merge } t_1 t_2) + D t_1 + D t_2 + 1
\]

by (induction \texttt{t1 t2} rule: \texttt{merge.induct})(auto)
Main proof

\[ t_{\text{merge}} t_1 t_2 + \Phi (\text{merge } t_1 t_2) - \Phi t_1 - \Phi t_2 \leq G (\text{merge } t_1 t_2) + D t_1 + D t_2 + 1 \]
\[ \leq \log_2 |\text{merge } t_1 t_2|_1 + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1 \]
\[ = \log_2 (|t_1|_1 + |t_2|_1 - 1) + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1 \]
\[ \leq \log_2 (|t_1|_1 + |t_2|_1) + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1 \]
\[ \leq \log_2 (|t_1|_1 + |t_2|_1) + 2 * \log_2 (|t_1|_1 + |t_2|_1) + 1 \]

because \( \log_2 x + \log_2 y \leq 2 * \log_2 (x + y) \) if \( x, y > 0 \)
\[ = 3 * \log_2 (|t_1|_1 + |t_2|_1) + 1 \]
Sources

The inventors of skew heaps:
Daniel Sleator and Robert Tarjan.  
Self-adjusting Heaps.  

The formalization is based on  
Anne Kaldewaij and Berry Schoenmakers.  
The Derivation of a Tighter Bound for Top-down Skew Heaps.  
11 Skew Heap

12 Splay Tree

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

Functions *isin, insert and delete* have amortized logarithmic complexity.
Splay Tree
Algorithm
Amortized Analysis
Splay tree

Implementation type $=$ binary tree

Key operation \textit{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 \textit{isin}/\textit{insert}/\textit{delete} $a$ :

1. \textit{splay} $a$
2. Perform \textit{isin}/\textit{insert}/\textit{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 $= \text{two single rotations}$
Functional definition

\[ \text{splay} :: 'a \Rightarrow 'a \text{ tree} \Rightarrow 'a \text{ tree} \]
Zig-zig and zig-zag

\[ [a < y; a < z; T \neq \langle \rangle] \]
\[ \implies \text{splay } a \langle \langle T, y, C \rangle, z, D \rangle = \]
\[ \quad \text{(case splay } a \text{ } T \text{ of} \]
\[ \quad \langle A, x, B \rangle \Rightarrow \langle A, x, \langle B, y, \langle C, z, D \rangle \rangle \rangle) \]

\[ [a < z; y < a; T \neq \langle \rangle] \]
\[ \implies \text{splay } a \langle \langle A, y, T \rangle, z, D \rangle = \]
\[ \quad \text{(case splay } a \text{ } T \text{ of} \]
\[ \quad \langle B, x, C \rangle \Rightarrow \langle \langle A, y, B \rangle, x, \langle C, z, D \rangle \rangle) \]
Functional correctness proofs

Automatic
Splay Tree
Algorithm
Amortized Analysis
Potential

Sum of logarithms of the size of all nodes:

\[ \Phi \langle \rangle = 0 \]
\[ \Phi \langle l, a, r \rangle = \Phi l + \Phi r + \phi \langle l, a, r \rangle \]

where \( \phi t = \log_2 (|t| + 1) \)

Amortized complexity of \textit{splay}:

\[ a_{\text{splay}} a t = t_{\text{splay}} a t + \Phi (splay a t) - \Phi t \]
Analysis of \textit{splay}

Theorem
\[
[bst \ t; \ \langle l, \ x, \ r \rangle \in \text{subtrees} \ t] \\
\implies a\_splay \ x \ t \leq 3 \ast (\varphi \ t - \varphi \langle l, \ x, \ r \rangle) + 1
\]

Corollary
\[
[bst \ t; \ a \in \text{set\_tree} \ t] \\
\implies a\_splay \ a \ t \leq 3 \ast (\varphi \ t - 1) + 1
\]

Corollary
\[
bst \ t \implies a\_splay \ a \ t \leq 3 \ast \varphi \ t + 1
\]

Lemma
\[
[t \neq \langle \rangle; \ bst \ t] \\
\implies \exists a' \in \text{set\_tree} \ t. \\
\quad splay \ a' \ t = splay \ a \ t \land t\_splay \ a' \ t = t\_splay \ a \ t
\]
Definition

\[
\text{insert } x \ t = \\
(\text{if } t = \langle \rangle \ \text{then } \langle \langle \rangle, x, \langle \rangle \rangle \\
\text{else case } \text{splay } x \ t \ \text{of} \\
\langle l, a, r \rangle \Rightarrow \\
\quad \text{if } x = a \ \text{then } \langle l, a, r \rangle \\
\quad \text{else if } x < a \ \text{then } \langle l, x, \langle \langle \rangle, a, r \rangle \rangle \\
\quad \text{else } \langle \langle l, a, \langle \rangle \rangle, x, r \rangle)
\]

Counting only the cost of \text{splay}:

Lemma

\[
\text{bst } t \Rightarrow \\
\text{t-splay } a \ t + \Phi \ (\text{insert } a \ t) - \Phi \ t \leq 4 \ast \varphi \ t + 2
\]
Definition

\[\text{delete } x \, t = \]
\[
\begin{cases} 
\langle \rangle & \text{if } t = \langle \rangle \\
\text{case } \text{splay } x \, t \text{ of} \\
\langle l, a, r \rangle \Rightarrow & \text{if } x = a \\
\text{then if } l = \langle \rangle \text{ then } r \\
\text{else case } \text{splay} \text{ max } l \text{ of} \\
\langle l', m, r' \rangle \Rightarrow & \langle l', m, r \rangle \\
\text{else } \langle l, a, r \rangle & 
\end{cases}
\]

Lemma

\[\text{bst } t \implies \]
\[t_{\text{delete}} \, a \, t + \Phi (\text{delete } a \, t) - \Phi t \leq 6 \times \varphi \, t + 2\]
\[ isin :: 'a \text{ tree} \Rightarrow 'a \Rightarrow \text{bool} \]

Single threaded \[\implies isin\ t\ a\ \text{eats up}\ t\]

Otherwise:

\[\text{let}\ \text{bad} = \text{build unbalanced splay tree};\]
\[\_ = isin\ \text{bad}\ a;\]
\[\_ = isin\ \text{bad}\ a;\]
\[\_ = isin\ \text{bad}\ a;\]
\[\_ = isin\ \text{bad}\ a;\]
\[\_ = isin\ \text{bad}\ a;\]
Solution 1:

\[ isin :: \textquote{'}a \text{ tree} \Rightarrow \textquote{'}a \Rightarrow \text{bool} \times \textquote{'}a \text{ tree} \]

Observer function returns new data structure:

**Definition**

\[ isin \ t \ a = \]

\[
(\text{let } t' = splay \ a \ t \ \text{in} \ (\text{case } t' \ \text{of} \]

\[ \langle \rangle \Rightarrow \text{False} \]

\[ |\langle l, x, r \rangle \Rightarrow a = x, \]

\[ t'\)) \]
Solution 2:

\[ isin = \text{splay}; \text{is}_{\text{root}} \]

Client uses \textit{splay} before calling \textit{is\_root}:

**Definition**

\[
is_{\text{root}} :: 'a \Rightarrow 'a \, \text{tree} \Rightarrow \text{bool}\]

\[
is_{\text{root}} \, a \, t = (\text{case} \, t \, \text{of} \quad
\langle \rangle \Rightarrow \text{False} \quad
| \langle l, x, r \rangle \Rightarrow x = a)\]

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

\[\rightarrow is_{\text{root}} \_ t \text{ does not eat up } t\]
Splay trees have an imperative flavour and are a bit awkward to use in a purely functional language
The inventors of splay trees:
Daniel Sleator and Robert Tarjan. 

The formalization is based on
Skew Heap

Splay Tree

Pairing Heap
Implementation type

**datatype** 
\[ \text{'a heap} = \text{Empty} \mid \text{Hp 'a ('a heap list)} \]

Heap invariant:
\[
\text{pheap Empty} = \text{True}
\]
\[
\text{pheap (Hp x hs)} = \\
(\forall h \in \text{set hs.} \ (\forall y \in \text{set_heap h.} \ x \leq y) \land \text{pheap h})
\]

Also: \text{Empty} must only occur at the root
**insert**

\[
\text{insert } x \ h = \text{merge } (Hp \ x \ [];) \ h
\]

\[
\text{merge } h \ \text{Empty} = h
\]

\[
\text{merge } \text{Empty} \ h = h
\]

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

Like function \text{link} for binomial heaps
\text{del\_min}

\text{del\_min Empty} = \text{Empty} \\
\text{del\_min (Hp x hs)} = \text{merge\_pairs hs}

\text{merge\_pairs [\]} = \text{Empty} \\
\text{merge\_pairs [h]} = h \\
\text{merge\_pairs (h$_1$ \# h$_2$ \# hs)} = \\
\text{merge (merge h$_1$ h$_2$) (merge\_pairs hs)}
merge_pairs

\[\text{merge} \_ \text{pairs} \ [\ ] = \text{Empty}\]
\[\text{merge} \_ \text{pairs} \ [h] = h\]
\[\text{merge} \_ \text{pairs} \ (h_1 \ # \ h_2 \ # \ hs) =\]
\[\text{merge} \ (\text{merge} \ h_1 \ h_2) \ (\text{merge} \_ \text{pairs} \ hs)\]

\[\text{merge} \_ \text{pairs} \ hs = \text{pass}_2 \ (\text{pass}_1 \ hs)\]

\[\text{pass}_1 \ [\ ] = [\ ]\]
\[\text{pass}_1 \ [h] = [h]\]
\[\text{pass}_1 \ (h_1 \ # \ h_2 \ # \ hs) = \text{merge} \ h_1 \ h_2 \ # \ \text{pass}_1 \ hs\]

\[\text{pass}_2 \ [\ ] = \text{Empty}\]
\[\text{pass}_2 \ (h \ # \ hs) = \text{merge} \ h \ (\text{pass}_2 \ hs)\]
Functional correctness proofs

Straightforward
13 Pairing Heap

Amortized Analysis
Analysis

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

\[
\text{homs} :: \ 'a \ \text{heap} \ \text{list} \ \rightarrow \ 'a \ \text{tree}
\]
\[
\text{homs} \ [\ ] = \langle \rangle
\]
\[
\text{homs} \ (\text{Hp} \ x \ \text{hs}_1 \ \# \ \text{hs}_2) = \langle \text{homs} \ \text{hs}_1, \ x, \ \text{homs} \ \text{hs}_2 \rangle
\]

\[
\text{hom} :: \ 'a \ \text{heap} \ \rightarrow \ 'a \ \text{tree}
\]
\[
\text{hom} \ \text{Empty} = \langle \rangle
\]
\[
\text{hom} \ (\text{Hp} \ x \ \text{hs}) = \langle \text{homs} \ \text{hs}, \ x, \ \langle \rangle \rangle
\]

Potential function same as for splay trees
Verified:

The functions $insert$, $del_{\text{min}}$ and $merge$ all have $O(\log_2 n)$ amortized complexity.

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

\begin{align*}
insert \in O(1), & \quad del_{\text{min}} \in O(\log_2 n), & \quad merge \in O(1)
\end{align*}

The exact complexity is still open.
The inventors of the pairing heap:

The functional version: