Root-Balanced Tree

Tobias Nipkow

March 17, 2025

Abstract

Andersson [1, 2] introduced *general balanced trees*, search trees based on the design principle of partial rebuilding: perform update operations naively until the tree becomes too unbalanced, at which point a whole subtree is rebalanced. This article defines and analyzes a functional version of general balanced trees, which we call *root-balanced trees*. Using a lightweight model of execution time, amortized logarithmic complexity is verified in the theorem prover Isabelle.

This is the Isabelle formalization of the material decribed in the APLAS 2017 article *Verified Root-Balanced Trees* by the same author [3] which also presents experimental results that show competitiveness of root-balanced with AVL and red-black trees.

1 Time Monad

theory *Time-Monad*
imports
Main
HOL−Library.Monad-Syntax
begin

datatype $'a\ tm = TM\ (val{:}\ 'a)\ nat$

fun $val :: 'a\ tm \Rightarrow 'a$ **where**
$val\ (TM\ v\ n) = v$

fun $time :: 'a\ tm \Rightarrow nat$ **where**
$time\ (TM\ v\ n) = n$

definition $bind\text{-}tm :: 'a\ tm \Rightarrow ('a \Rightarrow 'b\ tm) \Rightarrow 'b\ tm$ **where**
$bind\text{-}tm\ s\ f = (case\ s\ of\ TM\ u\ m \Rightarrow case\ f\ u\ of\ TM\ v\ n \Rightarrow TM\ v\ (m{+}n))$

adhoc-overloading $Monad\text{-}Syntax.bind \rightleftharpoons bind\text{-}tm$

definition $tick\ v = TM\ v\ 1$

definition $return\ v = TM\ v\ 0$

abbreviation *eqtick* :: *'a tm* ⇒ *'a tm* ⇒ *bool* (**infix** ‹=1› *50*) **where**
eqtick l r ≡ (*l* = (*r* ⤜ *tick*))

translations *CONST eqtick l r* <= (*l* = (*bind-tm r CONST tick*))

lemmas *tm-simps* = *bind-tm-def return-def tick-def*

lemma *time-return*[*simp*]: *time* (*return x*) = *0*
⟨*proof*⟩

lemma *surj-TM*: *v* = *val tm* ⟹ *t* = *time tm* ⟹ *tm* = *TM v t*
⟨*proof*⟩

The following lemmas push *Time-Monad.val* into a monadic term:

lemma *val-return*[*simp*]: *val* (*return x*) = *x*
⟨*proof*⟩

lemma *val-bind-tm*[*simp*]: *val* (*bind-tm m f*) = (*let x* = *val m in val*(*f x*))
⟨*proof*⟩

lemma *val-tick*[*simp*]: *val* (*tick x*) = *x*
⟨*proof*⟩

lemma *val-let*: *val* (*let x* = *t in f*(*x*)) = (*let x* = *t in val*(*f x*))
⟨*proof*⟩

lemma *let-id*: (*let x* = *t in x*) = *t*
⟨*proof*⟩

lemmas *val-simps* =
val-return
val-bind-tm
val-tick
val-let
let-id
if-distrib[*of val*]
prod.case-distrib[*of val*]

lemmas *val-cong* = *arg-cong*[**where** *f*=*val*]

The following congruence rule enables termination proofs for recursive functions using this monad.

lemma *bind-tm-cong*[*fundef-cong*]:
assumes *m1* = *m2*
assumes *f1* (*val m1*) = *f2* (*val m2*)
shows *m1* ⤜ *f1* = *m2* ⤜ *f2*
⟨*proof*⟩

hide-const *TM*

end

2 Root Balanced Tree

theory *Root-Balanced-Tree*
imports
Amortized-Complexity.Amortized-Framework0
HOL−Library.Tree-Multiset
HOL−Data-Structures.Tree-Set
HOL−Data-Structures.Balance
Time-Monad
begin

declare *Let-def*[*simp*]

2.1 Time Prelude

Redefinition of some auxiliary functions, but now with *tm* monad:

2.1.1 *size-tree*

fun *size-tree-tm* :: *'a tree* ⇒ *nat tm* **where**
size-tree-tm ⟨⟩ =*1 return 0* |
size-tree-tm ⟨*l*, *x*, *r*⟩ =*1*
do { *m* ← *size-tree-tm l*;
n ← *size-tree-tm r*;
return (*m*+*n*+*1*)}

definition *size-tree* :: *'a tree* ⇒ *nat* **where**
size-tree t = *val*(*size-tree-tm t*)

lemma *size-tree-Leaf*[*simp*,*code*]: *size-tree* ⟨⟩ = *0*
⟨*proof*⟩

lemma *size-tree-Node*[*simp*,*code*]:
size-tree ⟨*l*, *x*, *r*⟩ =
(*let m* = *size-tree l*;
n = *size-tree r*
in m+*n*+*1*)
⟨*proof*⟩

lemma *size-tree*: *size-tree t* = *size t*
⟨*proof*⟩

definition *T-size-tree* :: *'a tree* ⇒ *nat* **where**
T-size-tree t = *time*(*size-tree-tm t*)

lemma *T-size-tree-Leaf*: *T-size-tree* ⟨⟩ = *1*
⟨*proof*⟩

lemma *T-size-tree-Node*:
T-size-tree ⟨*l*, *x*, *r*⟩ = *T-size-tree l* + *T-size-tree r* + *1*
⟨*proof*⟩

lemma *T-size-tree*: *T-size-tree t* = *2* ∗ *size t* + *1*
⟨*proof*⟩

2.1.2 *inorder*

fun *inorder2-tm* :: *'a tree* ⇒ *'a list* ⇒ *'a list tm* **where**
inorder2-tm ⟨⟩ *xs* =*1 return xs* |
inorder2-tm ⟨*l*, *x*, *r*⟩ *xs* =*1*
do { *rs* ← *inorder2-tm r xs*; *inorder2-tm l* (*x*#*rs*) }

definition *inorder2* :: *'a tree* ⇒ *'a list* ⇒ *'a list* **where**
inorder2 t xs = *val*(*inorder2-tm t xs*)

lemma *inorder2-Leaf*[*simp*,*code*]: *inorder2* ⟨⟩ *xs* = *xs*
⟨*proof*⟩

lemma *inorder2-Node*[*simp*,*code*]:
inorder2 ⟨*l*, *x*, *r*⟩ *xs* = (*let rs* = *inorder2 r xs in inorder2 l* (*x* # *rs*))
⟨*proof*⟩

lemma *inorder2*: *inorder2 t xs* = *Tree.inorder2 t xs*
⟨*proof*⟩

definition *T-inorder2* :: *'a tree* ⇒ *'a list* ⇒ *nat* **where**
T-inorder2 t xs = *time*(*inorder2-tm t xs*)

lemma *T-inorder2-Leaf*: *T-inorder2* ⟨⟩ *xs* = *1*
⟨*proof*⟩

lemma *T-inorder2-Node*:
T-inorder2 ⟨*l*, *x*, *r*⟩ *xs* = *T-inorder2 r xs* + *T-inorder2 l* (*x* # *inorder2 r xs*) +
1
⟨*proof*⟩

lemma *T-inorder2*: *T-inorder2 t xs* = *2*∗*size t* + *1*
⟨*proof*⟩

2.1.3 *split-min*

fun *split-min-tm* :: *'a tree* ⇒ (*'a* ∗ *'a tree*) *tm* **where**
split-min-tm Leaf =*1 return undefined* |
split-min-tm (*Node l x r*) =*1*

(*if* l = *Leaf then return* (x,r)
else do { (y,l') ← *split-min-tm* l; *return* (y, *Node* l' x r)})

definition *split-min* :: $'a$ *tree* ⇒ ($'a$ ∗ $'a$ *tree*) **where**
split-min t = *val* (*split-min-tm* t)

lemma *split-min-Node*[*simp*,*code*]:
split-min (*Node* l x r) =
(*if* l = *Leaf then* (x,r)
else let (y,l') = *split-min* l *in* (y, *Node* l' x r))
⟨*proof*⟩

definition *T-split-min* :: $'a$ *tree* ⇒ *nat* **where**
T-split-min t = *time* (*split-min-tm* t)

lemma *T-split-min-Node*[*simp*]:
T-split-min (*Node* l x r) = (*if* l = *Leaf then* 1 *else T-split-min* l + 1)
⟨*proof*⟩

lemma *split-minD*:
split-min t = (x,t') ⟹ t ≠ *Leaf* ⟹ x # *inorder* t' = *inorder* t
⟨*proof*⟩

2.1.4 Balancing

fun *bal-tm* :: *nat* ⇒ $'a$ *list* ⇒ ($'a$ *tree* ∗ $'a$ *list*) *tm* **where**
bal-tm n *xs* =1
(*if* n=0 *then return* (*Leaf*,*xs*) *else*
(*let* m = n *div* 2
in do { (l, *ys*) ← *bal-tm* m *xs*;
(r, *zs*) ← *bal-tm* (n−1−m) (*tl ys*);
return (*Node* l (*hd ys*) r, *zs*)}))

declare *bal-tm.simps*[*simp del*]

lemma *bal-tm-simps*:
bal-tm 0 *xs* =1 *return*(*Leaf*, *xs*)
n > 0 ⟹
bal-tm n *xs* =1
(*let* m = n *div* 2
in do { (l, *ys*) ← *bal-tm* m *xs*;
(r, *zs*) ← *bal-tm* (n−1−m) (*tl ys*);
return (*Node* l (*hd ys*) r, *zs*)})
⟨*proof*⟩

definition *bal* :: *nat* ⇒ $'a$ *list* ⇒ ($'a$ *tree* ∗ $'a$ *list*) **where**
bal n *xs* = *val* (*bal-tm* n *xs*)

lemma *bal-def2*[*code*]:

```
bal n xs =
  (if n=0 then (Leaf,xs) else
  (let m = n div 2;
       (l, ys) = bal m xs;
       (r, zs) = bal (n−1−m) (tl ys)
   in (Node l (hd ys) r, zs)))
⟨proof⟩

lemma bal-simps:
  bal 0 xs = (Leaf, xs)
  n > 0 ⟹
   bal n xs =
  (let m = n div 2;
      (l, ys) = bal m xs;
      (r, zs) = bal (n−1−m) (tl ys)
  in (Node l (hd ys) r, zs))
⟨proof⟩

lemma bal-eq: bal n xs = Balance.bal n xs
⟨proof⟩


definition T-bal :: nat ⇒ 'a list ⇒ nat where
T-bal n xs = time (bal-tm n xs)

lemma T-bal: T-bal n xs = 2∗n+1
⟨proof⟩

definition bal-list-tm :: nat ⇒ 'a list ⇒ 'a tree tm where
bal-list-tm n xs = do { (t,-) ← bal-tm n xs; return t }

definition bal-list :: nat ⇒ 'a list ⇒ 'a tree where
bal-list n xs = val (bal-list-tm n xs)

lemma bal-list-def2[code]: bal-list n xs = (let (t,ys) = bal n xs in t)
⟨proof⟩

lemma bal-list: bal-list n xs = Balance.bal-list n xs
⟨proof⟩

definition bal-tree-tm :: nat ⇒ 'a tree ⇒ 'a tree tm where
bal-tree-tm n t =1 do { xs ← inorder2-tm t []; bal-list-tm n xs }

definition bal-tree :: nat ⇒ 'a tree ⇒ 'a tree where
bal-tree n t = val (bal-tree-tm n t)

lemma bal-tree-def2[code]:
  bal-tree n t = (let xs = inorder2 t [] in bal-list n xs)
⟨proof⟩
```

lemma *bal-tree*: *bal-tree n t = Balance.bal-tree n t*
⟨*proof*⟩

definition *T-bal-tree* :: *nat ⇒ 'a tree ⇒ nat* **where**
T-bal-tree n xs = time (*bal-tree-tm n xs*)

lemma *T-bal-tree*: *n = size xs ⟹ T-bal-tree n xs = 4∗n+3*
⟨*proof*⟩

2.2 Naive implementation (insert only)

fun *node* :: *bool ⇒ 'a tree ⇒ 'a ⇒ 'a tree ⇒ 'a tree* **where**
node twist s x t = (*if twist then Node t x s else Node s x t*)

datatype *'a up = Same | Bal 'a tree | Unbal 'a tree*

locale *RBTi1 =*
fixes *bal-i* :: *nat ⇒ nat ⇒ bool*
assumes *bal-i-balance*:
bal-i (*size t*) (*height* (*balance-tree* (*t*::*'a*::*linorder tree*)))
assumes *mono-bal-i*: ⟦ *bal-i n h*; *n ≤ n'*; *h' ≤ h* ⟧ *⟹ bal-i n' h'*
begin

2.2.1 Functions

definition *up* :: *'a ⇒ 'a tree ⇒ bool ⇒ 'a up ⇒ 'a up* **where**
up x sib twist u = (*case u of Same ⇒ Same |*
Bal t ⇒ Bal(*node twist t x sib*) |
Unbal t ⇒ let t' = *node twist t x sib*; *h'* = *height t'*; *n'* = *size t'*
in if bal-i n' h' then Unbal t'
else Bal(*balance-tree t'*))

declare *up-def*[*simp*]

fun *ins* :: *nat ⇒ nat ⇒ 'a*::*linorder ⇒ 'a tree ⇒ 'a up* **where**
ins n d x Leaf =
(*if bal-i* (*n+1*) (*d+1*) *then Bal* (*Node Leaf x Leaf*) *else Unbal* (*Node Leaf x Leaf*))
|
ins n d x (*Node l y r*) =
(*case cmp x y of*
LT ⇒ up y r False (*ins n* (*d+1*) *x l*) |
EQ ⇒ Same |
GT ⇒ up y l True (*ins n* (*d+1*) *x r*))

fun *insert* :: *'a*::*linorder ⇒ 'a tree ⇒ 'a tree* **where**
insert x t =
(*case ins* (*size t*) *0 x t of*
Same ⇒ t |
Bal t' ⇒ t')

2.2.2 Functional Correctness and Invariants

lemma *height-balance*: $[\![\neg \ bal\text{-}i\ (size\ t)\ h\ ]\!]$
$\Longrightarrow height\ (balance\text{-}tree\ (t::'a::linorder\ tree)) < h$
⟨*proof*⟩

lemma *mono-bal-i'*:
$[\![\ ASSUMPTION(bal\text{-}i\ n\ h);\ n \leq n';\ h' \leq h\ ]\!] \Longrightarrow bal\text{-}i\ n'\ h'$
⟨*proof*⟩

lemma *inorder-ins*: $sorted(inorder\ t) \Longrightarrow$
$(ins\ n\ d\ x\ t = Same \longrightarrow ins\text{-}list\ x\ (inorder\ t) = inorder\ t) \wedge$
$(ins\ n\ d\ x\ t = Bal\ t' \longrightarrow ins\text{-}list\ x\ (inorder\ t) = inorder\ t') \wedge$
$(ins\ n\ d\ x\ t = Unbal\ t' \longrightarrow ins\text{-}list\ x\ (inorder\ t) = inorder\ t')$
⟨*proof*⟩

lemma *ins-size*:
shows $ins\ n\ d\ x\ t = Bal\ t' \Longrightarrow size\ t' = size\ t + 1$
and $ins\ n\ d\ x\ t = Unbal\ t' \Longrightarrow size\ t' = size\ t + 1$
⟨*proof*⟩

lemma *ins-height*:
shows $ins\ n\ d\ x\ t = Bal\ t' \Longrightarrow height\ t' \leq height\ t + 1$
and $ins\ n\ d\ x\ t = Unbal\ t' \Longrightarrow height\ t \leq height\ t' \wedge height\ t' \leq height\ t + 1$
⟨*proof*⟩

lemma *bal-i0*: *bal-i 0 0*
⟨*proof*⟩

lemma *bal-i1*: *bal-i 1 1*
⟨*proof*⟩

lemma *bal-i-ins-Unbal*:
assumes $ins\ n\ d\ x\ t = Unbal\ t'$ **shows** $bal\text{-}i\ (size\ t')\ (height\ t')$
⟨*proof*⟩

lemma *unbal-ins-Unbal*:
$ins\ n\ d\ x\ t = Unbal\ t' \Longrightarrow \neg\ bal\text{-}i\ (n+1)\ (height\ t' + d)$
⟨*proof*⟩

lemma *height-Unbal-l*: **assumes** $ins\ n\ (d+1)\ x\ l = Unbal\ l'$
$bal\text{-}i\ n\ (height\ \langle l, y, r\rangle + d)$
shows $height\ r < height\ l'$ (**is** *?P*)
⟨*proof*⟩
lemma *height-Unbal-r*: **assumes** $ins\ n\ (d+1)\ x\ r = Unbal\ r'$
$bal\text{-}i\ n\ (height\ \langle l, y, r\rangle + d)$
shows $height\ l < height\ r'$ (**is** *?P*)
⟨*proof*⟩

lemma *ins-bal-i-Bal*:

⟦ *ins n d x t* = *Bal t'*; *bal-i n* (*height t* + *d*) ⟧
⟹ *bal-i* (*n*+*1*) (*height t'* + *d*)
⟨*proof*⟩

lemma *ins0-neq-Unbal*: **assumes** *n* ≥ *size t* **shows** *ins n 0 a t* ≠ *Unbal t'*
⟨*proof*⟩

lemma *inorder-insert*: *sorted*(*inorder t*)
⟹ *inorder* (*insert x t*) = *ins-list x* (*inorder t*)
⟨*proof*⟩

lemma *bal-i-insert*: **assumes** *bal-i* (*size t*) (*height t*)
shows *bal-i* (*size*(*insert x t*)) (*height*(*insert x t*))
⟨*proof*⟩

end

This is just a test that (a simplified version of) the intended interpretation works (so far):

interpretation *Test*: *RBTi1* λ*n h*. *h* ≤ *log 2* (*real*(*n* + *1*)) + *1*
⟨*proof*⟩

2.3 Efficient Implementation (insert only)

fun *imbal* :: *'a tree* ⇒ *nat* **where**
imbal Leaf = *0* |
imbal (*Node l - r*) = *nat*(*abs*(*int*(*size l*) − *int*(*size r*))) − *1*

declare *imbal.simps* [*simp del*]

lemma *imbal0-if-wbalanced*: *wbalanced t* ⟹ *imbal t* = *0*
⟨*proof*⟩

The degree of imbalance allowed: how far from the perfect balance may the tree degenerate.

axiomatization *c* :: *real* **where**
c1: *c* > *1*

definition *bal-log* :: *'a tree* ⇒ *bool* **where**
bal-log t = (*height t* ≤ *ceiling*(*c* ∗ *log 2* (*size1 t*)))

fun *hchild* :: *'a tree* ⇒ *'a tree* **where**
hchild (*Node l - r*) = (*if height l* ≤ *height r then r else l*)

lemma *size1-imbal*:
assumes ¬ *bal-log t* **and** *bal-log* (*hchild t*) **and** *t* ≠ *Leaf*
shows *imbal t* > (*2 powr* (*1* − *1*/*c*) − *1*) ∗ *size1* (*t*) − *1*
⟨*proof*⟩

The following key lemma shows that *imbal* is a suitable potential because

it can pay for the linear-time cost of restructuring a tree that is not balanced but whose higher son is.

lemma *size1-imbal2*:
assumes ¬ *bal-log t* **and** *bal-log* (*hchild t*) **and** $t \neq Leaf$
shows *size1* (*t*) < (*2 powr* (*1*/*c*) / (*2* − *2 powr* (*1*/*c*))) ∗ (*imbal t* + *1*)
⟨*proof*⟩

datatype *'a up2* = *Same2* | *Bal2 'a tree* | *Unbal2 'a tree nat nat*

type-synonym *'a rbt1* = *'a tree* ∗ *nat*

An implementation where size and height are computed incrementally:

locale *RBTi2* = *RBTi1* +
fixes *e* :: *real*
assumes *e0*: *e* > *0*
assumes *imbal-size*:
⟦ ¬ *bal-i* (*size t*) (*height t*);
bal-i (*size*(*hchild t*)) (*height*(*hchild t*));
$t \neq Leaf$ ⟧
⟹ *e* ∗ (*imbal t* + *1*) ≥ *size1* (*t*::*'a*::*linorder tree*)
begin

2.3.1 Functions

definition *up2* :: *'a* ⇒ *'a tree* ⇒ *bool* ⇒ *'a up2* ⇒ *'a up2* **where**
up2 x sib twist u = (*case u of Same2* ⇒ *Same2* |
Bal2 t ⇒ *Bal2*(*node twist t x sib*) |
Unbal2 t n1 h1 ⇒
let n2 = *size sib*; *h2* = *height sib*;
t' = *node twist t x sib*;
n' = *n1*+*n2*+*1*; *h'* = *max h1 h2* + *1*
in if bal-i n' h' then Unbal2 t' n' h'
else Bal2(*bal-tree n' t'*))

declare *up2-def*[*simp*]

up2 traverses *sib* twice; unnecessarily, as it turns out:

definition *up3-tm* :: *'a* ⇒ *'a tree* ⇒ *bool* ⇒ *'a up2* ⇒ *'a up2 tm* **where**
up3-tm x sib twist u =*1* (*case u of*
Same2 ⇒ *return Same2* |
Bal2 t ⇒ *return* (*Bal2*(*node twist t x sib*)) |
Unbal2 t n1 h1 ⇒
do { *n2* ← *size-tree-tm sib*;
let t' = *node twist t x sib*;
n' = *n1*+*n2*+*1*;
h' = *h1* + *1*
in if bal-i n' h' then return (*Unbal2 t' n' h'*)
else do { *t''* ← *bal-tree-tm n' t'*;
return (*Bal2 t''*)}})

definition *up3* :: *'a* ⇒ *'a tree* ⇒ *bool* ⇒ *'a up2* ⇒ *'a up2* **where**
up3 a sib twist u = *val* (*up3-tm a sib twist u*)

lemma *up3-def2*[*simp*,*code*]:
up3 x sib twist u = (*case u of*
Same2 ⇒ *Same2* |
Bal2 t ⇒ *Bal2* (*node twist t x sib*) |
Unbal2 t n1 h1 ⇒
let n2 = *size-tree sib*; *t'* = *node twist t x sib*; *n'* = *n1* + *n2* + *1*; *h'* = *h1* + *1*
in if bal-i n' h' then Unbal2 t' n' h'
else let t'' = *bal-tree n' t' in Bal2 t''*)
⟨*proof*⟩

definition *T-up3* :: *'a* ⇒ *'a tree* ⇒ *bool* ⇒ *'a up2* ⇒ *nat* **where**
T-up3 x sib twist u = *time* (*up3-tm x sib twist u*)

lemma *T-up3-def2*[*simp*]: *T-up3 x sib twist u* =
(*case u of Same2* ⇒ *1* |
Bal2 t ⇒ *1* |
Unbal2 t n1 h1 ⇒
let n2 = *size sib*; *t'* = *node twist t x sib*; *h'* = *h1* + *1*; *n'* = *n1*+*n2*+*1*
in 2 ∗ *size sib* + *1* + (*if bal-i n' h' then 1 else T-bal-tree n' t'* + *1*))
⟨*proof*⟩

fun *ins2* :: *nat* ⇒ *nat* ⇒ *'a*::*linorder* ⇒ *'a tree* ⇒ *'a up2* **where**
ins2 n d x Leaf =
(*if bal-i* (*n*+*1*) (*d*+*1*) *then Bal2* (*Node Leaf x Leaf*) *else Unbal2* (*Node Leaf x Leaf*) *1 1*) |
ins2 n d x (*Node l y r*) =
(*case cmp x y of*
LT ⇒ *up2 y r False* (*ins2 n* (*d*+*1*) *x l*) |
EQ ⇒ *Same2* |
GT ⇒ *up2 y l True* (*ins2 n* (*d*+*1*) *x r*))

Definition of timed final insertion function:

fun *ins3-tm* :: *nat* ⇒ *nat* ⇒ *'a*::*linorder* ⇒ *'a tree* ⇒ *'a up2 tm* **where**
ins3-tm n d x Leaf =*1*
(*if bal-i* (*n*+*1*) (*d*+*1*) *then return*(*Bal2* (*Node Leaf x Leaf*))
else return(*Unbal2* (*Node Leaf x Leaf*) *1 1*)) |
ins3-tm n d x (*Node l y r*) =*1*
(*case cmp x y of*
LT ⇒ *do* {*l'* ← *ins3-tm n* (*d*+*1*) *x l*; *up3-tm y r False l'*} |
EQ ⇒ *return Same2* |
GT ⇒ *do* {*r'* ← *ins3-tm n* (*d*+*1*) *x r*; *up3-tm y l True r'*})

definition *ins3* :: *nat* ⇒ *nat* ⇒ *'a*::*linorder* ⇒ *'a tree* ⇒ *'a up2* **where**
ins3 n d x t = *val*(*ins3-tm n d x t*)

lemma *ins3-Leaf*[*simp*,*code*]:
ins3 n d x Leaf =
(*if bal-i* (*n*+1) (*d*+1) *then Bal2* (*Node Leaf x Leaf*) *else Unbal2* (*Node Leaf x Leaf*) *1 1*)
⟨*proof*⟩

lemma *ins3-Node*[*simp*,*code*]:
ins3 n d x (*Node l y r*) =
(*case cmp x y of*
LT ⇒ *let l'* = *ins3 n* (*d*+1) *x l in up3 y r False l'* |
EQ ⇒ *Same2* |
GT ⇒ *let r'* = *ins3 n* (*d*+1) *x r in up3 y l True r'*)
⟨*proof*⟩

definition *T-ins3* :: *nat* ⇒ *nat* ⇒ *'a*::*linorder* ⇒ *'a tree* ⇒ *nat* **where**
T-ins3 n d x t = *time*(*ins3-tm n d x t*)

lemma *T-ins3-Leaf*[*simp*]: *T-ins3 n d x Leaf* = *1*
⟨*proof*⟩

lemma *T-ins3-Node*[*simp*]: *T-ins3 n d x* (*Node l y r*) =
(*case cmp x y of*
LT ⇒ *T-ins3 n* (*d*+1) *x l* + *T-up3 y r False* (*ins3 n* (*d*+1) *x l*) |
EQ ⇒ *0* |
GT ⇒ *T-ins3 n* (*d*+1) *x r* + *T-up3 y l True* (*ins3 n* (*d*+1) *x r*)) + *1*
⟨*proof*⟩

fun *insert2* :: *'a*::*linorder* ⇒ *'a rbt1* ⇒ *'a rbt1* **where**
insert2 x (*t*,*n*) =
(*case ins2 n 0 x t of*
Same2 ⇒ (*t*,*n*) |
Bal2 t' ⇒ (*t'*,*n*+*1*))

fun *insert3-tm* :: *'a*::*linorder* ⇒ *'a rbt1* ⇒ *'a rbt1 tm* **where**
insert3-tm x (*t*,*n*) =*1*
(*do* { *u* ← *ins3-tm n 0 x t*;
case u of
Same2 ⇒ *return* (*t*,*n*) |
Bal2 t' ⇒ *return* (*t'*,*n*+*1*) |
Unbal2 - - - ⇒ *return undefined* })

definition *insert3* :: *'a*::*linorder* ⇒ *'a rbt1* ⇒ *'a rbt1* **where**
insert3 a t = *val* (*insert3-tm a t*)

lemma *insert3-def2*[*simp*]: *insert3 x* (*t*,*n*) =
(*let t'* = *ins3 n 0 x t in*

case t′ of
Same2 ⇒ (*t*,*n*) |
Bal2 t′ ⇒ (*t′*,*n*+*1*))
⟨*proof*⟩

definition *T-insert3* :: *′a*::*linorder* ⇒ *′a rbt1* ⇒ *nat* **where**
T-insert3 a t = *time* (*insert3-tm a t*)

lemma *T-insert3-def2*: *T-insert3 x* (*t*,*n*) = *T-ins3 n 0 x t* + *1*
⟨*proof*⟩

2.3.2 Equivalence Proofs

lemma *ins-ins2*:
shows *ins2 n d x t* = *Same2* ⟹ *ins n d x t* = *Same*
and *ins2 n d x t* = *Bal2 t′* ⟹ *ins n d x t* = *Bal t′*
and *ins2 n d x t* = *Unbal2 t′ n′ h′*
⟹ *ins n d x t* = *Unbal t′* ∧ *n′* = *size t′* ∧ *h′* = *height t′*
⟨*proof*⟩

lemma *ins2-ins*:
shows *ins n d x t* = *Same* ⟹ *ins2 n d x t* = *Same2*
and *ins n d x t* = *Bal t′* ⟹ *ins2 n d x t* = *Bal2 t′*
and *ins n d x t* = *Unbal t′*
⟹ *ins2 n d x t* = *Unbal2 t′* (*size t′*) (*height t′*)
⟨*proof*⟩

corollary *ins2-iff-ins*:
shows *ins2 n d x t* = *Same2* ⟷ *ins n d x t* = *Same*
and *ins2 n d x t* = *Bal2 t′* ⟷ *ins n d x t* = *Bal t′*
and *ins2 n d x t* = *Unbal2 t′ n′ h′* ⟷
ins n d x t = *Unbal t′* ∧ *n′* = *size t′* ∧ *h′* = *height t′*
⟨*proof*⟩

lemma *ins3-ins2*:
bal-i n (*height t* + *d*) ⟹ *ins3 n d x t* = *ins2 n d x t*
⟨*proof*⟩

lemma *insert2-insert*:
insert2 x (*t*,*size t*) = (*t′*,*n′*) ⟷ *t′* = *insert x t* ∧ *n′* = *size t′*
⟨*proof*⟩

lemma *insert3-insert2*:
bal-i n (*height t*) ⟹ *insert3 x* (*t*,*n*) = *insert2 x* (*t*,*n*)
⟨*proof*⟩

2.3.3 Amortized Complexity

fun Φ :: *′a tree* ⇒ *real* **where**
Φ *Leaf* = *0* |

$\Phi\ (Node\ l\ x\ r) = 6 * e * imbal\ (Node\ l\ x\ r) + \Phi\ l + \Phi\ r$

lemma Φ-*nn*: $\Phi\ t \geq 0$
⟨*proof*⟩

lemma Φ-*sum-mset*: $\Phi\ t = (\sum s \in\# subtrees\text{-}mset\ t.\ 6{*}e{*}imbal\ s)$
⟨*proof*⟩

lemma Φ-*wbalanced*: **assumes** *wbalanced t* **shows** $\Phi\ t = 0$
⟨*proof*⟩

lemma *imbal-ins-Bal*: $ins\ n\ d\ x\ t = Bal\ t' \Longrightarrow$
$real(imbal\ (node\ tw\ t'\ y\ s)) - imbal\ (node\ tw\ t\ y\ s) \leq 1$
⟨*proof*⟩

lemma *imbal-ins-Unbal*: $ins\ n\ d\ x\ t = Unbal\ t' \Longrightarrow$
$real(imbal\ (node\ tw\ t'\ y\ s)) - imbal\ (node\ tw\ t\ y\ s) \leq 1$
⟨*proof*⟩

lemma *T-ins3-Same*:
$ins3\ n\ d\ x\ t = Same2 \Longrightarrow T\text{-}ins3\ n\ d\ x\ t \leq 2 * height\ t + 1$
⟨*proof*⟩

lemma *T-ins3-Unbal*:
$[\![\ ins3\ n\ d\ x\ t = Unbal2\ t'\ n'\ h';\ \ bal\text{-}i\ n\ (height\ t + d)\ ]\!] \Longrightarrow$
$T\text{-}ins3\ n\ d\ x\ t \leq 2 * size\ t + 1 + height\ t$
⟨*proof*⟩

lemma *Phi-diff-Unbal*:
$[\![\ ins3\ n\ d\ x\ t = Unbal2\ t'\ n'\ h';\ \ bal\text{-}i\ n\ (height\ t + d)\ ]\!] \Longrightarrow$
$\Phi\ t' - \Phi\ t \leq 6{*}e * height\ t$
⟨*proof*⟩

lemma *amor-Unbal*:
$[\![\ ins3\ n\ d\ x\ t = Unbal2\ t'\ n'\ h';\ \ bal\text{-}i\ n\ (height\ t + d)\ ]\!] \Longrightarrow$
$T\text{-}ins3\ n\ d\ x\ t + \Phi\ t' - \Phi\ t \leq 2{*}size1\ t + (6{*}e + 1) * height\ t$
⟨*proof*⟩

lemma *T-ins3-Bal*:
$[\![\ ins3\ n\ d\ x\ t = Bal2\ t';\ bal\text{-}i\ n\ (height\ t + d)\ ]\!]$
$\Longrightarrow T\text{-}ins3\ n\ d\ x\ t + \Phi\ t' - \Phi\ t \leq (6{*}e{+}2) * (height\ t + 1)$
⟨*proof*⟩

lemma *T-insert3-amor*: **assumes** $n = size\ t\ bal\text{-}i\ (size\ t)\ (height\ t)$
$insert3\ a\ (t,n) = (t',n')$
shows $T\text{-}insert3\ a\ (t,n) + \Phi\ t' - \Phi\ t \leq (6{*}e{+}2) * (height\ t + 1) + 1$
⟨*proof*⟩

end

The insert-only version is shown to have the desired logarithmic amortized complexity. First it is shown to be linear in the height of the tree.

locale *RBTi2-Amor* = *RBTi2*
begin

fun *nxt* :: *'a* ⇒ *'a rbt1* ⇒ *'a rbt1* **where**
nxt x tn = *insert3 x tn*

fun t_s :: *'a* ⇒ *'a rbt1* ⇒ *real* **where**
t_s *x tn* = *T-insert3 x tn*

interpretation *I-RBTi2-Amor*: *Amortized*
where *init* = (*Leaf*,*0*)
and *nxt* = *nxt*
and *inv* = λ(*t*,*n*). *n* = *size t* ∧ *bal-i* (*size t*) (*height t*)
and *T* = t_s **and** Φ = λ(*t*,*n*). Φ *t*
and *U* = λ*x* (*t*,-). (*6*∗*e*+*2*) ∗ (*height t* + *1*) + *1*
⟨*proof*⟩

end

Now it is shown that a certain instantiation of *bal-i* that guarantees logarithmic height satisfies the assumptions of locale *RBTi2*.

interpretation *I-RBTi2*: *RBTi2*
where *bal-i* = λ*n h*. *h* ≤ *ceiling*(*c* ∗ *log 2* (*n*+*1*))
and *e* = *2 powr* (*1*/*c*) / (*2* − *2 powr* (*1*/*c*))
⟨*proof*⟩

2.4 Naive implementation (with delete)

axiomatization *cd* :: *real* **where**
cd0: *cd* > *0*

definition *bal-d* :: *nat* ⇒ *nat* ⇒ *bool* **where**
bal-d n dl = (*dl* < *cd*∗(*n*+*1*))

lemma *bal-d0*: *bal-d n 0*
⟨*proof*⟩

lemma *mono-bal-d*: ⟦ *bal-d n dl*; *n* ≤ *n'* ⟧ ⟹ *bal-d n' dl*
⟨*proof*⟩

locale *RBTid1* = *RBTi1*
begin

2.4.1 Functions

fun *insert-d* :: *'a*::*linorder* ⇒ *'a rbt1* ⇒ *'a rbt1* **where**
insert-d x (*t*,*dl*) =

```
  (case ins (size t + dl) 0 x t of
    Same ⇒ t |
    Bal t' ⇒ t', dl)


definition up-d :: 'a ⇒ 'a tree ⇒ bool ⇒ 'a tree option ⇒ 'a tree option where
up-d x sib twist u =
  (case u of
     None ⇒ None |
     Some t ⇒ Some(node twist t x sib))

declare up-d-def[simp]

fun del-tm :: 'a::linorder ⇒ 'a tree ⇒ 'a tree option tm where
del-tm x Leaf =1 return None |
del-tm x (Node l y r) =1
  (case cmp x y of
     LT ⇒ do { l' ← del-tm x l; return (up-d y r False l')} |
     EQ ⇒ if r = Leaf then return (Some l)
          else do { (a',r') ← split-min-tm r;
                    return (Some(Node l a' r'))} |
     GT ⇒ do { r' ← del-tm x r; return (up-d y l True r')})

definition del :: 'a::linorder ⇒ 'a tree ⇒ 'a tree option where
del x t = val(del-tm x t)

lemma del-Leaf[simp]: del x Leaf = None
⟨proof⟩

lemma del-Node[simp]: del x (Node l y r) =
  (case cmp x y of
     LT ⇒ let l' = del x l in up-d y r False l' |
     EQ ⇒ if r = Leaf then Some l
          else let (a',r') = split-min r in Some(Node l a' r') |
     GT ⇒ let r' = del x r in up-d y l True r')
⟨proof⟩

definition T-del :: 'a::linorder ⇒ 'a tree ⇒ nat where
T-del x t = time(del-tm x t)

lemma T-del-Leaf[simp]: T-del x Leaf = 1
⟨proof⟩

lemma T-del-Node[simp]: T-del x (Node l y r) =
  (case cmp x y of
     LT ⇒ T-del x l + 1 |
     EQ ⇒ if r = Leaf then 1 else T-split-min r + 1 |
     GT ⇒ T-del x r + 1)
⟨proof⟩
```

fun *delete* :: *'a::linorder* ⇒ *'a rbt1* ⇒ *'a rbt1* **where**
delete x (t,dl) =
(*case del x t of*
None ⇒ *(t,dl)* |
Some t' ⇒
if bal-d (size t') (dl+1) then (t',dl+1) else (balance-tree t', 0))

declare *delete.simps* [*simp del*]

2.4.2 Functional Correctness

lemma *size-insert-d*: *insert-d x (t,dl)* = *(t',dl')* ⟹ *size t* ≤ *size t'*
⟨*proof*⟩

lemma *inorder-insert-d*: *insert-d x (t,dl)* = *(t',dl')* ⟹ *sorted(inorder t)*
⟹ *inorder t'* = *ins-list x (inorder t)*
⟨*proof*⟩

lemma *bal-i-insert-d*: **assumes** *insert-d x (t,dl)* = *(t',dl') bal-i (size t + dl) (height t)*
shows *bal-i (size t' + dl) (height t')*
⟨*proof*⟩

lemma *inorder-del*:
sorted(inorder t) ⟹
inorder(case del x t of None ⇒ *t* | *Some t'* ⇒ *t')* = *del-list x (inorder t)*
⟨*proof*⟩

lemma *inorder-delete*:
⟦ *delete x (t,dl)* = *(t',dl')*; *sorted(inorder t)* ⟧ ⟹
inorder t' = *del-list x (inorder t)*
⟨*proof*⟩

lemma *size-split-min*:
⟦ *split-min t* = *(a,t')*; *t* ≠ *Leaf* ⟧ ⟹ *size t'* = *size t* − *1*
⟨*proof*⟩

lemma *height-split-min*:
⟦ *split-min t* = *(a,t')*; *t* ≠ *Leaf* ⟧ ⟹ *height t'* ≤ *height t*
⟨*proof*⟩

lemma *size-del*: *del x t* = *Some t'* ⟹ *size t'* = *size t* − *1*
⟨*proof*⟩

lemma *height-del*: *del x t* = *Some t'* ⟹ *height t'* ≤ *height t*
⟨*proof*⟩

lemma *bal-i-delete*:

assumes *bal-i (size t + dl) (height t) delete x (t,dl) = (t′,dl′)*
shows *bal-i (size t′ + dl′) (height t′)*
⟨*proof*⟩

lemma *bal-d-delete*:
⟦ *bal-d (size t) dl*; *delete x (t,dl) = (t′,dl′)* ⟧
⟹ *bal-d (size t′) dl′*
⟨*proof*⟩

Full functional correctness of the naive implementation:

interpretation *Set-by-Ordered*
where *empty = (Leaf,0)* **and** *isin = λ(t,n). isin t*
and *insert = insert-d* **and** *delete = delete*
and *inorder = λ(t,n). inorder t* **and** *inv = λ-. True*
⟨*proof*⟩

end

interpretation *I-RBTid1*: *RBTid1*
where *bal-i = λn h. h ≤ log 2 (real(n + 1)) + 1* ⟨*proof*⟩

2.5 Efficient Implementation (with delete)

type-synonym *'a rbt2 = 'a tree ∗ nat ∗ nat*

locale *RBTid2 = RBTi2 + RBTid1*
begin

2.5.1 Functions

fun *insert2-d* :: *'a*::*linorder* ⇒ *'a rbt2* ⇒ *'a rbt2* **where**
insert2-d x (t,n,dl) =
(*case ins2 (n+dl) 0 x t of*
Same2 ⇒ *(t,n,dl)* |
Bal2 t′ ⇒ *(t′,n+1,dl)*)

fun *insert3-d-tm* :: *'a*::*linorder* ⇒ *'a rbt2* ⇒ *'a rbt2 tm* **where**
insert3-d-tm x (t,n,dl) =1
do { *t′* ← *ins3-tm (n+dl) 0 x t*;
case t′ of
Same2 ⇒ *return (t,n,dl)* |
Bal2 t′ ⇒ *return (t′,n+1,dl)* |
Unbal2 - - - ⇒ *return undefined*}

definition *insert3-d* :: *'a*::*linorder* ⇒ *'a rbt2* ⇒ *'a rbt2* **where**
insert3-d a t = val (insert3-d-tm a t)

lemma *insert3-d-def2*[*simp*,*code*]: *insert3-d x (t,n,dl) =*

```
  (let t' = ins3 (n+dl) 0 x t in
   case t' of
     Same2 ⇒ (t,n,dl) |
     Bal2 t' ⇒ (t',n+1,dl) |
     Unbal2 - - - ⇒ undefined)
⟨proof⟩

definition T-insert3-d :: 'a::linorder ⇒ 'a rbt2 ⇒ nat where
T-insert3-d x t = time(insert3-d-tm x t)

lemma T-insert3-d-def2[simp]:
  T-insert3-d x (t,n,dl) = (T-ins3 (n+dl) 0 x t + 1)
⟨proof⟩

fun delete2-tm :: 'a::linorder ⇒ 'a rbt2 ⇒ 'a rbt2 tm where
delete2-tm x (t,n,dl) =1
  do { t' ← del-tm x t;
       case t' of
         None ⇒ return (t,n,dl) |
         Some t' ⇒
           (let n' = n−1; dl' = dl + 1
            in if bal-d n' dl' then return (t',n',dl')
               else do { t'' ← bal-tree-tm n' t';
                         return (t'', n', 0)})}

definition delete2 :: 'a::linorder ⇒ 'a rbt2 ⇒ 'a rbt2 where
delete2 x t = val(delete2-tm x t)

lemma delete2-def2:
  delete2 x (t,n,dl) =
  (let t' = del x t in
   case t' of
    None ⇒ (t,n,dl) |
    Some t' ⇒ (let n' = n−1; dl' = dl + 1
      in if bal-d n' dl' then (t',n',dl')
         else let t'' = bal-tree n' t' in (t'', n', 0)))
⟨proof⟩

definition T-delete2 :: 'a::linorder ⇒ 'a rbt2 ⇒ nat where
T-delete2 x t = time(delete2-tm x t)

lemma T-delete2-def2:
  T-delete2 x (t,n,dl) = (T-del x t +
  (case del x t of
     None ⇒ 1 |
     Some t' ⇒ (let n' = n−1; dl' = dl + 1
       in if bal-d n' dl' then 1 else T-bal-tree n' t' + 1)))
⟨proof⟩
```

2.5.2 Equivalence proofs

lemma *insert2-insert-d*:
insert2-d x (t,size t,dl) = (t',n',dl') ⟷
(t',dl') = insert-d x (t,dl) ∧ n' = size t'
⟨*proof*⟩

lemma *insert3-insert2-d*:
bal-i (n+dl) (height t) ⟹ insert3-d x (t,n,dl) = insert2-d x (t,n,dl)
⟨*proof*⟩

lemma *delete2-delete*:
delete2 x (t,size t,dl) = (t',n',dl') ⟷
(t',dl') = delete x (t,dl) ∧ n' = size t'
⟨*proof*⟩

2.5.3 Amortized complexity

fun Φ_d :: *'a rbt2 ⇒ real* **where**
Φ_d *(t,n,dl) = Φ t + 4∗dl/cd*

lemma Φ_d*-case*: Φ_d *tndl = (case tndl of (t,n,dl) ⇒ Φ t + 4∗dl/cd)*
⟨*proof*⟩

lemma *imbal-diff-decr*:
size r' = size r − 1 ⟹
real(imbal (Node l x' r')) − imbal (Node l x r) ≤ 1
⟨*proof*⟩

lemma *tinsert-d-amor*:
assumes *n = size t insert-d a (t,dl) = (t',dl') bal-i (size t + dl) (height t)*
shows *T-insert3-d a (t,n,dl) + Φ t' − Φ t ≤ (6∗e+2) ∗ (height t + 1) + 1*
⟨*proof*⟩

lemma *T-split-min-ub*:
t ≠ Leaf ⟹ T-split-min t ≤ height t + 1
⟨*proof*⟩

lemma *T-del-ub*:
T-del x t ≤ height t + 1
⟨*proof*⟩

lemma *imbal-split-min*:
split-min t = (x,t') ⟹ t ≠ Leaf ⟹ real(imbal t') − imbal t ≤ 1
⟨*proof*⟩

lemma *imbal-del-Some*:
del x t = Some t' ⟹ real(imbal t') − imbal t ≤ 1
⟨*proof*⟩

lemma *Phi-diff-split-min*:
split-min t = (*x*, *t'*) ⟹ *t* ≠ *Leaf* ⟹ Φ *t'* − Φ *t* ≤ 6∗*e*∗*height t*
⟨*proof*⟩

lemma *Phi-diff-del-Some*:
del x t = *Some t'* ⟹ Φ *t'* − Φ *t* ≤ 6∗*e* ∗ *height t*
⟨*proof*⟩

lemma *amor-del-Some*:
del x t = *Some t'* ⟹
T-del x t + Φ *t'* − Φ *t* ≤ (6∗*e* + 1) ∗ *height t* + 1
⟨*proof*⟩

lemma *cd1*: 1/*cd* > 0
⟨*proof*⟩

lemma *T-delete-amor*: **assumes** *n* = *size t*
shows *T-delete2 x* (*t*,*n*,*dl*) + Φ$_d$ (*delete2 x* (*t*,*n*,*dl*)) − Φ$_d$ (*t*,*n*,*dl*)
≤ (6∗*e*+1) ∗ *height t* + 4/*cd* + 4
⟨*proof*⟩

datatype (*plugins del*: *lifting*) *'b ops* = *Insert 'b* | *Delete 'b*

fun *nxt* :: *'a ops* ⇒ *'a rbt2* ⇒ *'a rbt2* **where**
nxt (*Insert x*) *t* = *insert3-d x t* |
nxt (*Delete x*) *t* = *delete2 x t*

fun t_s :: *'a ops* ⇒ *'a rbt2* ⇒ *real* **where**
t_s (*Insert x*) *t* = *T-insert3-d x t* |
t_s (*Delete x*) *t* = *T-delete2 x t*

interpretation *RBTid2-Amor*: *Amortized*
where *init* = (*Leaf*,*0*,*0*)
and *nxt* = *nxt*
and *inv* = λ(*t*,*n*,*dl*). *n* = *size t* ∧
bal-i (*size t*+*dl*) (*height t*) ∧ *bal-d* (*size t*) *dl*
and *T* = t_s **and** Φ = Φ$_d$
and *U* = λ*f* (*t*,-). *case f of*
Insert - ⇒ (6∗*e*+2) ∗ (*height t* + 1) + 1 |
Delete - ⇒ (6∗*e*+1) ∗ *height t* + 4/*cd* + 4
⟨*proof*⟩

end

axiomatization *b* :: *real* **where**
b0: *b* > 0

axiomatization where
cd-le-log: $cd \leq 2\ powr\ (b/c) - 1$

This axiom is only used to prove that the height remains logarithmic in the size.

interpretation *I-RBTid2*: *RBTid2*
where $bal\text{-}i = \lambda n\ h.\ h \leq ceiling(c * log\ 2\ (n+1))$
and $e = 2\ powr\ (1/c)\ /\ (2 - 2\ powr\ (1/c))$
⟨*proof*⟩

Finally we show that under the above interpretation of *bal-i* the height is logarithmic:

definition *bal-i* :: *nat* ⇒ *nat* ⇒ *bool* **where**
$bal\text{-}i\ n\ h = (h \leq ceiling(c * log\ 2\ (n+1)))$

lemma assumes $bal\text{-}i\ (size\ t + dl)\ (height\ t)\ bal\text{-}d\ (size\ t)\ dl$
shows $height\ t \leq ceiling(c * log\ 2\ (size1\ t) + b)$
⟨*proof*⟩

end

3 Tabulating the Balanced Predicate

theory *Root-Balanced-Tree-Tab*
imports
Root-Balanced-Tree
HOL−Decision-Procs.Approximation
HOL−Library.IArray
begin

locale *Min-tab* =
fixes *p* :: *nat* ⇒ *nat* ⇒ *bool*
fixes *tab* :: *nat list*
assumes *mono-p*: $n \leq n' \Longrightarrow p\ n\ h \Longrightarrow p\ n'\ h$
assumes *p*: $\exists n.\ p\ n\ h$
assumes *tab-LEAST*: $h < length\ tab \Longrightarrow tab!h = (LEAST\ n.\ p\ n\ h)$
begin

lemma *tab-correct*: $h < length\ tab \Longrightarrow p\ n\ h = (n \geq tab\ !\ h)$
⟨*proof*⟩

end

definition *bal-tab* :: *nat list* **where**
bal-tab = [*0*, *1*, *1*, *2*, *4*, *6*, *10*, *16*, *25*, *40*, *64*, *101*, *161*, *256*, *406*, *645*, *1024*, *1625*, *2580*, *4096*, *6501*, *10321*, *16384*, *26007*, *41285*, *65536*, *104031*, *165140*, *262144*, *416127*, *660561*, *1048576*, *1664510*, *2642245*, *4194304*, *6658042*, *10568983*, *16777216*, *26632170*, *42275935*, *67108864*, *106528681*, *169103740*, *268435456*,

426114725, *676414963*, *1073741824*, *1704458900*, *2705659852*, *4294967296*, *6814897603*]

axiomatization where *c-def*: $c = 3/2$

fun *is-floor* :: *nat* ⇒ *nat* ⇒ *bool* **where**
is-floor n h = (*let m* = *floor*((2::*real*) *powr* ((*real*(*h*)−1)/*c*)) *in n* ≤ *m* ∧ *m* ≤ *n*)

Note that $n \leq m \wedge m \leq n$ avoids the technical restriction of the *approximation* method which does not support =, even on integers.

lemma *bal-tab-correct*:
∀ *i* < *length bal-tab*. *is-floor* (*bal-tab*!*i*) *i*
⟨*proof*⟩

lemma *ceiling-least-real*: *ceiling*(*r*::*real*) = (*LEAST i*. *r* ≤ *i*)
⟨*proof*⟩

lemma *floor-greatest-real*: *floor*(*r*::*real*) = (*GREATEST i*. *i* ≤ *r*)
⟨*proof*⟩

lemma *LEAST-eq-floor*:
(*LEAST n*. *int h* ≤ ⌈*c* ∗ *log* 2 (*real n* + 1)⌉) = *floor*((2::*real*) *powr* ((*real*(*h*)−1)/*c*))
⟨*proof*⟩

interpretation *Min-tab*
where *p* = *bal-i* **and** *tab* = *bal-tab*
⟨*proof*⟩

Now we replace the list by an immutable array:

definition *bal-array* :: *nat iarray* **where**
bal-array = *IArray bal-tab*

A trick for code generation: how to get rid of the precondition:

lemma *bal-i-code*:
bal-i n h =
(*if h* < *IArray.length bal-array then IArray.sub bal-array h* ≤ *n else bal-i n h*)
⟨*proof*⟩

end

References

[1] A. Andersson. Improving partial rebuilding by using simple balance criteria. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures (WADS '89)*, volume 382 of *LNCS*, pages 393–402. Springer, 1989.

[2] A. Andersson. General balanced trees. *J. Algorithms*, 30(1):1–18, 1999.

[3] T. Nipkow. Verified root-balanced trees. In B.-Y. E. Chang, editor, *Programming Languages and Systems, APLAS 2017*, volume ? of *LNCS*. Springer, 2017. http://www.in.tum.de/~nipkow/pubs/aplas17.html.