Root-Balanced Tree

Tobias Nipkow

December 14, 2021

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* ⇒ *'a* **where**
val (*TM v n*) = *v*

fun *time* :: *'a tm* ⇒ *nat* **where**
time (*TM v n*) = *n*

definition *bind-tm* :: *'a tm* ⇒ (*'a* ⇒ *'b tm*) ⇒ *'b tm* **where**
bind-tm s f = (*case s of TM u m* ⇒ *case f u of TM v n* ⇒ *TM v* (*m*+*n*))

adhoc-overloading *Monad-Syntax.bind bind-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*
by(*simp add*: *return-def*)

lemma *surj-TM*: *v* = *val tm* ⟹ *t* = *time tm* ⟹ *tm* = *TM v t*
by (*metis time.simps tm.exhaust val.simps*)

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

lemma *val-return*[*simp*]: *val* (*return x*) = *x*
by(*simp add*: *return-def*)

lemma *val-bind-tm*[*simp*]: *val* (*bind-tm m f*) = (*let x* = *val m in val*(*f x*))
by(*simp add*: *bind-tm-def split*: *tm.split*)

lemma *val-tick*[*simp*]: *val* (*tick x*) = *x*
by(*simp add*: *tick-def*)

lemma *val-let*: *val* (*let x* = *t in f*(*x*)) = (*let x* = *t in val*(*f x*))
by *simp*

lemma *let-id*: (*let x* = *t in x*) = *t*
by *simp*

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*]

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*
using *val-cong*[*OF size-tree-tm.simps*(*1*)]
by(*simp only*: *size-tree-def val-simps*)

lemma *size-tree-Node*[*simp*,*code*]:
size-tree ⟨*l*, *x*, *r*⟩ =
(*let m* = *size-tree l*;
n = *size-tree r*
in m+*n*+*1*)
using *val-cong*[*OF size-tree-tm.simps*(*2*)]
by(*simp only*: *size-tree-def val-simps*)

lemma *size-tree*: *size-tree t* = *size t*
by(*induction t rule*: *size-tree-tm.induct*)(*auto*)

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*
by(*simp add*: *T-size-tree-def tm-simps*)

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

by(*simp add*: *T-size-tree-def size-tree-def tm-simps split*: *tm.split*)

lemma *T-size-tree*: *T-size-tree t = 2 * size t + 1*
by(*induction t*)(*auto simp*: *T-size-tree-Leaf T-size-tree-Node*)

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*
using *val-cong*[*OF inorder2-tm.simps*(*1*)]
by(*simp only*: *inorder2-def val-simps*)

lemma *inorder2-Node*[*simp*,*code*]:
inorder2 ⟨l, x, r⟩ xs = (let rs = inorder2 r xs in inorder2 l (x # rs))
using *val-cong*[*OF inorder2-tm.simps*(*2*), *of l*]
by(*simp only*: *inorder2-def val-simps*)

lemma *inorder2*: *inorder2 t xs = Tree.inorder2 t xs*
by(*induction t xs rule*: *inorder2-tm.induct*)(*auto*)

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*
by(*simp add*: *T-inorder2-def tm-simps*)

lemma *T-inorder2-Node*:
T-inorder2 ⟨l, x, r⟩ xs = T-inorder2 r xs + T-inorder2 l (x # inorder2 r xs) + 1
by(*simp add*: *T-inorder2-def inorder2-def tm-simps split*: *tm.split*)

lemma *T-inorder2*: *T-inorder2 t xs = 2∗size t + 1*
by(*induction t arbitrary*: *xs*)(*auto simp*: *T-inorder2-Leaf T-inorder2-Node*)

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))
using *val-cong*[*OF split-min-tm.simps*(*2*)]
by(*simp only*: *split-min-def val-simps*)

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)
using *val-cong*[*OF split-min-tm.simps*(*2*)]
by(*simp add*: *T-split-min-def tm-simps split*: *tm.split*)

lemma *split-minD*:
split-min t = (x,t′) ⟹ t ≠ Leaf ⟹ x # inorder t′ = inorder t
by(*induction t arbitrary*: *t′ rule*: *split-min.induct*)
(auto simp: *sorted-lems split*: *prod.splits if-splits*)

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)})
by(*simp-all add*: *bal-tm.simps*)

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)))
using val-cong[OF bal-tm.simps(1)]
by(simp only: bal-def val-simps)

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))
by(simp-all add: bal-def2)

lemma bal-eq: bal n xs = Balance.bal n xs
apply(induction n xs rule: bal.induct)
apply(case-tac n=0)
 apply(simp add: bal-simps Balance.bal-simps)
apply(simp add: bal-simps Balance.bal-simps split: prod.split)
done


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
unfolding T-bal-def
apply(induction n xs rule: bal-tm.induct)
apply(case-tac n=0)
apply(simp add: bal-tm-simps)
  apply(auto simp add: bal-tm-simps tm-simps simp del: subst-all split: tm.split)
subgoal premises p for n xs t1 xs1
  using p(2)[OF refl,of xs1] p(3−) by(simp)
done

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)
using val-cong[OF bal-list-tm-def]
by(simp only: bal-list-def bal-def val-simps)
```

lemma *bal-list*: *bal-list n xs = Balance.bal-list n xs*
by(*auto simp add*: *bal-list-def2 Balance.bal-list-def bal-eq split*: *prod.split*)

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*)
using *val-cong*[*OF bal-tree-tm-def*, *of n t*]
by(*simp only*: *bal-tree-def bal-list-def inorder2-def val-simps*)

lemma *bal-tree*: *bal-tree n t = Balance.bal-tree n t*
by(*simp add*: *bal-tree-def2 Balance.bal-tree-def bal-list inorder2 inorder2-inorder*)

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*
by(*simp add*: *T-bal-tree-def bal-tree-tm-def tm-simps bal-list-tm-def*
surj-TM[*OF inorder2-def T-inorder2-def*] *T-inorder2*
surj-TM[*OF bal-def T-bal-def*] *T-bal size1-size*
split: *tm.split prod.split*)

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: ⟦ ¬ bal-i (size t) h ⟧
  ⟹ height (balance-tree (t::'a::linorder tree)) < h
by (meson bal-i-balance leI le-refl mono-bal-i)

lemma mono-bal-i':
  ⟦ ASSUMPTION(bal-i n h); n ≤ n'; h' ≤ h ⟧ ⟹ bal-i n' h'
unfolding ASSUMPTION-def by(rule mono-bal-i)

lemma inorder-ins: sorted(inorder t) ⟹
  (ins n d x t = Same ⟶ ins-list x (inorder t) = inorder t) ∧
  (ins n d x t = Bal t' ⟶ ins-list x (inorder t) = inorder t') ∧
  (ins n d x t = Unbal t' ⟶ ins-list x (inorder t) = inorder t')
by(induction t arbitrary: d t')
  (auto simp: ins-list-simps bal.simps[of Suc 0] bal.simps[of 0]
        split!: if-splits prod.splits up.splits)

lemma ins-size:
shows ins n d x t = Bal t' ⟹ size t' = size t + 1
and ins n d x t = Unbal t' ⟹ size t' = size t + 1
by(induction t arbitrary: d t')
  (auto split: if-splits up.splits)

lemma ins-height:
shows ins n d x t = Bal t' ⟹ height t' ≤ height t + 1
and ins n d x t = Unbal t' ⟹ height t ≤ height t' ∧ height t' ≤ height t + 1
proof(induction t arbitrary: d t')
  case Leaf
  { case 1 thus ?case by (auto split: if-splits)
  next
    case 2 thus ?case by (auto split: if-splits)
```

```
  }
next
  case (Node l y r)
  { case 1
    consider (ls) x < y | (eq) x = y | (gr) x > y by(metis less-linear)
    thus ?case
    proof cases
      case ls
      show ?thesis
      proof (cases ins n (d+1) x l)
        case Same thus ?thesis using 1 ls by (simp)
      next
        case Bal
        thus ?thesis
          using 1 ls by (auto simp: max-def dest: Node)
      next
        case (Unbal l')
        let ?t = Node l' y r let ?h = height ?t let ?n = size ?t
        have ¬ bal-i ?n ?h using 1 ls Unbal by (auto)
        thus ?thesis
          using 1 ls Unbal Node.IH(2)[OF Unbal]
            height-balance[of ?t height ?t]
          by(auto)
      qed
    next
      case eq
      thus ?thesis using 1 by(simp)
    next
      case gr
      show ?thesis
      proof (cases ins n (d+1) x r)
        case Same
        thus ?thesis using 1 gr by (simp)
      next
        case Bal
        thus ?thesis
          using 1 gr by (auto simp: max-def dest: Node)
      next
        case (Unbal r')
        let ?t = Node l y r' let ?h = height ?t let ?n = size ?t
        have ¬ bal-i ?n ?h using 1 gr Unbal by (auto)
        thus ?thesis
          using 1 gr Unbal Node.IH(4)[OF Unbal]
            height-balance[of ?t height ?t]
          by(auto)
      qed
    qed
  next
    case 2
```

```
    thus ?case
      by(auto simp: max-def dest: Node split: if-splits up.splits)
  }
qed

lemma bal-i0: bal-i 0 0
using bal-i-balance[of Leaf]
by(auto simp add: Balance.bal-tree-def balance-tree-def Balance.bal-list-def Balance.bal-simps)

lemma bal-i1: bal-i 1 1
using bal-i-balance[of Node Leaf undefined Leaf]
by(auto simp add: balance-tree-def Balance.bal-tree-def Balance.bal-list-def Balance.bal-simps)

lemma bal-i-ins-Unbal:
  assumes ins n d x t = Unbal t' shows bal-i (size t') (height t')
proof(cases t)
  case Leaf thus ?thesis
    using assms bal-i1 by(auto split: if-splits)
next
  case Node thus ?thesis
    using assms by(auto split: if-splits up.splits)
qed

lemma unbal-ins-Unbal:
  ins n d x t = Unbal t' ⟹ ¬ bal-i (n+1) (height t' + d)
proof(induction t arbitrary: d t')
  case Leaf thus ?case
    by (auto split: if-splits)
next
  case Node thus ?case
    by(fastforce simp: mono-bal-i' split: if-splits up.splits)
qed

lemma height-Unbal-l: assumes ins n (d+1) x l = Unbal l'
  bal-i n (height ⟨l, y, r⟩ + d)
shows height r < height l' (is ?P)
proof(rule ccontr)
  assume ¬ ?P
  thus False
    using assms(2) unbal-ins-Unbal[OF assms(1)]
    by (auto simp: mono-bal-i')
qed
lemma height-Unbal-r: assumes ins n (d+1) x r = Unbal r'
  bal-i n (height ⟨l, y, r⟩ + d)
shows height l < height r' (is ?P)
proof(rule ccontr)
  assume ¬ ?P
  thus False
    using assms(2) unbal-ins-Unbal[OF assms(1)]
```

```
    by (auto simp: mono-bal-i' split: if-splits)
qed

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(induction t arbitrary: d t')
  case Leaf
  thus ?case
    by (auto split: if-splits)
next
  case (Node l y r)
  consider (ls) x < y | (eq) x = y | (gr) x > y
    by(metis less-linear)
  thus ?case
  proof cases
    case ls
    have 2: bal-i n (height l + (d + 1))
      using Node.prems(2) by (simp add: mono-bal-i')
    show ?thesis
    proof (cases ins n (d+1) x l)
      case Same
      thus ?thesis
        using Node.prems ls by (simp)
    next
      case Bal
      thus ?thesis
        using Node.prems ls ins-height(1)[OF Bal] Node.IH(1)[OF Bal 2]
        by (auto simp: max-def mono-bal-i')
    next
      case (Unbal l')
      let ?t = Node l' y r let ?h = height ?t let ?n = size ?t
      have ¬ bal-i ?n ?h using Node.prems ls Unbal by (auto)
      thus ?thesis
        using Node.prems ls Unbal height-balance[of ?t height ?t]
          ins-height(2)[OF Unbal]
        by (auto simp: mono-bal-i')
    qed
  next
    case eq
    thus ?thesis
      using Node.prems by(simp)
  next
    case gr
    have 2: bal-i n (height r + (d + 1))
      using Node.prems(2) by(simp add: mono-bal-i')
    show ?thesis
    proof (cases ins n (d+1) x r)
      case Same
```

```
    thus ?thesis
      using Node.prems gr by (simp)
  next
    case Bal
    thus ?thesis
      using Node.prems gr ins-height(1)[OF Bal] Node.IH(2)[OF Bal 2]
      by (auto simp: max-def mono-bal-i')
  next
    case (Unbal r')
    let ?t = Node l y r' let ?h = height ?t let ?n = size ?t
    have ¬ bal-i ?n ?h using Node.prems gr Unbal by (auto)
    thus ?thesis
      using Node.prems gr Unbal
        height-balance[of ?t height ?t] ins-height(2)[OF Unbal]
      by (auto simp: mono-bal-i')
  qed
 qed
qed

lemma ins0-neq-Unbal: assumes n ≥ size t shows ins n 0 a t ≠ Unbal t'
proof(cases t)
 case Leaf thus ?thesis using bal-i1 by(simp add: numeral-eq-Suc mono-bal-i')
next
 case Node
 thus ?thesis
  using unbal-ins-Unbal[of n 0 a t t'] assms
  by(auto simp: ins-size mono-bal-i' split: up.splits)
qed

lemma inorder-insert: sorted(inorder t)
 ⟹ inorder (insert x t) = ins-list x (inorder t)
using ins0-neq-Unbal
by(auto simp add: insert-def inorder-ins split: prod.split up.split)

lemma bal-i-insert: assumes bal-i (size t) (height t)
shows bal-i (size(insert x t)) (height(insert x t))
proof (cases ins (size t) 0 x t)
 case Same
 with assms show ?thesis by simp
next
 case Bal
 thus ?thesis
  using ins-bal-i-Bal[OF Bal] assms ins-size by(simp add: size1-size)
next
 case (Unbal t')
 hence False using ins0-neq-Unbal by blast
 thus ?thesis ..
qed
```

end

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

interpretation *Test*: *RBTi1* $\lambda n\ h.\ h \leq log\ 2\ (real(n + 1)) + 1$
proof (*standard*, *goal-cases*)
  case (*1 t*)
  have ∗: $log\ 2\ (1 + real(size\ t)) \geq 0$ **by** (*simp*)
  show *?case* **apply**(*simp add*: *height-balance-tree*) **using** ∗ **by** *linarith*
next
  case (*2 n h n′ h′*)
  have $real\ h' \leq real\ h$ **by**(*simp add*: *2*)
  also have $\ldots \leq log\ 2\ (n+1) + 1$ **by**(*rule 2*)
  also have $\ldots \leq log\ 2\ (n'+1) + 1$ **using** *2*(*2*,*3*) **by**(*simp*)
  finally show *?case* **.**
qed

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*
by (*cases t*) (*auto simp add*: *imbal.simps*)

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 −
  obtain *l x r* **where** *t*: *t* = *Node l x r*
    using ‹*t* ≠ *Leaf*› **by**(*auto simp*: *neq-Leaf-iff*)
  let *?sh* = *hchild t*
  have ∗: *c* ∗ *log 2* (*size1 ?sh*) ≥ *0*
    using *c1* **apply**(*simp add*: *zero-le-mult-iff*)
    using *size1-ge0*[*of ?sh*] **by** *linarith*

```
  have (2 powr (1 − 1/c) − 1) ∗ size1 t − 1
    = 2 powr (1 − 1/c) ∗ size1 t − size1 t − 1
    by (simp add: ring-distribs)
  also have ... = 2 ∗ (2 powr (− 1/c) ∗ size1 t) − size1 t − 1
    using c1 by(simp add: powr-minus powr-add[symmetric] field-simps)
  also have 2 powr (− 1/c) ∗ size1 t < size1 ?sh
  proof −
    have ceiling(c ∗ log 2 (size1 t)) < ceiling (c ∗ log 2 (size1 ?sh)) + 1
    proof −
      have ceiling(c ∗ log 2 (size1 t)) < height t
        using assms(1) by (simp add: bal-log-def)
      also have ... = height(?sh) + 1 by(simp add: t max-def)
      finally show ?thesis
        using assms(2) unfolding bal-log-def by linarith
    qed
    hence c ∗ log 2 (size1 t) < c ∗ log 2 (size1 ?sh) + 1
      using ∗ by linarith
    hence log 2 (size1 t) − 1/c < log 2 (size1 ?sh)
      using c1 by(simp add: field-simps)
    from powr-less-mono[OF this, of 2] show ?thesis
      by (simp add: powr-diff powr-minus field-simps)
  qed
  also have 2 ∗ real(size1 ?sh) − size1 t − 1
        = real(size1 ?sh) − (real(size1 t) − size1 ?sh) − 1
    by (simp add: assms(1))
  also have ... ≤ imbal t
    by (auto simp add: t assms(1) imbal.simps size1-size)
  finally show ?thesis by simp
qed
```

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 ≠ Leaf
  shows size1 (t) < (2 powr (1/c) / (2 − 2 powr (1/c))) ∗ (imbal t + 1)
proof −
  have ∗: 2 powr (1 − 1 / c) − 1 > 0
    using c1 by(simp add: field-simps log-less-iff[symmetric])
  have (2 powr (1 − 1 / c) − 1) ∗ size1 t < imbal t + 1
    using size1-imbal[OF assms] by linarith
  hence size1 t < 1 / (2 powr (1 − 1 / c) − 1) ∗ (imbal t + 1)
    using ∗ by(simp add: field-simps)
  also have 1 / (2 powr (1 − 1 / c) − 1) = 2 powr (1/c) / (2 − 2 powr (1/c))
  proof −
    have 1 / (2 powr (1 − 1 / c) − 1) = 1 / (2 / 2 powr (1/c) − 1)
      by(simp add: powr-diff)
    also have ... = 2 powr (1/c) / (2 − 2 powr (1/c))
      by(simp add: field-simps)
```

finally show *?thesis* **.**
qed
finally show *?thesis* **.**
qed

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 ≠ *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''*)
using *val-cong*[*OF up3-tm-def*]
by(*simp only*: *up3-def size-tree-def bal-tree-def val-simps up2.case-distrib*[*of val*])

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*))
by(*simp add*: *T-up3-def up3-tm-def surj-TM*[*OF size-tree-def T-size-tree-def*]
size-tree T-size-tree T-bal-tree-def tm-simps split: *tm.split up2.split*)

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)
using val-cong[OF ins3-tm.simps(1)]
by(simp only: ins3-def val-simps cmp-val.case-distrib[of val])

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')
using val-cong[OF ins3-tm.simps(2)]
by(simp only: ins3-def up3-def val-simps cmp-val.case-distrib[of val])

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
by(simp add: tm-simps T-ins3-def)

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
apply(subst T-ins3-def)
apply(subst ins3-tm.simps)
apply(auto simp add: tm-simps surj-TM[OF ins3-def T-ins3-def] surj-TM[OF
up3-def T-up3-def]
          simp del: T-up3-def2 split: tm.splits up2.split)
done


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*))
using *val-cong*[*OF insert3-tm.simps*(*1*)]
by(*simp only*: *insert3-def ins3-def val-simps up2.case-distrib*[*of val*])

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*
by(*simp add*: *T-insert3-def ins3-def T-ins3-def tm-simps split*: *tm.split up2.split*)

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'*
by(*induction t arbitrary*: *d t' n' h'*)
(*auto simp*: *size-height add.commute max.commute balance-tree-def bal-tree*
split: *if-splits up2.splits prod.splits*)

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'*)
by(*induction t arbitrary*: *d t'*)
(*auto simp*: *size-height add.commute max.commute balance-tree-def bal-tree*
split: *if-splits up.splits prod.splits*)

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'*
using *ins2-ins*(*1*) *ins-ins2*(*1*) **apply** *blast*
using *ins2-ins*(*2*) *ins-ins2*(*2*) **apply** *blast*
using *ins2-ins*(*3*) *ins-ins2*(*3*) **by** *blast*

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

```
proof(induction t arbitrary: d)
  case Leaf
  thus ?case by (auto)
next
  case (Node l y r)
  consider (ls) x < y | (eq) x = y | (gr) x > y
    by(metis less-linear)
  thus ?case
  proof cases
    case ls
    have *: bal-i n (height l + (d + 1))
      using Node.prems by (simp add: mono-bal-i')
    note IH = Node.IH(1)[OF *]
    show ?thesis
    proof (cases ins2 n (d+1) x l)
      case Same2
      thus ?thesis
        using IH ls by (simp)
    next
      case Bal2
      thus ?thesis
        using IH ls by (simp)
    next
      case (Unbal2 l' nl' hl')
      let ?t' = Node l' y r let ?h' = height ?t' let ?n' = size ?t'
      have ins: ins n (d+1) x l = Unbal l'
        and nl' = size l' ∧ hl' = height l'
        using ins2-iff-ins(3)[THEN iffD1, OF Unbal2] by auto
      thus ?thesis
        using ls IH Unbal2 height-Unbal-l[OF ins Node.prems(1)]
        by(auto simp add: size-height mono-bal-i size-tree)
    qed
  next
    case eq
    thus ?thesis
      using Node.prems by(simp)
  next
    case gr
    have *: bal-i n (height r + (d + 1))
      using Node.prems by (simp add: mono-bal-i')
    note IH = Node.IH(2)[OF *]
    show ?thesis
    proof (cases ins2 n (d+1) x r)
      case Same2
      thus ?thesis
        using IH gr by (simp)
    next
      case Bal2
      thus ?thesis
```

```
      using IH gr by (simp)
    next
      case (Unbal2 r' nr' hr')
      let ?t' = Node r' y r let ?h' = height ?t' let ?n' = size ?t'
      have ins: ins n (d+1) x r = Unbal r'
        and nr' = size r' ∧ hr' = height r'
        using ins2-iff-ins(3)[THEN iffD1, OF Unbal2] by auto
      thus ?thesis
        using gr IH Unbal2 height-Unbal-r[OF ins Node.prems]
        by(auto simp add: size-height mono-bal-i size-tree)
    qed
  qed
qed

lemma insert2-insert:
  insert2 x (t,size t) = (t',n') ⟷ t' = insert x t ∧ n' = size t'
using ins0-neq-Unbal
by(auto simp: ins2-iff-ins ins-size split: up2.split up.split)

lemma insert3-insert2:
  bal-i n (height t) ⟹ insert3 x (t,n) = insert2 x (t,n)
by(simp add: ins3-ins2 split: up2.split)
```

2.3.3 Amortized Complexity

```
fun Φ :: 'a tree ⇒ real where
Φ Leaf = 0 |
Φ (Node l x r) = 6 * e * imbal (Node l x r) + Φ l + Φ r

lemma Φ-nn: Φ t ≥ 0
by(induction t) (use e0 in auto)

lemma Φ-sum-mset: Φ t = (∑ s ∈# subtrees-mset t. 6*e*imbal s)
proof(induction t)
  case Leaf show ?case by(simp add: imbal.simps)
next
  case Node thus ?case by(auto)
qed

lemma Φ-wbalanced: assumes wbalanced t shows Φ t = 0
proof −
  have Φ t = 6*e * (∑ s∈#subtrees-mset t. real (imbal s))
    by(simp add: Φ-sum-mset sum-mset-distrib-left)
  also have ... = (6*e) * real(∑ s∈#subtrees-mset t. imbal s)
    using e0 by (simp add: multiset.map-comp o-def)
  also have ... = 0 using e0 assms
    by(simp add: imbal0-if-wbalanced wbalanced-subtrees del: of-nat-sum-mset)
  finally show ?thesis .
qed
```

lemma *imbal-ins-Bal*: *ins n d x t = Bal t' ⟹*
real(imbal (node tw t' y s)) − imbal (node tw t y s) ≤ 1
apply(*drule ins-size*)
apply(*auto simp add*: *size1-size imbal.simps*)
done

lemma *imbal-ins-Unbal*: *ins n d x t = Unbal t' ⟹*
real(imbal (node tw t' y s)) − imbal (node tw t y s) ≤ 1
apply(*drule ins-size*)
apply(*auto simp add*: *size1-size imbal.simps*)
done

lemma *T-ins3-Same*:
ins3 n d x t = Same2 ⟹ T-ins3 n d x t ≤ 2 ∗ height t + 1
apply(*induction t arbitrary*: *d*)
apply *simp*
apply (*force simp*: *max-def split*!: *up2.splits if-splits*)
done

lemma *T-ins3-Unbal*:
⟦ *ins3 n d x t = Unbal2 t' n' h'*; *bal-i n (height t + d)* ⟧ *⟹*
T-ins3 n d x t ≤ 2 ∗ size t + 1 + height t
apply(*induction t arbitrary*: *d t' n' h'*)
apply *simp*
apply (*auto simp*: *ins3-ins2 ins2-iff-ins ins-height size-tree size1-size max-def mono-bal-i'*
dest: *unbal-ins-Unbal split*!: *up2.splits if-splits*)
apply (*fastforce simp*: *mono-bal-i'*)+
done

lemma *Phi-diff-Unbal*:
⟦ *ins3 n d x t = Unbal2 t' n' h'*; *bal-i n (height t + d)* ⟧ *⟹*
Φ t' − Φ t ≤ 6∗e ∗ height t
proof(*induction t arbitrary*: *d t' n' h'*)
case *Leaf* **thus** *?case*
by (*auto simp*: *imbal.simps split*: *if-splits*)
next
case (*Node l y r*)
have *ins*: *ins n d x ⟨l, y, r⟩ = Unbal t'*
using *Node.prems(1)*
by (*simp only*: *ins2-iff-ins(3) ins3-ins2*[*OF Node.prems(2)*])
consider (*ls*) *x < y* | (*eq*) *x = y* | (*gr*) *x > y*
by(*metis less-linear*)
thus *?case*
proof *cases*
case *ls*
with *Node.prems* **obtain** *l' nl' nh'* **where** *rec*: *ins3 n (d+1) x l = Unbal2 l' nl' nh'*
and *t'*: *t' = Node l' y r*

```
    by (auto split: up2.splits if-splits)
  have bal: bal-i n (height l + (d+1))
    using Node.prems(2) by (simp add: mono-bal-i' split: if-splits)
  have rec': ins n (d+1) x l = Unbal l'
    using rec ins-ins2(3) ins3-ins2[OF bal] by simp
  have Φ t' − Φ ⟨l,y,r⟩ = 6∗e∗imbal⟨l',y,r⟩ − 6∗e∗imbal⟨l,y,r⟩ + Φ l' − Φ l
    using t' by simp
  also have ... = 6∗e ∗ (real(imbal⟨l',y,r⟩) − imbal⟨l,y,r⟩) + Φ l' − Φ l
    by (simp add: ring-distribs)
  also have ... ≤ 6∗e + Φ l' − Φ l
    using imbal-ins-Unbal[OF rec', of False y r] e0 t' by (simp)
  also have ... ≤ 6∗e ∗ (height l + 1)
    using Node.IH(1)[OF rec bal] by (simp add: ring-distribs)
  also have ... ≤ 6∗e ∗ height ⟨l, y, r⟩
    using e0 by(simp del: times-divide-eq-left)
  finally show ?thesis .
 next
  case eq
  thus ?thesis using Node.prems by(simp)
 next
  case gr
  with Node.prems obtain r' rn' rh' where rec: ins3 n (d+1) x r = Unbal2 r'
rn' rh'
    and t': t' = Node l y r'
    by (auto split: up2.splits if-splits)
  have bal: bal-i n (height r + (d+1))
    using Node.prems(2) by (simp add: mono-bal-i' split: if-splits)
  have rec': ins n (d+1) x r = Unbal r'
    using rec ins-ins2(3) ins3-ins2[OF bal] by simp
  have Φ t' − Φ ⟨l,y,r⟩ = 6∗e∗imbal⟨l,y,r'⟩ − 6∗e∗imbal⟨l,y,r⟩ + Φ r' − Φ r
    using t' by simp
  also have ... = 6∗e ∗ (real(imbal⟨l,y,r'⟩) − imbal⟨l,y,r⟩) + Φ r' − Φ r
    by (simp add: ring-distribs)
  also have ... ≤ 6∗e + Φ r' − Φ r
    using imbal-ins-Unbal[OF rec', of True y l] e0 t'
    by (simp)
  also have ... ≤ 6∗e ∗ (height r + 1)
    using Node.IH(2)[OF rec bal] by (simp add: ring-distribs)
  also have ... ≤ 6∗e ∗ height ⟨l, y, r⟩
    using e0 by(simp del: times-divide-eq-left)
  finally show ?thesis .
 qed
qed

lemma amor-Unbal:
 ⟦ ins3 n d x t = Unbal2 t' n' h';  bal-i n (height t + d) ⟧ ⟹
 T-ins3 n d x t + Φ t' − Φ t ≤ 2∗size1 t + (6∗e + 1) ∗ height t
apply(frule (1) T-ins3-Unbal)
apply(drule (1) Phi-diff-Unbal)
```

by(*simp add*: *ring-distribs size1-size*)

lemma *T-ins3-Bal*:
⟦ *ins3 n d x t* = *Bal2 t'*; *bal-i n* (*height t* + *d*) ⟧
⟹ *T-ins3 n d x t* + Φ *t'* − Φ *t* ≤ (*6*e+2*) ∗ (*height t* + *1*)
proof(*induction t arbitrary*: *d t'*)
case *Leaf*
thus *?case*
using *e0* **by** (*auto simp*: *imbal.simps split*: *if-splits*)
next
case (*Node l y r*)
have *Bal*: *ins n d x* ⟨*l*, *y*, *r*⟩ = *Bal t'*
by (*metis Node.prems ins3-ins2 ins-ins2*(*2*))
consider (*ls*) *x* < *y* | (*eq*) *x* = *y* | (*gr*) *x* > *y* **by**(*metis less-linear*)
thus *?case*
proof *cases*
case *ls*
have ∗: *bal-i n* (*height l* + (*d+1*))
using *Node.prems*(*2*) **by** (*simp add*: *mono-bal-i'*)
show *?thesis*
proof (*cases ins3 n* (*d+1*) *x l*)
case *Same2*
thus *?thesis* **using** *Node ls* **by** (*simp*)
next
case (*Bal2 l'*)
have *Bal*: *ins n* (*d* + *1*) *x l* = *Bal l'*
using ∗ *Bal2* **by** (*auto simp*: *ins3-ins2 ins2-iff-ins*(*2*))
let *?t* = *Node l y r*
let *?t'* = *Node l' y r*
from *Bal2* **have** *t'*: *t'* = *?t'* **using** *Node.prems ls* **by** (*simp*)
have *T-ins3 n d x ?t* + Φ *t'* − Φ *?t* = *T-ins3 n* (*d+1*) *x l* + *2* + Φ *t'* − Φ *?t*
using *ls Bal2* **by** *simp*
also have ...
= *T-ins3 n* (*d+1*) *x l* + *6*e*imbal ?t'* + Φ *l'* − *6*e*imbal ?t* − Φ *l* + *2*
using *t'* **by** *simp*
also have ...
≤ *T-ins3 n* (*d+1*) *x l* + Φ *l'* − Φ *l* + *6*e*imbal ?t'* − *6*e*imbal ?t* + *2*
by *linarith*
also have ... ≤ (*6*e+2*) ∗ *height l* + *6*e*imbal ?t'* − *6*e*imbal ?t* + *6*e
+ *4*
using *Node.IH*(*1*)[*OF Bal2* ∗] **by**(*simp add*: *ring-distribs*)
also have ... = (*6*e+2*) ∗ *height l* + *6*e*(*real*(*imbal ?t'*) − *imbal ?t*) +
*6*e* + *4*
by(*simp add*: *algebra-simps*)
also have ... ≤ (*6*e+2*) ∗ *height l* + *6*e* + *6*e* + *4*
using *imbal-ins-Bal*[*OF Bal*, *of False y r*] *e0*
by (*simp del*: *times-divide-eq-left*)
also have ... = (*6*e+2*) ∗ (*height l* + *1*) + *6*e* + *2*
by (*simp add*: *ring-distribs*)

```
    also have ... ≤ (6*e+2) * (max (height l) (height r) + 1) + 6*e + 2
      using e0 by (simp add: mult-left-mono)
    also have ... ≤ (6*e+2) * (height ?t + 1)
      using e0 by(simp add: field-simps)
    finally show ?thesis .
  next
    case (Unbal2 l' nl' hl')
    have Unbal: ins n (d + 1) x l = Unbal l'
      and inv: nl' = size l' hl' = height l'
      using  Unbal2 ins3-ins2[OF *] by(auto simp add: ins2-iff-ins(3))
    have bal-l': bal-i (size l') (height l')
      by(fact bal-i-ins-Unbal[OF Unbal])
    let ?t = Node l y r let ?h = height ?t let ?n = size ?t
    let ?t' = Node l' y r let ?h' = height ?t' let ?n' = size ?t'
    have bal-t': ¬ bal-i ?n' ?h' using ls Unbal Bal by (auto)
    hence t': t' = balance-tree ?t' using ls Unbal Bal by (auto)
    have hl': height r < height l'
      by(fact height-Unbal-l[OF Unbal Node.prems(2)])
    have T-ins3 n d x ?t + Φ t' − Φ ?t = T-ins3 n d x ?t − Φ ?t
      by(simp add: t' Φ-wbalanced wbalanced-balance-tree)
    also have ... =  T-ins3 n d x ?t − 6*e * imbal ?t − Φ l − Φ r by simp
    also have ... ≤ T-ins3 n d x ?t − 6*e * imbal ?t − Φ l
      using Φ-nn[of r] by linarith
    also have ... ≤ T-ins3 n d x ?t − 6*e * imbal ?t' − Φ l + 6*e
      using mult-left-mono[OF imbal-ins-Unbal[OF Unbal, of False y r], of 4*e]
e0
      apply (simp only: node.simps if-False ring-distribs)
      by (simp)
    also have ... ≤ real(T-ins3 n d x ?t) − 6*(size1 ?t' − e) − Φ l + 6*e + 1
      using imbal-size[OF bal-t'] hl' bal-l' by(simp add: ring-distribs)
   also have ... = real(T-ins3 n (d+1) x l) + 2*size1 l' + 4*size1 r − 4*size1
?t' − Φ l + 6*e + 6*e + 1
      using ls Unbal2 inv bal-t' hl' by (simp add: T-bal-tree max-def size1-size)
    also have ... = real(T-ins3 n (d+1) x l) − 2*size1 l' − Φ l + 6*e + 6*e
+ 1
      by simp
    also have ... ≤ (6*e + 2) * height l + 6*e + 6*e
      using amor-Unbal[OF Unbal2 *] ins-size(2)[OF Unbal] Φ-nn[of l']
      by(simp add: ring-distribs size1-size)
    also have ... ≤ (6*e + 2) * (height l + 2)
      by (simp add: ring-distribs)
    also have ... ≤ (6*e+2) * (height ⟨l, y, r⟩ + 1)
      using e0 by (simp add: mult-mono del: times-divide-eq-left)
    finally show ?thesis by linarith
  qed
 next
  case eq thus ?thesis using Node.prems by(simp)
 next
  case gr
```

have $*$: *bal-i n* (*height r* + (*d*+*1*))
using *Node.prems*(*2*) **by** (*simp add*: *mono-bal-i'*)
show *?thesis*
proof (*cases ins3 n* (*d*+*1*) *x r*)
case *Same2*
thus *?thesis* **using** *Node gr* **by** (*simp*)
next
case (*Bal2 r'*)
have *Bal*: *ins n* (*d* + *1*) *x r* = *Bal r'*
using $*$ *Bal2* **by** (*auto simp*: *ins3-ins2 ins2-iff-ins*(*2*))
let *?t* = *Node l y r*
let *?t'* = *Node l y r'*
from *Bal2* **have** *t'*: *t'* = *?t'* **using** *Node.prems gr* **by** (*simp*)
have *T-ins3 n d x ?t* + Φ *t'* − Φ *?t* = *T-ins3 n* (*d*+*1*) *x r* + *2* + Φ *t'* − Φ *?t*
using *gr Bal2* **by** *simp*
also have ...
= *T-ins3 n* (*d*+*1*) *x r* + *6*∗*e*∗*imbal ?t'* + Φ *r'* − *6*∗*e*∗*imbal ?t* − Φ *r* + *2*
using *t'* **by** *simp*
also have ...
≤ *T-ins3 n* (*d*+*1*) *x r* + Φ *r'* − Φ *r* + *6*∗*e*∗*imbal ?t'* − *6*∗*e*∗*imbal ?t* + *2*
by *linarith*
also have ... ≤ (*6*∗*e*+*2*) ∗ *height r* + *6*∗*e*∗*imbal ?t'* − *6*∗*e*∗*imbal ?t* + *6*∗*e* + *4*
using *Node.IH*(*2*)[*OF Bal2* $*$] **by**(*simp add*: *ring-distribs*)
also have ... = (*6*∗*e*+*2*) ∗ *height r* + *6*∗*e*∗(*real*(*imbal ?t'*) − *imbal ?t*) + *6*∗*e* + *4*
by(*simp add*: *algebra-simps*)
also have ... ≤ (*6*∗*e*+*2*) ∗ *height r* + *6*∗*e* + *6*∗*e* + *4*
using *imbal-ins-Bal*[*OF Bal*, *of True y l*] *e0*
by (*simp del*: *times-divide-eq-left*)
also have ... = (*6*∗*e*+*2*) ∗ (*height r* + *1*) + *6*∗*e* + *2*
by (*simp add*: *ring-distribs*)
also have ... ≤ (*6*∗*e*+*2*) ∗ (*max* (*height l*) (*height r*) + *1*) + *6*∗*e* + *2*
using *e0* **by** (*simp add*: *mult-left-mono*)
also have ... = (*6*∗*e*+*2*) ∗ (*height ?t* + *1*)
using *e0* **by**(*simp add*: *field-simps*)
finally show *?thesis* **.**
next
case (*Unbal2 r' nr' hr'*)
have *Unbal*: *ins n* (*d* + *1*) *x r* = *Unbal r'*
and *inv*: *nr'* = *size r' hr'* = *height r'*
using *Unbal2 ins3-ins2*[*OF* $*$] **by**(*auto simp add*: *ins2-iff-ins*(*3*))
have *bal-r'*: *bal-i* (*size r'*) (*height r'*)
by(*fact bal-i-ins-Unbal*[*OF Unbal*])
let *?t* = *Node l y r* **let** *?h* = *height ?t* **let** *?n* = *size ?t*
let *?t'* = *Node l y r'* **let** *?h'* = *height ?t'* **let** *?n'* = *size ?t'*
have *bal-t'*: ¬ *bal-i ?n' ?h'* **using** *gr Unbal Bal* **by** (*auto*)
hence *t'*: *t'* = *balance-tree ?t'* **using** *gr Unbal Bal* **by** (*auto*)

```
    have hr': height l < height r'
      by(fact height-Unbal-r[OF Unbal Node.prems(2)])
    have T-ins3 n d x ?t + Φ t' − Φ ?t = T-ins3 n d x ?t − Φ ?t
      by(simp add: t' Φ-wbalanced wbalanced-balance-tree)
    also have ... =  T-ins3 n d x ?t − 6∗e ∗ imbal ?t − Φ r − Φ l by simp
    also have ... ≤ T-ins3 n d x ?t − 6∗e ∗ imbal ?t − Φ r
      using Φ-nn[of l] by linarith
    also have ... ≤ T-ins3 n d x ?t − 6∗e ∗ imbal ?t' − Φ r + 6∗e
     using mult-left-mono[OF imbal-ins-Unbal[OF Unbal, of True y l], of 4∗e] e0
      apply (simp only: node.simps if-True ring-distribs)
      by (simp)
    also have ... ≤ real(T-ins3 n d x ?t) − 6∗(size1 ?t' − e) − Φ r + 6∗e + 1
      using imbal-size[OF bal-t'] hr' bal-r' by (simp add: ring-distribs)
   also have ... = real(T-ins3 n (d+1) x r) + 2∗size1 r' + 4∗size1 l − 4∗size1
?t' − Φ r + 6∗e + 6∗e + 1
      using gr Unbal2 inv bal-t' hr' by (simp add: T-bal-tree max-def size1-size
add-ac)
    also have ... = real(T-ins3 n (d+1) x r) − 2∗size1 r' − Φ r + 6∗e + 6∗e
+ 1
      by simp
    also have ... ≤ (6∗e + 2) ∗ height r + 6∗e + 6∗e
      using amor-Unbal[OF Unbal2 ∗] ins-size(2)[OF Unbal] Φ-nn[of r']
      by(simp add: ring-distribs size1-size)
    also have ... ≤ (6∗e + 2) ∗ (height r + 2)
      by (simp add: ring-distribs)
    also have ... ≤ (6∗e+2) ∗ (height ⟨l, y, r⟩ + 1)
      using e0 by (simp add: mult-mono del: times-divide-eq-left)
    finally show ?thesis by linarith
  qed
 qed
qed

lemma T-insert3-amor: assumes n = size t bal-i (size t) (height t)
 insert3 a (t,n) = (t',n')
shows T-insert3 a (t,n) + Φ t' − Φ t ≤ (6∗e+2) ∗ (height t + 1) + 1
proof (cases ins3 (size t) 0 a t)
 case Same2
 have ∗: 5∗e ∗ real (height t') ≥ 0 using e0 by simp
 show ?thesis using Same2 assms(1,3) e0 T-ins3-Same[OF Same2]
  apply (simp add: ring-distribs T-insert3-def2) using ∗ by linarith
next
 case (Bal2 t')
 thus ?thesis
  using T-ins3-Bal[OF Bal2] assms by(simp add: ins-size T-insert3-def2)
next
 case Unbal2
 hence False using ins0-neq-Unbal
  using assms(1,2) ins3-ins2[of n t 0] by (fastforce simp: ins2-iff-ins(3))
 thus ?thesis ..
```

qed

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 (*standard*, *goal-cases*)
case *1*
show *?case* **using** *bal-i0* **by** (*simp split*: *prod.split*)
next
case (*2 s x*)
thus *?case* **using** *insert2-insert*[*of x fst s*] *bal-i-insert*[*of fst s*]
by (*simp del*: *insert2.simps insert3-def2 insert.simps*
add: *insert3-insert2 split*: *prod.splits*)
next
case (*3 s*)
thus *?case*
using Φ-*nn*[*of fst s*] **by** (*auto split*: *prod.splits*)
next
case *4*
show *?case* **by**(*simp*)
next
case (*5 s x*)
thus *?case* **using** *T-insert3-amor*[*of snd s fst s x*]
by (*auto simp del*: *insert3-def2 split*: *prod.splits*)
qed

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 (*standard*, *goal-cases*)
case (*1 t*)
have *0*: *log 2* (*1* + *real* (*size t*)) ≥ *0* **by** *simp*
have *1*: *log 2* (*1* + *real* (*size t*)) ≤ *c* ∗ *log 2* (*1* + *real* (*size t*))
using *c1 0 less-eq-real-def* **by** *auto*
thus *?case*
apply(*simp add*: *height-balance-tree add-ac ceiling-mono*)
using *0* **by** *linarith*
next
case (*2 n h n′ h′*)
have *int h′* ≤ *int h* **by**(*simp add*: *2*)
also have ... ≤ *ceiling*(*c* ∗ *log 2* (*real n* + *1*)) **by**(*rule 2*)
also have ... ≤ *ceiling*(*c* ∗ *log 2* (*real n′* + *1*))
using *c1 2*(*2*,*3*) **by** (*simp add*: *ceiling-mono*)
finally show *?case* **.**
next
case *3*
have *2 powr* (*1*/*c*) < *2 powr 1*
using *c1* **by** (*simp only*: *powr-less-cancel-iff*) *simp*
hence *2* − *2 powr* (*1* / *c*) > *0* **by** *simp*
thus *?case* **by**(*simp*)
next
case (*4 t*) **thus** *?case*
using *size1-imbal2*[*of t*]
by(*simp add*: *bal-log-def size1-size add-ac ring-distribs*)
qed

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*
using *cd0* **by**(*simp add*: *bal-d-def*)

lemma *mono-bal-d*: ⟦ *bal-d n dl*; *n* ≤ *n′* ⟧ ⟹ *bal-d n′ dl*
unfolding *bal-d-def*
using *cd0 mult-left-mono*[*of real n* + *1 real n′* + *1 cd*]
by *linarith*

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
using val-cong[OF del-tm.simps(1)]
by(simp only: del-def val-simps)

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')
using val-cong[OF del-tm.simps(2)]
by(simp only: del-def split-min-def val-simps cmp-val.case-distrib[of val])

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
by(simp add: T-del-def tm-simps)

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)
by(simp add: T-del-def T-split-min-def tm-simps split: tm.split prod.split)

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'
by(auto simp: ins-size ins0-neq-Unbal split: if-splits up.splits)

lemma inorder-insert-d: insert-d x (t,dl) = (t',dl') ⟹ sorted(inorder t)
  ⟹ inorder t' = ins-list x (inorder t)
by(auto simp add: ins0-neq-Unbal insert-def inorder-ins split: prod.split up.split)

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 (cases ins (size t + dl) 0 x t)
  case Same
  with assms show ?thesis by (simp)
next
  case Bal
  thus ?thesis
    using ins-bal-i-Bal[OF Bal] assms ins-size by(simp add: size1-size)
next
  case (Unbal t')
  hence False by(simp add: ins0-neq-Unbal)
  thus ?thesis ..
qed

lemma inorder-del:
  sorted(inorder t) ⟹
  inorder(case del x t of None ⇒ t | Some t' ⇒ t') = del-list x (inorder t)
by(induction t)
  (auto simp add: del-list-simps split-minD split: option.splits prod.splits)

lemma inorder-delete:
  ⟦ delete x (t,dl) = (t',dl'); sorted(inorder t) ⟧ ⟹
  inorder t' = del-list x (inorder t)
using inorder-del[of t x]
by(auto simp add: delete.simps split: option.splits if-splits)
```

```
lemma size-split-min:
  ⟦ split-min t = (a,t'); t ≠ Leaf ⟧ ⟹ size t' = size t − 1
by(induction t arbitrary: t')
  (auto simp add: zero-less-iff-neq-zero split: if-splits prod.splits)

lemma height-split-min:
  ⟦ split-min t = (a,t'); t ≠ Leaf ⟧ ⟹ height t' ≤ height t
apply(induction t arbitrary: t')
 apply simp
by(fastforce split: if-splits prod.splits)

lemma size-del: del x t = Some t' ⟹ size t' = size t − 1
proof(induction x t arbitrary: t' rule: del-tm.induct)
  case 1 thus ?case by simp
next
  case (2 x l y r)
  consider (ls) x < y | (eq) x = y | (gr) x > y
    by(metis less-linear)
  thus ?case
  proof cases
    case ls
    with 2.prems obtain l' where l': del x l = Some l'
      by(auto split: option.splits)
    hence [arith]: size l ≠ 0 by(cases l) auto
    show ?thesis using ls 2 l' by(auto)
  next
    case eq
    show ?thesis
    proof (cases r = Leaf)
      case True thus ?thesis using eq 2.prems by(simp)
    next
      case False
      thus ?thesis
        using eq 2.prems eq-size-0[of r]
        by (auto simp add: size-split-min simp del: eq-size-0 split: prod.splits)
    qed
  next
    case gr
    with 2.prems obtain r' where r': del x r = Some r'
      by(auto split: option.splits)
    hence [arith]: size r ≠ 0 by(cases r) auto
    show ?thesis using gr 2 r' by(auto)
  qed
qed

lemma height-del: del x t = Some t' ⟹ height t' ≤ height t
proof(induction x t arbitrary: t' rule: del-tm.induct)
  case 1 thus ?case by simp
next
```

```
  case (2 x l y r)
  consider (ls) x < y | (eq) x = y | (gr) x > y
    by(metis less-linear)
  thus ?case
  proof cases
    case ls
    thus ?thesis
      using 2 by(fastforce split: option.splits)
  next
    case eq
    thus ?thesis
      using 2.prems
      by (auto dest: height-split-min split: if-splits prod.splits)
  next
    case gr
    thus ?thesis
      using 2 by(fastforce split: option.splits)
  qed
qed

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 (cases del x t)
  case None
  with assms show ?thesis by (simp add: delete.simps)
next
  case Some
  hence size t ≠ 0 by(cases t) auto
  thus ?thesis
    using Some assms size-del height-del[OF Some]
    by(force simp add: delete.simps bal-i-balance mono-bal-i' split: if-splits)
qed

lemma bal-d-delete:
  ⟦ bal-d (size t) dl; delete x (t,dl) = (t',dl') ⟧
   ⟹ bal-d (size t') dl'
by (auto simp add: delete.simps bal-d0 size-del split: option.splits if-splits)
```

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 (standard, goal-cases)
  case 1 show ?case by simp
next
  case 2 thus ?case by(simp add: isin-set split: prod.splits)
next
```

case (*3 t*) **thus** *?case*
by(*auto simp del*: *insert-d.simps simp add*: *inorder-insert-d split*: *prod.splits*)
next
case (*4 tn x*)
obtain *t n* **where** *tn* = (*t*,*n*) **by** *fastforce*
thus *?case*
using *4* **by**(*auto simp*: *inorder-delete split*: *prod.splits*)
qed (*rule TrueI*)+

end

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

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*)
using *val-cong*[*OF insert3-d-tm.simps*(*1*)]
by(*simp only*: *insert3-d-def ins3-def val-simps up2.case-distrib*[*of val*])

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*)
by(*simp add*: *T-insert3-d-def T-ins3-def tm-simps split*: *tm.split up2.split*)

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*)))
using *val-cong*[*OF delete2-tm.simps*(*1*)]
by(*simp only*: *delete2-def ins3-def del-def bal-tree-def val-simps option.case-distrib*[*of val*])

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*)))
by(*auto simp add*: *T-delete2-def tm-simps T-del-def del-def T-bal-tree-def split*: *tm.split option.split*)

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′
by(*auto simp*: *ins2-iff-ins ins-size ins0-neq-Unbal split*: *up2.split up.split*)

lemma *insert3-insert2-d*:
bal-i (n+dl) (height t) ⟹ insert3-d x (t,n,dl) = insert2-d x (t,n,dl)
by(*simp add*: *ins3-ins2 split*: *up2.split*)

lemma *delete2-delete*:
delete2 x (t,size t,dl) = (t′,n′,dl′) ⟷
(t′,dl′) = delete x (t,dl) ∧ n′ = size t′
by(*auto simp*: *delete2-def2 delete.simps size-del balance-tree-def bal-tree split*: *option.splits*)

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)*
by(*simp split*: *prod.split*)

lemma *imbal-diff-decr*:
size r′ = size r − 1 ⟹
real(imbal (Node l x′ r′)) − imbal (Node l x r) ≤ 1
by(*simp add*: *imbal.simps*)

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 *(cases ins (size t + dl) 0 a t)*
case *Same*
have ∗: *5∗e ∗ real (height t) ≥ 0* **using** *e0* **by** *simp*
show *?thesis* **using** *T-ins3-Same[of size t + dl 0 a t] Same assms*
apply *(auto simp add: ring-distribs ins3-ins2 ins2-ins)*
using ∗ *e0*
apply *safe*
by *linarith*
next
case *(Bal t′)*
thus *?thesis*
using *T-ins3-Bal[of size t + dl 0 a t t′] Bal assms*
by(*simp add*: *ins-size ins3-ins2 ins2-ins*)
next
case *Unbal*
hence *False* **by**(*simp add*: *ins0-neq-Unbal*)
thus *?thesis* **..**
qed

lemma *T-split-min-ub*:
  $t \neq Leaf \Longrightarrow$ *T-split-min* $t \leq$ *height* $t + 1$
by(*induction t*) *auto*

lemma *T-del-ub*:
  T-del $x\ t \leq$ *height* $t + 1$
by(*induction t*) (*auto dest*: *T-split-min-ub*)

lemma *imbal-split-min*:
  split-min $t = (x,t') \Longrightarrow t \neq Leaf \Longrightarrow real(imbal\ t') - imbal\ t \leq 1$
proof(*induction t arbitrary*: t')
  case *Leaf* **thus** *?case*
    by *simp*
next
  case (*Node l y r*)
  thus *?case* **using** *size-split-min*[*OF Node.prems*]
    apply(*auto split*: *if-splits option.splits prod.splits*)
    apply(*auto simp*: *imbal.simps*)
    apply(*cases* t')
    apply (*simp add*: *imbal.simps*)
    apply (*simp add*: *imbal.simps*)
    done
qed

lemma *imbal-del-Some*:
  del $x\ t =$ *Some* $t' \Longrightarrow real(imbal\ t') - imbal\ t \leq 1$
proof(*induction t arbitrary*: t')
  case *Leaf*
  thus *?case*
    by (*auto simp add*: *imbal.simps split*!: *if-splits*)
next
  case (*Node t1 x2 t2*)
  thus *?case* **using** *size-del*[*OF Node.prems*]
    apply(*auto split*: *if-splits option.splits prod.splits*)
    apply(*auto simp*: *imbal.simps*)
    apply(*cases* t')
    apply (*simp add*: *imbal.simps*)
    apply (*simp add*: *imbal.simps*)
    done
qed

lemma *Phi-diff-split-min*:
  split-min $t = (x,\ t') \Longrightarrow t \neq Leaf \Longrightarrow \Phi\ t' - \Phi\ t \leq 6{*}e{*}height\ t$
proof(*induction t arbitrary*: t')
  case *Leaf* **thus** *?case*
    by *simp*
next
  case (*Node l y r*)

```
  note [arith] = e0
  thus ?case
  proof (cases l = Leaf)
    have *: - a ≤ b ⟷ 0 ≤ a+b for a b :: real by linarith
    case True
    thus ?thesis using Node.prems
      by(cases r)(auto simp: imbal.simps *)
  next
    case False
    with Node.prems obtain l' where rec: split-min l = (x,l')
      and t': t' = Node l' y r
      by (auto split: prod.splits)
    hence Φ t' - Φ ⟨l,y,r⟩ = 6*e*imbal⟨l',y,r⟩ - 6*e*imbal⟨l,y,r⟩ + Φ l' - Φ l
      by simp
    also have ... = 6*e * (real(imbal⟨l',y,r⟩) - imbal⟨l,y,r⟩) + Φ l' - Φ l
      by (simp add: ring-distribs)
    also have ... ≤ 6*e + Φ l' - Φ l
      using imbal-split-min[OF Node.prems(1)] t'
      by (simp)
    also have ... ≤ 6*e * (height l + 1)
      using Node.IH(1)[OF rec False] by (simp add: ring-distribs)
    also have ... ≤ 6*e * height ⟨l, y, r⟩
      by(simp del: times-divide-eq-left)
    finally show ?thesis .
  qed
qed

lemma Phi-diff-del-Some:
  del x t = Some t' ⟹ Φ t' - Φ t ≤ 6*e * height t
proof(induction t arbitrary: t')
  case Leaf thus ?case
    by (auto simp: imbal.simps)
next
  case (Node l y r)
  note [arith] = e0
  consider (ls) x < y | (eq) x = y | (gr) x > y
    by(metis less-linear)
  thus ?case
  proof cases
    case ls
    with Node.prems obtain l' where rec: del x l = Some l'
      and t': t' = Node l' y r
      by (auto split: option.splits)
    hence Φ t' - Φ ⟨l,y,r⟩ = 6*e*imbal⟨l',y,r⟩ - 6*e*imbal⟨l,y,r⟩ + Φ l' - Φ l
      by simp
    also have ... = 6*e * (real(imbal⟨l',y,r⟩) - imbal⟨l,y,r⟩) + Φ l' - Φ l
      by (simp add: ring-distribs)
    also have ... ≤ 6*e + Φ l' - Φ l
      using imbal-del-Some[OF Node.prems] t'
```

```
    by (simp)
  also have ... ≤ 6*e * (height l + 1)
    using Node.IH(1)[OF rec] by (simp add: ring-distribs)
  also have ... ≤ 6*e * height ⟨l, y, r⟩
    by(simp del: times-divide-eq-left)
  finally show ?thesis .
 next
  case [simp]: eq
  show ?thesis
  proof (cases r = Leaf)
    case [simp]: True
    show ?thesis
    proof (cases size t' = 0)
      case True
      thus ?thesis
        using Node.prems by(auto simp: imbal.simps of-nat-diff)
    next
      case [arith]: False
      show ?thesis using Node.prems by(simp add: imbal.simps of-nat-diff alge-
bra-simps)
    qed
  next
    case False
    then obtain a r' where *: split-min r = (a,r') using Node.prems
      by(auto split: prod.splits)
    from  mult-left-mono[OF imbal-diff-decr[OF size-split-min[OF this False], of
l a y], of 5*e]
    have 6*e*real (imbal ⟨l, a, r'⟩) - 6*e*real (imbal ⟨l, y, r⟩) ≤ 6*e
      by(simp add: ring-distribs)
    thus ?thesis using Node.prems * False Phi-diff-split-min[OF *]
      apply(auto simp add: max-def ring-distribs)
      using mult-less-cancel-left-pos[of 6*e height r height l] by linarith
  qed
 next
  case gr
  with Node.prems obtain r' where rec: del x r = Some r'
    and t': t' = Node l y r'
    by (auto split: option.splits)
  hence Φ t' - Φ ⟨l,y,r⟩ = 6*e*imbal⟨l,y,r'⟩ - 6*e*imbal⟨l,y,r⟩ + Φ r' - Φ r
    by simp
  also have ... = 6*e * (real(imbal⟨l,y,r'⟩) - imbal⟨l,y,r⟩) + Φ r' - Φ r
    by (simp add: ring-distribs)
  also have ... ≤ 6*e + Φ r' - Φ r
    using imbal-del-Some[OF Node.prems] t'
    by (simp)
  also have ... ≤ 6*e * (height r + 1)
    using Node.IH(2)[OF rec] by (simp add: ring-distribs)
  also have ... ≤ 6*e * height ⟨l, y, r⟩
    by(simp del: times-divide-eq-left)
```

```
    finally show ?thesis .
  qed
qed


lemma amor-del-Some:
  del x t = Some t' ⟹
  T-del x t + Φ t' − Φ t ≤ (6∗e + 1) ∗ height t + 1
apply(drule Phi-diff-del-Some)
using T-del-ub[of x t]
by (simp add: ring-distribs)

lemma cd1: 1/cd > 0
by(simp add: cd0)

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 (cases del x t)
  case None
  have ∗: 6∗e ∗ real (height t) ≥ 0 using e0 by simp
  show ?thesis using None
    apply (simp add: delete2-def2 T-delete2-def2 ring-distribs)
    using ∗ T-del-ub[of x t] cd1 by linarith
next
  case (Some t')
  show ?thesis
  proof (cases bal-d (n−1) (dl+1))
    case True
    thus ?thesis
      using assms Some amor-del-Some[OF Some]
     by(simp add: size-del delete2-def2 T-delete2-def2 algebra-simps add-divide-distrib)
  next
    case False
    from Some have [arith]: size t ≠ 0 by(cases t) (auto)
    have T-delete2 x (t, n, dl) + Φ_d (delete2 x (t,n,dl)) − Φ_d (t,n,dl) =
      T-delete2 x (t, n, dl) − Φ t − 4∗dl/cd
      using False Some
     by(simp add: delete2-def2 T-delete2-def2 Φ-wbalanced bal-tree assms size-del)
    also have ... = T-del x t + 4 ∗ size t − Φ t − 4∗dl/cd
       using False assms Some by(simp add: T-delete2-def2 T-bal-tree size-del
size1-size)
    also have ... ≤ (6∗e+1)∗height t + 4∗(size t − dl/cd + 1)
      using amor-del-Some[OF Some] Φ-nn[of t] Φ-nn[of t']
      by(simp add: ring-distribs)
    also have size t − dl/cd + 1 ≤ 1/cd + 1
      using assms False cd0 unfolding bal-d-def
      by(simp add: algebra-simps of-nat-diff)(simp add: field-simps)
    finally show ?thesis
```

```
    by(simp add: ring-distribs)
  qed
qed

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 = λf (t,-). case f of
  Insert - ⇒ (6∗e+2) ∗ (height t + 1) + 1 |
  Delete - ⇒ (6∗e+1) ∗ height t + 4/cd + 4
proof (standard, goal-cases)
  case 1
  show ?case using bal-i0 bal-d0 by (simp split: prod.split)
next
  case (2 s f)
  obtain t n dl where [simp]: s = (t,n,dl)
    using prod-cases3 by blast
  show ?case
  proof (cases f)
    case (Insert x)
    thus ?thesis
      using 2 insert2-insert-d[of x t dl] bal-i-insert-d[of x t dl]
        mono-bal-d[OF - size-insert-d]
      by (simp del: insert2-d.simps insert3-d-def2 insert-d.simps
            add: insert3-insert2-d split: prod.splits)
         fastforce
  next
    case (Delete x)
    thus ?thesis
      using 2 bal-i-delete[of t dl x] bal-d-delete[of t dl x]
      by (auto simp: delete2-delete)
  qed
next
  case (3 s)
  thus ?case
    using Φ-nn[of fst s ] cd0 by (auto split: prod.splits)
```

```
next
  case 4
  show ?case by(simp)
next
  case (5 s f)
  obtain t n dl where [simp]: s = (t,n,dl)
    using prod-cases3 by blast
  show ?case
  proof (cases f)
    case (Insert x)
    thus ?thesis
      using 5 insert2-insert-d[of x t dl] tinsert-d-amor[of n t x dl]
      by (fastforce simp del: insert2-d.simps insert3-d-def2 insert.simps
            simp add: insert3-insert2-d Φ_d-case split: prod.split)
  next
    case (Delete x)
    then show ?thesis
      using 5 delete2-delete[of x t dl] T-delete-amor[of n t x dl]
      by (simp)
  qed
qed

end

axiomatization b :: real where
b0: b > 0

axiomatization where
cd-le-log: cd ≤ 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-i = λn h. h ≤ ceiling(c ∗ log 2 (n+1))
and e = 2 powr (1/c) / (2 − 2 powr (1/c))
..
```

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

```
definition bal-i :: nat ⇒ nat ⇒ bool where
bal-i n h = (h ≤ ceiling(c ∗ log 2 (n+1)))

lemma assumes bal-i (size t + dl) (height t) bal-d (size t) dl
shows height t ≤ ceiling(c ∗ log 2 (size1 t)) + b)
proof −
  have ∗: 0 < real (size t + 1) + cd ∗ real (size t + 1)
    using cd0 by (simp add: add-pos-pos)
  have 0 < 2 powr (b / c) − 1
    using b0 c1 by(auto simp: less-powr-iff)
```

hence $**$: $0 < real\ (size\ t + 1) + (2\ powr\ (b\ /\ c) - 1) * real\ (size\ t + 1)$
by (*simp add*: *add-pos-pos*)
have $height\ t \leq ceiling(c * log\ 2\ (size\ t + 1 + dl))$
using *assms*(*1*) **by**(*simp add*: *bal-i-def add-ac*)
also have $\ldots \leq ceiling(c * log\ 2\ (size\ t + 1 + cd * (size\ t + 1)))$
using *c1 cd0 assms*(*2*)
by(*simp add*: *ceiling-mono add-pos-nonneg bal-d-def add-ac*)
also have $\ldots \leq ceiling(c * log\ 2\ (size\ t + 1 + (2\ powr\ (b/c) - 1) * (size\ t + 1)))$
using $*$ $**$ *cd-le-log c1* **by**(*simp add*: *ceiling-mono mult-left-mono*)
also have $\ldots = ceiling(c * log\ 2\ (2\ powr\ (b/c) * (size1\ t)))$
by(*simp add*:*algebra-simps size1-size*)
also have $\ldots = ceiling(c * (b/c + log\ 2\ (size1\ t)))$
by(*simp add*: *log-mult*)
also have $\ldots = ceiling(c * log\ 2\ (size1\ t) + b)$
using *c1* **by**(*simp add*: *algebra-simps*)
finally show *?thesis* **.**
qed

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 $p :: nat \Rightarrow nat \Rightarrow bool$
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*: $h < length\ tab \Longrightarrow p\ n\ h = (n \geq tab\ !\ h)$
apply *auto*
using *not-le-imp-less not-less-Least tab-LEAST* **apply** *auto*[*1*]
by (*metis LeastI mono-p p tab-LEAST*)

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*, *6817835603*]

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*
apply(*simp add*: *bal-tab-def c-def All-less-Suc*)
apply (*approximation 50*)
done

lemma *ceiling-least-real*: *ceiling*(*r*::*real*) = (*LEAST i*. *r* ≤ *i*)
by (*metis Least-equality ceiling-le le-of-int-ceiling*)

lemma *floor-greatest-real*: *floor*(*r*::*real*) = (*GREATEST i*. *i* ≤ *r*)
by (*metis Greatest-equality le-floor-iff of-int-floor-le*)

lemma *LEAST-eq-floor*:
(*LEAST n*. *int h* ≤ ⌈*c* ∗ *log* 2 (*real n* + 1)⌉) = *floor*((2::*real*) *powr* ((*real*(*h*)−1)/*c*))
proof −
have *int h* ≤ ⌈*c* ∗ *log* 2 (*real n* + 1)⌉
⟷ 2 *powr* ((*real*(*h*)−1)/*c*) < *real*(*n*)+1 (**is** *?L* = *?R*) **for** *n*
proof −
have *?L* ⟷ *h* < *c* ∗ *log* 2 (*real n* + 1) + 1 **by** *linarith*
also have ... ⟷ (*real h*−1)/*c* < *log* 2 (*real n* + 1)
using *c1* **by**(*simp add*: *field-simps*)
also have ... ⟷ 2 *powr* ((*real h*−1)/*c*) < 2 *powr* (*log* 2 (*real n* + 1))
by(*simp del*: *powr-log-cancel*)
also have ... ⟷ *?R*
by(*simp*)
finally show *?thesis* **.**
qed
moreover have ((*LEAST n*::*nat*. *r* < *n*+1) = *nat*(*floor r*)) **for** *r* :: *real*
by(*rule Least-equality*) *linarith*+
ultimately show *?thesis* **by** *simp*
qed

interpretation *Min-tab*
where *p* = *bal-i* **and** *tab* = *bal-tab*
proof(*unfold bal-i-def*, *standard*, *goal-cases*)
case (1 *n n*′ *h*)

```
  have int h ≤ ceiling(c * log 2 (real n + 1)) by(rule 1[unfolded bal-i-def])
  also have ... ≤ ceiling(c * log 2 (real n' + 1))
    using c1 1(1) by (simp add: ceiling-mono)
  finally show ?case .
next
  case (2 h)
  show ?case
  proof
    show int h ≤ ⌈c * log 2 (real (2 ^ h - 1) + 1)⌉
      apply(simp add: of-nat-diff log-nat-power) using c1
       by (metis ceiling-mono ceiling-of-nat order.order-iff-strict mult.left-neutral
mult-eq-0-iff of-nat-0-le-iff mult-le-cancel-iff1)
  qed
next
  case 3
  thus ?case using bal-tab-correct LEAST-eq-floor
    by (simp add: eq-iff[symmetric]) (metis nat-int)
qed
```

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)
by (simp add: bal-array-def tab-correct)

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.