Weight-Balanced Trees

Tobias Nipkow and Stefan Dirix

March 17, 2025

Abstract

This theory provides a verified implementation of weight-balanced trees following the work of Hirai and Yamamoto [4] who proved that all parameters in a certain range are valid, i.e. guarantee that insertion and deletion preserve weight-balance. Instead of a general theorem we provide parameterized proofs of preservation of the invariant that work for many (all?) valid parameters.

1 Introduction

Weight-balanced trees (*WB* trees) are a class of binary search trees of logarithmic height. They were invented by Nievergelt and Reingold [5, 6] who called them *trees of bounded balance*. They are parametrized by a constant. Parameters are called *valid* if they guarantee that insertion and deletion preserve the WB invariant. Blum and Mehlhorn [3] later discovered that there is a flaw in Nievergelt and Reingold's analysis of valid parameters and gave a detailed correctness proof for a modified range of parameters. Adams [1, 2] considered a slightly modified version of WB trees and analyzed which parameters are valid. The Haskell libraries `Data.Set` and `Data.Map` are based on Adams' papers but it was found that the implementation did not preserve the invariant. This motivated Hirai and Yamamoto [4] to verify the valid parameter range for the original definition of WB tree formally in Coq. They also showed that Adams' analysis is flawed by giving a counterexample to Adams' claimed range of valid parameters. Straka [8] analyzes valid parameters for Adam's variant. Yet another variant of WB trees was considered by Roura [7].

2 Weight-Balanced Trees Have Logarithmic Height, and More

theory *Weight_Balanced_Trees_log*
imports
  Complex_Main

HOL−Library.Tree
begin

lemmas *neq0_if* = *less_imp_neq dual_order.strict_implies_not_eq*

2.1 Logarithmic Height

The locale below is parameterized wrt to Δ. The original definition of weight-balanced trees [5, 6] uses α. The constants α and Δ are interdefinable. Below we start from Δ but derive α-versions of theorems as well.

locale *WBT0* =
fixes Δ :: *real*
begin

fun *balanced1* :: *'a tree* ⇒ *'a tree* ⇒ *bool* **where**
balanced1 t1 t2 = (*size1 t1* ≤ Δ ∗ *size1 t2*)

fun *wbt* :: *'a tree* ⇒ *bool* **where**
wbt Leaf = *True* |
wbt (*Node l _ r*) = (*balanced1 l r* ∧ *balanced1 r l* ∧ *wbt l* ∧ *wbt r*)

end

locale *WBT1* = *WBT0* +
assumes *Delta*: Δ ≥ 1
begin

definition α :: *real* **where**
α = 1/(Δ+1)

lemma *Delta_def*: Δ = 1/α − 1
unfolding *α_def* **by** *auto*

lemma shows *alpha_pos*: 0 < α **and** *alpha_ub*: α ≤ 1/2
unfolding *α_def* **using** *Delta* **by** *auto*

lemma *wbt_Node_alpha*: *wbt* (*Node l x r*) =
((*let q* = *size1 l* / (*size1 l* + *size1 r*)
in α ≤ *q* ∧ *q* ≤ 1 − α) ∧
wbt l ∧ *wbt r*)
proof −
have *l* > 0 ⟹ *r* > 0 ⟹
(1/(Δ+1) ≤ *l*/(*l*+*r*) ⟷ *r*/*l* ≤ Δ) ∧
(1/(Δ+1) ≤ *r*/(*l*+*r*) ⟷ *l*/*r* ≤ Δ) ∧
(*l*/(*l*+*r*) ≤ 1 − 1/(Δ+1) ⟷ *l*/*r* ≤ Δ) ∧
(*r*/(*l*+*r*) ≤ 1 − 1/(Δ+1) ⟷ *r*/*l* ≤ Δ) **for** *l r*
using *Delta* **by** (*simp add*: *field_simps divide_le_eq*)
thus *?thesis* **using** *Delta* **by**(*auto simp*: *α_def Let_def pos_divide_le_eq add_pos_pos*)

```
qed

lemma height_size1_Delta:
  wbt t ⟹ (1 + 1/Δ) ^(height t) ≤ size1 t
proof(induction t)
  case Leaf thus ?case by simp
next
  case (Node l a r)
  let ?t = Node l a r let ?s = size1 ?t let ?d = 1 + 1/Δ
  from Node.prems(1) have 1: size1 l * ?d ≤ ?s and 2: size1 r * ?d ≤ ?s
    using Delta by (auto simp: Let_def field_simps add_pos_pos neq0_if)
  show ?case
  proof (cases height l ≤ height r)
    case True
    hence ?d ^(height ?t) = ?d ^(height r) * ?d by(simp)
    also have ... ≤ size1 r * ?d
      using Node.IH(2) Node.prems Delta unfolding wbt.simps
      by (smt (verit) divide_nonneg_nonneg mult_mono of_nat_0_le_iff)
    also have ... ≤ ?s using 2 by (simp)
    finally show ?thesis .
  next
    case False
    hence ?d ^(height ?t) = ?d ^(height l) * ?d by(simp)
    also have ... ≤ size1 l * ?d
      using Node.IH(1) Node.prems Delta unfolding wbt.simps
      by (smt (verit) divide_nonneg_nonneg mult_mono of_nat_0_le_iff)
    also have ... ≤ ?s using 1 by (simp)
    finally show ?thesis .
  qed
qed

lemma height_size1_alpha:
  wbt t ⟹ (1/(1−α)) ^(height t) ≤ size1 t
proof(induction t)
  case Leaf thus ?case by simp
next
  note wbt.simps[simp del] wbt_Node_alpha[simp]
  case (Node l a r)
  let ?t = Node l a r let ?s = size1 ?t
  from Node.prems(1) have 1: size1 l / (1−α) ≤ ?s and 2: size1 r / (1−α) ≤ ?s
    using alpha_ub by (auto simp: Let_def field_simps add_pos_pos neq0_if)
  show ?case
  proof (cases height l ≤ height r)
    case True
    hence (1/(1−α)) ^(height ?t) = (1/(1−α)) ^(height r) * (1/(1−α)) by(simp)
    also have ... ≤ size1 r * (1/(1−α))
      using Node.IH(2) Node.prems unfolding wbt_Node_alpha
      by (smt (verit) mult_right_mono zero_le_divide_1_iff)
    also have ... ≤ ?s using 2 by (simp)
```

```
    finally show ?thesis .
  next
    case False
    hence (1/(1−α)) ^(height ?t) = (1/(1−α)) ^(height l) ∗ (1/(1−α)) by(simp)
    also have . . . ≤ size1 l ∗ (1/(1−α))
      using Node.IH(1) Node.prems unfolding wbt_Node_alpha
      by (smt (verit) mult_right_mono zero_le_divide_1_iff)
    also have . . . ≤ ?s using 1 by (simp)
    finally show ?thesis .
  qed
qed

lemma height_size1_log_Delta: assumes wbt t
shows height t ≤ log 2 (size1 t) / log 2 (1 + 1/Δ)
proof −
  from height_size1_Delta[OF assms]
  have height t ≤ log (1 + 1/Δ) (size1 t)
    using Delta le_log_of_power by auto
  also have . . . = log 2 (size1 t) / log 2 (1 + 1/Δ)
    by (simp add: log_base_change)
  finally show ?thesis .
qed

lemma height_size1_log_alpha: assumes wbt t
shows height t ≤ log 2 (size1 t) / log 2 (1/(1−α))
proof −
  from height_size1_alpha[OF assms]
  have height t ≤ log (1/(1−α)) (size1 t)
    using alpha_pos alpha_ub le_log_of_power by auto
  also have . . . = log 2 (size1 t) / log 2 (1/(1−α))
    by (simp add: log_base_change)
  finally show ?thesis .
qed

end
```

2.2 Every $1 \leq \Delta < 2$ Yields Exactly the Complete Trees

```
declare WBT0.wbt.simps [simp] WBT0.balanced1.simps [simp]

lemma wbt1_if_complete: assumes 1 ≤ Δ shows complete t =⇒ WBT0.wbt Δ t
apply(induction t)
 apply simp
apply (simp add: assms size1_if_complete)
done

lemma complete_if_wbt2: assumes Δ < 2 shows WBT0.wbt Δ t =⇒ complete t
proof(induction t)
  case Leaf
```

```
  then show ?case by simp
next
  case (Node t1 x t2)
  let ?h1 = height t1 let ?h2 = height t2
  from Node have *: complete t1 ∧ complete t2 by auto
  hence sz: size1 t1 = 2 ^ ?h1 ∧ size1 t2 = 2 ^ ?h2
    using size1_if_complete by blast
  show ?case
  proof (rule ccontr)
    assume ¬ complete ⟨t1, x, t2⟩
    hence ?h1 ≠ ?h2 using * by auto
    thus False
    proof (cases ?h1 < ?h2)
      case True
      hence 2 * (2::real) ^ ?h1 ≤ 2 ^ ?h2
        by (metis Suc_leI one_le_numeral power_Suc power_increasing)
      also have ... ≤ Δ * 2 ^ ?h1 using sz Node.prems by (simp)
      finally show False using ‹Δ < 2› by simp
    next
      case False
      with ‹?h1 ≠ ?h2› have ?h2 < ?h1 by linarith
      hence 2 * (2::real) ^ ?h2 ≤ 2 ^ ?h1
        by (metis Suc_leI one_le_numeral power_Suc power_increasing)
      also have ... ≤ Δ * 2 ^ ?h2 using sz Node.prems by (simp)
      finally show False using ‹Δ < 2› by simp
    qed
  qed
qed

end
```

3 Weight Balanced Tree Implementation of Sets

This theory follows Hirai and Yamamoto but we do not prove their general theorem. Instead we provide a short parameterized theory that, when interpreted with valid parameters, will prove perservation of the invariant for these parameters.

```
theory Weight_Balanced_Trees
imports
  HOL−Data_Structures.Isin2
begin

lemma neq_Leaf2_iff: t ≠ Leaf ⟷ (∃ l a n r. t = Node l (a,n) r)
by(cases t) auto

type-synonym 'a wbt = ('a * nat) tree

fun size_wbt :: 'a wbt ⇒ nat where
```

$size\_wbt\ Leaf = 0\ |$
$size\_wbt\ (Node\ \_\ (\_,\ n)\ \_) = n$

Smart constructor:

fun $N :: 'a\ wbt \Rightarrow 'a \Rightarrow 'a\ wbt \Rightarrow 'a\ wbt$ **where**
$N\ l\ a\ r = Node\ l\ (a,\ size\_wbt\ l + size\_wbt\ r + 1)\ r$

Basic Rotations:

fun $rot1L :: 'a\ wbt \Rightarrow 'a \Rightarrow 'a\ wbt \Rightarrow 'a \Rightarrow 'a\ wbt \Rightarrow 'a\ wbt$ **where**
$rot1L\ A\ a\ B\ b\ C = N\ (N\ A\ a\ B)\ b\ C$

fun $rot1R :: 'a\ wbt \Rightarrow 'a \Rightarrow 'a\ wbt \Rightarrow 'a \Rightarrow 'a\ wbt \Rightarrow 'a\ wbt$ **where**
$rot1R\ A\ a\ B\ b\ C = N\ A\ a\ (N\ B\ b\ C)$

fun $rot2 :: 'a\ wbt \Rightarrow 'a \Rightarrow 'a\ wbt \Rightarrow 'a \Rightarrow 'a\ wbt \Rightarrow 'a\ wbt$ **where**
$rot2\ A\ a\ (Node\ B1\ (b,\_)\ B2)\ c\ C = N\ (N\ A\ a\ B1)\ b\ (N\ B2\ c\ C)$

3.1 WB trees

Parameters:

Δ determines when a tree needs to be rebalanced

Γ determines whether it needs to be single or double rotation.

We represent rational numbers as pairs: $\Delta = \Delta1/\Delta2$ and $\Gamma = \Gamma1/\Gamma2$.

Hirai and Yamamoto [4] proved that under the following constraints insertion and deletion preserve the WB invariant, i.e. Δ and Γ are *valid*:

definition $valid\_params :: nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**
$valid\_params\ \Delta1\ \Delta2\ \Gamma1\ \Gamma2 = ($
$\Delta1 * 2 < \Delta2 * 9$ — right: $\Delta < 4.5\ \wedge$
$\Gamma1 * \Delta2 + \Gamma2 * \Delta2 \le \Gamma2 * \Delta1$ — left: $\Gamma + 1 \le \Delta\ \wedge$
$\Gamma1 * \Delta1 \ge \Gamma2 * (\Delta1 + \Delta2)$ — lower: $\Gamma \ge (\Delta + 1) / \Delta\ \wedge$
— upper:
$(5*\Delta2 \le 2*\Delta1 \wedge 1*\Delta1 < 3*\Delta2 \longrightarrow \Gamma1*2 \le \Gamma2*3)$
— $\Gamma \le 3/2$ if $2.5 \le \Delta < 3\ \wedge$
$(3*\Delta2 \le 1*\Delta1 \wedge 2*\Delta1 < 7*\Delta2 \longrightarrow \Gamma1*2 \le \Gamma2*4)$
— $\Gamma \le 4/2$ if $3 \le \Delta < 3.5\ \wedge$
$(7*\Delta2 \le 2*\Delta1 \wedge 1*\Delta1 < 4*\Delta2 \longrightarrow \Gamma1*3 \le \Gamma2*4)$
— $\Gamma \le 4/3$ when $3.5 \le \Delta < 4\ \wedge$
$(4*\Delta2 \le 1*\Delta1 \wedge 2*\Delta1 < 9*\Delta2 \longrightarrow \Gamma1*3 \le \Gamma2*5)$
— $\Gamma \le 5/3$ when $4 \le \Delta < 4.5$
)

We do not make use of these constraints and do not prove that they guarantee preservation of the invariant. Instead, we provide generic proofs of invariant preservation that work for many (all?) interpretations of locale WBT (below) with valid parameters. Further down we demonstrate this by interpreting *WBT* with a selection of valid parameters. [For some parameters, some *smt* proofs fail because *smt* on *nat*s fails although on non-negative *int*s it succeeds, i.e. the goal should be provable. This is a shortcoming of *smt* that is under investigation.]

Locale *WBT* comes with some minimal assumptions ($\Gamma 1 > \Gamma 2$ and $\Delta 1 > \Delta 2$) which follow from *valid_params* and from which we conclude some simple lemmas.

locale *WBT* =
fixes Δ1 Δ2 :: *nat* **and** Γ1 Γ2 :: *nat*
assumes *Delta_gr1*: Δ1 > Δ2 **and** *Gamma_gr1*: Γ1 > Γ2
begin

3.1.1 Balance Indicators

fun *balanced1* :: *'a wbt ⇒ 'a wbt ⇒ bool* **where**
balanced1 t1 t2 = (Δ1 ∗ (*size_wbt t1* + 1) ≥ Δ2 ∗ (*size_wbt t2* + 1))

The global weight-balanced tree invariant:

fun *wbt* :: *'a wbt ⇒ bool* **where**
wbt Leaf = *True*|
wbt (*Node l* (_, *n*) *r*) =
(*n* = *size l* + *size r* + 1 ∧ *balanced1 l r* ∧ *balanced1 r l* ∧ *wbt l* ∧ *wbt r*)

lemma *size_wbt_eq_size*[*simp*]: *wbt t* ⟹ *size_wbt t* = *size t*
by(*induction t*) *auto*

fun *single* :: *'a wbt ⇒ 'a wbt ⇒ bool* **where**
single t1 t2 = (Γ1 ∗ (*size_wbt t2* + 1) > Γ2 ∗ (*size_wbt t1* + 1))

3.1.2 Code

fun *rotateL* :: *'a wbt ⇒ 'a ⇒ 'a wbt ⇒ 'a wbt* **where**
rotateL A a (*Node B* (*b*, _) *C*) =
(*if single B C then rot1L A a B b C else rot2 A a B b C*)

fun *balanceL* :: *'a wbt ⇒ 'a ⇒ 'a wbt ⇒ 'a wbt* **where**
balanceL l a r = (*if balanced1 l r then N l a r else rotateL l a r*)

fun *rotateR* :: *'a wbt ⇒ 'a ⇒ 'a wbt ⇒ 'a wbt* **where**
rotateR (*Node A* (*a*, _) *B*) *b C* =
(*if single B A then rot1R A a B b C else rot2 A a B b C*)

fun *balanceR* :: *'a wbt ⇒ 'a ⇒ 'a wbt ⇒ 'a wbt* **where**
balanceR l a r = (*if balanced1 r l then N l a r else rotateR l a r*)

fun *insert* :: *'a::linorder ⇒ 'a wbt ⇒ 'a wbt* **where**
insert x Leaf = *Node Leaf* (*x*, 1) *Leaf* |
insert x (*Node l* (*a*, *n*) *r*) =

```
  (case cmp x a of
    LT ⇒ balanceR (insert x l) a r |
    GT ⇒ balanceL l a (insert x r) |
    EQ ⇒ Node l (a, n) r )

fun split_min :: 'a wbt ⇒ 'a * 'a wbt where
split_min (Node l (a, _) r) =
  (if l = Leaf then (a,r) else let (x,l') = split_min l in (x, balanceL l' a r))

fun del_max :: 'a wbt ⇒ 'a * 'a wbt where
del_max (Node l (a, _) r) =
  (if r = Leaf then (a,l) else let (x,r') = del_max r in (x, balanceR l a r'))

fun combine :: 'a wbt ⇒ 'a wbt ⇒ 'a wbt  where
combine Leaf Leaf = Leaf|
combine Leaf r = r|
combine l Leaf = l|
combine l r =
  (if size l > size r then
    let (lMax, l') = del_max l in balanceL l' lMax r
   else
    let (rMin, r') = split_min r in balanceR l rMin r')

fun delete :: 'a::linorder ⇒ 'a wbt ⇒ 'a wbt where
delete _ Leaf = Leaf |
delete x (Node l (a, _) r) =
 (case cmp x a of
   LT ⇒ balanceL (delete x l) a r |
   GT ⇒ balanceR l a (delete x r) |
   EQ ⇒ combine l r)
```

3.2 Functional Correctness Proofs

A WB tree must be of a certain structure if balanced1 and single are False.

```
lemma not_Leaf_if_not_balanced1:
  assumes ¬ balanced1 l r
  shows r ≠ Leaf
proof
  assume r = Leaf with assms Delta_gr1 show False by simp
qed

lemma not_Leaf_if_not_single:
  assumes ¬ single l r
  shows l ≠ Leaf
proof
  assume l = Leaf with assms Gamma_gr1 show False by simp
qed
```

3.2.1 Inorder Properties

lemma *inorder_rot2*:
B ≠ *Leaf* ⟹ *inorder*(*rot2 A a B b C*) = *inorder A* @ *a* # *inorder B* @ *b* # *inorder C*
by (*cases* (*A*,*a*,*B*,*b*,*C*) *rule*: *rot2.cases*) (*auto*)

lemma *inorder_rotateL*:
r ≠ *Leaf* ⟹ *inorder*(*rotateL l a r*) = *inorder l* @ *a* # *inorder r*
by (*induction l a r rule*: *rotateL.induct*) (*auto simp add*: *inorder_rot2 not_Leaf_if_not_single*)

lemma *inorder_rotateR*:
l ≠ *Leaf* ⟹ *inorder*(*rotateR l a r*) = *inorder l* @ *a* # *inorder r*
by (*induction l a r rule*: *rotateR.induct*) (*auto simp add*: *inorder_rot2 not_Leaf_if_not_single*)

lemma *inorder_insert*:
sorted(*inorder t*) ⟹ *inorder*(*insert x t*) = *ins_list x* (*inorder t*)
by (*induction t*)
(*auto simp*: *ins_list_simps inorder_rotateL inorder_rotateR not_Leaf_if_not_balanced1*)

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 inorder_rotateL not_Leaf_if_not_balanced1*
split: *prod.splits if_splits*)

lemma *del_maxD*:
del_max t = (*x*,*t'*) ⟹ *t* ≠ *Leaf* ⟹ *inorder t'* @ [*x*] = *inorder t*
by (*induction t arbitrary*: *t' rule*: *del_max.induct*)
(*auto simp*: *sorted_lems inorder_rotateR not_Leaf_if_not_balanced1*
split: *prod.splits if_splits*)

lemma *inorder_combine*:
inorder(*combine l r*) = *inorder l* @ *inorder r*
by(*induction l r rule*: *combine.induct*)
(*auto simp*: *del_maxD split_minD inorder_rotateL inorder_rotateR not_Leaf_if_not_balanced1*
simp del: *rotateL.simps rotateR.simps split*: *prod.splits*)

lemma *inorder_delete*:
sorted(*inorder t*) ⟹ *inorder*(*delete x t*) = *del_list x* (*inorder t*)
by(*induction t*)
(*auto simp*: *del_list_simps inorder_combine inorder_rotateL inorder_rotateR*
not_Leaf_if_not_balanced1 simp del: *rotateL.simps rotateR.simps*)

3.3 Size Lemmas

3.3.1 Insertion

lemma *size_rot2L*[*simp*]:
B ≠ *Leaf* ⟹ *size*(*rot2 A a B b C*) = *size A* + *size B* + *size C* + 2

```
by(induction A a B b C rule: rot2.induct) auto

lemma size_rotateR[simp]:
  l ≠ Leaf ⟹ size(rotateR l a r) = size l + size r + 1
by(induction l a r rule: rotateR.induct)
  (auto simp: not_Leaf_if_not_single simp del: rot2.simps)

lemma size_rotateL[simp]:
  r ≠ Leaf ⟹ size(rotateL l a r) = size l + size r + 1
by(induction l a r rule: rotateL.induct)
  (auto simp: not_Leaf_if_not_single simp del: rot2.simps)

lemma size_length: size t = length (inorder t)
by (induction t rule: inorder.induct) auto

lemma size_insert: size (insert x t) = (if isin t x then size t else Suc (size t))
by (induction t rule: tree2_induct) (auto simp: not_Leaf_if_not_balanced1)
```

3.3.2 Deletion

```
lemma size_delete_if_isin: isin t x ⟹ size t = Suc (size(delete x t))
proof (induction t rule: tree2_induct)
  case (Node _ a _ _)
  thus ?case
  proof (cases cmp x a)
   case LT thus ?thesis using Node.prems by (simp add: Node.IH(1) not_Leaf_if_not_balanced1)
  next
   case EQ thus ?thesis by simp (metis size_length inorder_combine length_append)
  next
   case GT thus ?thesis using Node.prems by (simp add: Node.IH(2) not_Leaf_if_not_balanced1)
  qed
qed (auto)

lemma delete_id_if_wbt_notin: wbt t ⟹ ¬ isin t x ⟹ delete x t = t
by (induction t) auto

lemma size_split_min: t ≠ Leaf ⟹ size t = Suc (size (snd (split_min t)))
by(induction t) (auto simp: not_Leaf_if_not_balanced1 split: if_splits prod.splits)

lemma size_del_max: t ≠ Leaf ⟹ size t = Suc(size(snd(del_max t)))
by(induction t) (auto simp: not_Leaf_if_not_balanced1 split: if_splits prod.splits)
```

3.4 Auxiliary Definitions

```
fun balanced1_arith :: nat ⇒ nat ⇒ bool where
balanced1_arith a b = (Δ1 * (a + 1) ≥ Δ2 * (b + 1))

fun balanced2_arith :: nat ⇒ nat ⇒ bool where
balanced2_arith a b = (balanced1_arith a b ∧ balanced1_arith b a)
```

fun *singly_balanced_arith* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
singly_balanced_arith x y w = (*balanced2_arith x y* ∧ *balanced2_arith* (*x*+*y*+1) *w*)

fun *doubly_balanced_arith* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
doubly_balanced_arith x y z w =
(*balanced2_arith x y* ∧ *balanced2_arith z w* ∧ *balanced2_arith* (*x*+*y*+1) (*z*+*w*+1))

end

3.5 Preservation of WB tree Invariant for Concrete Parameters

A number of sample interpretations with valid parameters:

interpretation *WBT* **where**
$\Delta 1 = 25$ **and** $\Delta 2 = 10$ **and** $\Gamma 1 = 14$ **and** $\Gamma 2 = 10$

by (*auto simp add*: *WBT_def*)

lemma *wbt_insert*:
wbt t ⟹ *wbt* (*insert x t*)
proof (*induction t rule*: *tree2_induct*)
case *Leaf* **show** *?case* **by** *simp*
next
case (*Node l a _ r*)
show *?case*
proof (*cases cmp x a*)
case *EQ* **thus** *?thesis* **using** *Node.prems* **by** *auto*
next
case [*simp*]: *LT*
let *?l′* = *insert x l*
show *?thesis*
proof (*cases balanced1 r ?l′*)
case *True* **thus** *?thesis* **using** *Node size_insert*[*of x l*] **by** *auto*
next
case [*simp*]: *False*

```
      hence ?l′ ≠ Leaf using not_Leaf_if_not_balanced1 by auto
      then obtain k ll′ al′ rl′ where [simp]: ?l′ = (Node ll′ (al′, k) rl′)
        by(meson neq_Leaf2_iff)
      show ?thesis
      proof (cases single rl′ ll′)
        case True thus ?thesis using Node size_insert[of x l]
          by (auto split: if_splits)
      next
        case isDouble: False
        then obtain k llr′ alr′ rlr′ where [simp]: rl′ = (Node llr′ (alr′, k) rlr′)
          using not_Leaf_if_not_single tree2_cases by blast
        show ?thesis using isDouble Node size_insert[of x l]
          by (auto split: if_splits)
      qed
    qed
  next
    case [simp]: GT
    let ?r′ = insert x r
    show ?thesis
    proof (cases balanced1 l ?r′)
      case True thus ?thesis using Node size_insert[of x r] by auto
    next
      case [simp]: False
      hence ?r′ ≠ Leaf using not_Leaf_if_not_balanced1 by auto
      then obtain k lr′ ar′ rr′ where [simp]: ?r′ = (Node lr′ (ar′, k) rr′)
        by(meson neq_Leaf2_iff)
      show ?thesis
      proof (cases single lr′ rr′)
        case True thus ?thesis using Node size_insert[of x r]
          by (auto split: if_splits)
      next
        case isDouble: False
        hence lr′ ≠ Leaf using not_Leaf_if_not_single by auto
        thus ?thesis
          using Node isDouble size_insert[of x r]
          by (auto simp: neq_Leaf2_iff split: if_splits)
      qed
    qed
  qed
qed

declare [[smt_nat_as_int]]
```

Show that invariant is preserved by deletion in the left/right subtree:

```
lemma wbt_balanceL:
  assumes wbt (Node l (a, n) r) wbt l′ size l = size l′ + 1
  shows wbt (balanceL l′ a′ r)
proof −
  have rl′Balanced: balanced1 r l′ using assms by auto
```

```
  have rBalanced: wbt r using assms(1) by simp
  show ?thesis
  proof (cases balanced1 l' r)
    case True thus ?thesis using assms(2) rBalanced rl'Balanced by auto
  next
    case notBalanced: False
    hence r ≠ Leaf using not_Leaf_if_not_balanced1 by auto
     then obtain k lr ar rr where [simp]: r = Node lr (ar, k) rr by(meson
neq_Leaf2_iff)
    show ?thesis
    proof (cases single lr rr)
      case single: True
      have singly_balanced_arith (size l') (size lr) (size rr)
        using assms(1) notBalanced rl'Balanced rBalanced single assms
        by (simp) (smt?)
      thus ?thesis using notBalanced single assms(2) rBalanced by simp
    next
      case isDouble: False
      hence lr ≠ Leaf using not_Leaf_if_not_single by auto
      then obtain k2 llr alr rlr where [simp]: lr = (Node llr (alr, k2) rlr)
        by(meson neq_Leaf2_iff)
      have doubly_balanced_arith (size l') (size llr) (size rlr) (size rr)
        using assms(1) notBalanced rl'Balanced rBalanced isDouble assms(2,3)
        by auto
      thus ?thesis using notBalanced isDouble assms(2) rBalanced by simp
    qed
  qed
qed

lemma wbt_balanceR:
  assumes wbt (Node l (a, n) r) wbt r' size r = size r' + 1
  shows wbt (balanceR l a' r')
proof -
  have lr'Balanced: balanced1 l r' using assms by auto
  have lBalanced: wbt l using assms(1) by simp
  show ?thesis
  proof (cases balanced1 r' l)
    case True thus ?thesis using assms(2) lBalanced lr'Balanced by simp
  next
    case notBalanced: False
    hence l ≠ Leaf using not_Leaf_if_not_balanced1 by auto
     then obtain k ll al rl where [simp]: l = (Node ll (al, k) rl) by(meson
neq_Leaf2_iff)
    show ?thesis
    proof (cases single rl ll)
      case single: True
      have singly_balanced_arith (size rl) (size r') (size ll)
        using assms(1) notBalanced lr'Balanced lBalanced single assms(2,3)
        apply (auto) apply((thin_tac _ = _)+, smt)? done
```

```
      thus ?thesis using assms(2) lBalanced notBalanced single by simp
    next
      case isDouble: False
      hence rl ≠ Leaf using not_Leaf_if_not_single by auto
      then obtain k lrl arl rrl where [simp]: rl = (Node lrl (arl, k) rrl)
        by(meson neq_Leaf2_iff)
      have doubly_balanced_arith (size ll) (size lrl) (size rrl) (size r')
        using assms(1) notBalanced lr'Balanced lBalanced isDouble assms(2,3)
        apply (auto) apply((thin_tac _ = _)+, smt)? done
      thus ?thesis using assms(2) lBalanced notBalanced isDouble by simp
    qed
  qed
qed

lemma wbt_split_min: t ≠ Leaf ⟹ wbt t ⟹ wbt (snd (split_min t))
proof (induction t rule: split_min.induct)
  case (1 l a m r)
  show ?case
  proof (cases l rule: tree2_cases)
    case Leaf thus ?thesis using 1.prems(2) by simp
  next
    case (Node ll al n rl)
    let ?l' = snd (split_min (Node ll (al, n) rl))
    have delBalanceL: snd (split_min (Node l (a, m) r)) = balanceL ?l' a r
      using Node by(auto split: prod.splits)
    have wbt ?l' using 1(1) 1.prems(2) Node by auto
    moreover have size l = size ?l' + 1
      using Node size_split_min by (metis Suc_eq_plus1 neq_Leaf2_iff)
    ultimately have wbt (balanceL ?l' a r)
      by (meson 1.prems(2) wbt_balanceL)
    thus ?thesis using delBalanceL by auto
  qed
qed (blast)

lemma wbt_del_max: t ≠ Leaf ⟹ wbt t ⟹ wbt (snd (del_max t))
proof (induction t rule: del_max.induct)
  case (1 l a m r)
  show ?case
  proof (cases r rule: tree2_cases)
    case Leaf thus ?thesis using 1.prems(2) by simp
  next
    case (Node lr ar n rr)
    then obtain r' where delMaxR: r' = snd (del_max (Node lr (ar, n) rr))
      by simp
    hence delBalanceR: snd (del_max (Node l (a, m) r)) = balanceR l a r'
      using Node by(auto split: prod.splits)
    have wbt r' using 1(1) 1.prems(2) Node delMaxR by auto
    moreover have size r = size r' + 1 using size_del_max Node delMaxR
      by (metis Suc_eq_plus1 tree.simps(3))
```

```
    ultimately have wbt (balanceR l a r')
      using wbt_balanceR by (metis 1.prems(2))
    thus ?thesis using delBalanceR by auto
  qed
qed (blast)

lemma wbt_delete: wbt t ⟹ wbt (delete x t)
proof (induction t rule: tree2_induct)
  case Leaf thus ?case by simp
next
  case (Node l a n r)
  show ?case
  proof (cases isin (Node l (a, n) r) x)
    case False thus ?thesis using Node.prems delete_id_if_wbt_notin by metis
  next
    case isin: True
    thus ?thesis
    proof (cases cmp x a)
      case LT
      let ?l' = delete x l
      have size l = size ?l' + 1
        using LT isin by (auto simp: size_delete_if_isin)
      hence wbt (balanceL ?l' a r)
        using Node.IH(1) Node.prems by (fastforce intro: wbt_balanceL)
      thus ?thesis by (simp add: LT)
    next
      case GT
      let ?r' = delete x r
      have wbt ?r' using Node.IH(2) Node.prems by simp
      moreover have size r = size ?r' + 1
        using GT Node.prems isin size_delete_if_isin by auto
      ultimately have wbt (balanceR l a ?r')
        by (meson Node.prems wbt_balanceR)
      thus ?thesis by (simp add: GT)
    next
      case [simp]: EQ
      hence xCombine: delete x (Node l (a, n) r) = combine l r by simp
      {
        assume l = Leaf r = Leaf hence ?thesis by simp
      }
      moreover
      {
        assume l = Leaf r ≠ Leaf
        hence ?thesis using Node.prems by (auto simp: neq_Leaf2_iff)
      }
      moreover
      {
        assume l ≠ Leaf r = Leaf
        hence ?thesis using Node.prems by (auto simp: neq_Leaf2_iff)
```

```
      }
      moreover
      {
        assume lrNotLeaf: l ≠ Leaf r ≠ Leaf
        then obtain kl kr ll al rl lr ar rr
          where [simp]: l = (Node ll (al, kl) rl) r = (Node lr (ar, kr) rr)
          by (meson neq_Leaf2_iff)
        have ?thesis
        proof (cases size l > size r)
          case True
          obtain lMax l' where letMax: del_max l = (lMax, l')
            by (metis prod.exhaust)
          hence balanceLeft: combine l r = balanceL l' lMax r
            using ‹size l > size r› by (simp)
          have wbt l'
            using Node.prems wbt_del_max[OF lrNotLeaf(1)] letMax
            by (metis wbt.simps(2) snd_conv)
          moreover have size l = size l' + 1
            using size_del_max[OF lrNotLeaf(1)] letMax by (simp)
          ultimately have wbt(balanceL l' lMax r)
            using wbt_balanceL by (metis Node.prems)
          thus ?thesis using balanceLeft by simp
        next
          case False
          obtain rMin r' where letMin: split_min r = (rMin, r')
            by (metis prod.exhaust)
          hence balanceRight: combine l r = balanceR l rMin r'
            using ‹¬ size l > size r› by (simp)
          have wbt r'
            using Node.prems wbt_split_min[OF lrNotLeaf(2)] letMin
            by (metis wbt.simps(2) snd_conv)
          moreover have size r = size r' + 1
            using size_split_min[OF lrNotLeaf(2)] letMin by simp
          ultimately have wbt(balanceR l rMin r')
            using wbt_balanceR by (metis Node.prems)
          thus ?thesis using balanceRight by simp
        qed
      }
      ultimately show ?thesis by blast
    qed
  qed
qed
```

3.6 The final correctness proof

```
interpretation S: Set_by_Ordered
where empty = Leaf and isin = isin and insert = insert and delete = delete
and inorder = inorder and inv = wbt
proof (standard, goal_cases)
```

case 1 **show** *?case* **by** *simp*
next
case 2 **thus** *?case* **by**(*simp add*: *isin_set_inorder*)
next
case 3 **thus** *?case* **by**(*simp add*: *inorder_insert*)
next
case 4 **thus** *?case* **by**(*simp add*: *inorder_delete*)
next
case 5 **show** *?case* **by** *simp*
next
case 6 **thus** *?case* **using** *wbt_insert* **by** *blast*
next
case 7 **thus** *?case* **using** *wbt_delete* **by** *blast*
qed

end

References

[1] S. Adams. Implementing sets efficiently in a functional language. Technical Report CSTR 92-10, Department of Electronics and Computer Science, University of Southampton, 1992.

[2] S. Adams. Efficient sets - A balancing act. *Journal of Functional Programming*, 3(4):553–561, 1993.

[3] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Compututer Science*, 11:303–320, 1980.

[4] Y. Hirai and K. Yamamoto. Balancing weight-balanced trees. *Journal of Functional Programming*, 21(3):287–307, 2011.

[5] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proc. 4th ACM Symposium on Theory of Computing*, STOC '72, pages 137–142. ACM, 1972.

[6] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[7] S. Roura. A new method for balancing binary search trees. In F. Orejas, P. Spirakis, and J. van Leeuwen, editors, *Automata, Languages and Programming, ICALP 2001*, volume 2076 of *LNCS*, pages 469–480. Springer, 2001.

[8] M. Straka. Adams' trees revisited. Correct and efficient implementation. In *Trends in Functional Programming*, volume 7193 of *LNCS*, pages 130–145. Springer, 2011.