

Weight-Balanced Trees

Tobias Nipkow and Stefan Dirix

December 14, 2021

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

This theory is based on the original definition of weight-balanced trees [5, 6] where the size of the child of a node must be a minimum and a maximum fraction of the size of the node.

theory *Weight_Balanced_Trees_log*

```

imports
  Complex_Main
  HOL-Library.Tree
begin

lemmas neq0_if = less_imp_neq dual_order.strict_implies_not_eq

locale WBT0 =
fixes  $\alpha :: \text{real}$ 
assumes alpha_pos: 0 < \alpha and alpha_ub: \alpha \le 1/2
begin

fun wbt :: 'a tree \Rightarrow bool where
wbt Leaf = True |
wbt (Node l r) = (wbt l \wedge wbt r \wedge (let ratio = size1 l / (size1 l + size1 r)
in \alpha \le ratio \wedge ratio \le 1 - \alpha))

lemma height_size1_exp:
wbt t \Longrightarrow t \neq Leaf \Longrightarrow 2 \le (1-\alpha) ^ (height t - 1) * size1 t
proof(induction t)
  case Leaf thus ?case by simp
next
  case (Node l a r)
  have  $0: 0 \le (1 - \alpha) ^ k$  for  $k$  using alpha_ub by simp
  let  $?t = \text{Node } l \ a \ r$  let  $?s = \text{size1 } ?t$ 
  from Node.prem1 have  $1: \text{size1 } l \le (1-\alpha) * ?s$  and  $2: \text{size1 } r \le (1-\alpha) * ?s$ 
  by (auto simp: Let_def field_simps add_pos_pos neq0_if)
  show  $?case$ 
  proof (cases l = Leaf \wedge r = Leaf)
    case True thus ?thesis by simp
  next
    case not_Leafs: False
    show  $?thesis$ 
    proof (cases height l \le height r)
      case hlr: True
      hence  $r: r \neq \text{Leaf}$  and  $hr: \text{height } r \neq 0$  using not_Leafs by (auto)
      have  $2 \le (1-\alpha) ^ (\text{height } r - 1) * \text{size1 } r$ 
      using Node.IH(2)[OF _ r] Node.prem1 by simp
      also have  $\dots \le (1-\alpha) ^ (\text{height } r - 1) * ((1-\alpha) * ?s)$ 
      by(rule mult_left_mono[OF 2 0])
      also have  $\dots = (1-\alpha) ^ (\text{height } r - 1 + 1) * ?s$  by simp
      also have  $\dots = (1-\alpha) ^ (\text{height } r) * ?s$ 
      using  $hr$  by (auto simp del: eq_height_0)
      finally show  $?thesis$  using  $hlr$  by (simp)
    next
      case hlr: False
      hence  $l: l \neq \text{Leaf}$  and  $hl: \text{height } l \neq 0$  using not_Leafs by (auto)
      have  $2 \le (1-\alpha) ^ (\text{height } l - 1) * \text{size1 } l$ 

```

```

    using Node.IH(1)[OF - l] Node.premis by simp
  also have ... ≤ (1-α) ^ (height l - 1) * ((1-α) * ?s)
    by(rule mult_left_mono[OF 1 0])
  also have ... = (1-α) ^ (height l - 1 + 1) * ?s by simp
  also have ... = (1-α) ^ (height l) * ?s
    using hl by (auto simp del: eq_height_0)
  finally show ?thesis using hlr by (simp)
qed
qed
qed

```

```

lemma height_size1_log: assumes wbt t t ≠ Leaf
shows height t ≤ (log 2 (size1 t) - 1) / log 2 (1/(1-α)) + 1
proof -
  have 1 ≤ log 2 ((1-α) ^ (height t - 1) * size1 t)
    using height_size1_exp[OF assms] by simp
  hence 1 ≤ log 2 ((1-α) ^ (height t - 1)) + log 2 (size1 t)
    using alpha_ub by(simp add: log_mult)
  hence 1 ≤ (height t - 1) * log 2 (1-α) + log 2 (size1 t)
    using alpha_ub by(simp add: log_nat_power)
  hence - (height t - 1) * log 2 (1-α) ≤ log 2 (size1 t) - 1
    by(simp add: algebra_simps)
  hence (height t - 1) * log 2 (1/(1-α)) ≤ log 2 (size1 t) - 1
    using alpha_ub by(simp add: log_divide)
  hence height t - 1 ≤ (log 2 (size1 t) - 1) / log 2 (1/(1-α))
    using alpha_pos alpha_ub by(simp add: field_simps log_divide)
  thus ?thesis by(simp)
qed
end
end
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  $\Rightarrow$  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: $\Delta = \Delta_1/\Delta_2$ and $\Gamma = \Gamma_1/\Gamma_2$.

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  $\Delta_1$   $\Delta_2$   $\Gamma_1$   $\Gamma_2$  = (
   $\Delta_1 * 2 < \Delta_2 * 9$  — right:  $\Delta < 4.5 \wedge$ 
   $\Gamma_1 * \Delta_2 + \Gamma_2 * \Delta_2 \leq \Gamma_2 * \Delta_1$  — left:  $\Gamma + 1 \leq \Delta \wedge$ 
   $\Gamma_1 * \Delta_1 \geq \Gamma_2 * (\Delta_1 + \Delta_2)$  — lower:  $\Gamma \geq (\Delta + 1) / \Delta \wedge$ 
  — upper:
  ( $5 * \Delta_2 \leq 2 * \Delta_1 \wedge 1 * \Delta_1 < 3 * \Delta_2 \longrightarrow \Gamma_1 * 2 \leq \Gamma_2 * 3$ )
  —  $\Gamma \leq 3/2$  if  $2.5 \leq \Delta < 3 \wedge$ 
  ( $3 * \Delta_2 \leq 1 * \Delta_1 \wedge 2 * \Delta_1 < 7 * \Delta_2 \longrightarrow \Gamma_1 * 2 \leq \Gamma_2 * 4$ )
  —  $\Gamma \leq 4/2$  if  $3 \leq \Delta < 3.5 \wedge$ 
  ( $7 * \Delta_2 \leq 2 * \Delta_1 \wedge 1 * \Delta_1 < 4 * \Delta_2 \longrightarrow \Gamma_1 * 3 \leq \Gamma_2 * 4$ )
  —  $\Gamma \leq 4/3$  when  $3.5 \leq \Delta < 4 \wedge$ 
  ( $4 * \Delta_2 \leq 1 * \Delta_1 \wedge 2 * \Delta_1 < 9 * \Delta_2 \longrightarrow \Gamma_1 * 3 \leq \Gamma_2 * 5$ )
  —  $\Gamma \leq 5/3$  when  $4 \leq \Delta < 4.5$ 
)

```

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 *nats* fails although on non-negative *ints* 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 ($\Gamma1 > \Gamma2$ and $\Delta1 > \Delta2$) which follow from *valid_params* and from which we conclude some simple lemmas.

```

locale WBT =
fixes  $\Delta1 \Delta2 :: nat$  and  $\Gamma1 \Gamma2 :: nat$ 
assumes Delta_gr1:  $\Delta1 > \Delta2$  and Gamma_gr1:  $\Gamma1 > \Gamma2$ 
begin

```

3.1.1 Balance Indicators

```

fun balanced1 :: 'a wbt  $\Rightarrow$  'a wbt  $\Rightarrow$  bool where
balanced1 t1 t2 = ( $\Delta1 * (size\_wbt\ t1 + 1) \geq \Delta2 * (size\_wbt\ t2 + 1)$ )

```

The global weight-balanced tree invariant:

```

fun wbt :: 'a wbt  $\Rightarrow$  bool where
wbt Leaf = True|
wbt (Node l ( $\_$ , n) r) =
  ( $n = size\ l + size\ r + 1 \wedge balanced1\ l\ r \wedge balanced1\ r\ l \wedge wbt\ l \wedge wbt\ r$ )

```

```

lemma size_wbt_eq_size[simp]: wbt t  $\Longrightarrow$  size_wbt t = size t
by(induction t) auto

```

```

fun single :: 'a wbt  $\Rightarrow$  'a wbt  $\Rightarrow$  bool where
single t1 t2 = ( $\Gamma1 * (size\_wbt\ t2 + 1) > \Gamma2 * (size\_wbt\ t1 + 1)$ )

```

3.1.2 Code

```

fun rotateL :: 'a wbt  $\Rightarrow$  'a  $\Rightarrow$  'a wbt  $\Rightarrow$  '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  $\Rightarrow$  'a  $\Rightarrow$  'a wbt  $\Rightarrow$  '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  $\Rightarrow$  'a  $\Rightarrow$  'a wbt  $\Rightarrow$  '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  $\Rightarrow$  'a  $\Rightarrow$  'a wbt  $\Rightarrow$  '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  $\Rightarrow$  'a wbt  $\Rightarrow$  '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 \neg `balanced1 l r`
shows `r` \neq `Leaf`

proof

assume `r = Leaf` **with** `assms` `Delta_gr1` **show** `False` **by** `simp`

qed

lemma `not_Leaf_if_not_single`:

assumes \neg `single l r`
shows `l` \neq `Leaf`

proof

assume `l = Leaf` **with** `assms` `Gamma_gr1` **show** `False` **by** `simp`

qed

3.2.1 Inorder Properties

lemma *inorder_rot2*:

$B \neq \text{Leaf} \implies \text{inorder}(\text{rot2 } A \ a \ B \ b \ C) = \text{inorder } A \ @ \ a \ \# \ \text{inorder } B \ @ \ b \ \# \ \text{inorder } C$

by (*cases* (A,a,B,b,C) *rule*: *rot2.cases*) (*auto*)

lemma *inorder_rotateL*:

$r \neq \text{Leaf} \implies \text{inorder}(\text{rotateL } l \ a \ r) = \text{inorder } l \ @ \ a \ \# \ \text{inorder } r$

by (*induction* *l a r* *rule*: *rotateL.induct*) (*auto simp add*: *inorder_rot2 not_Leaf_if_not_single*)

lemma *inorder_rotateR*:

$l \neq \text{Leaf} \implies \text{inorder}(\text{rotateR } l \ a \ r) = \text{inorder } l \ @ \ a \ \# \ \text{inorder } r$

by (*induction* *l a r* *rule*: *rotateR.induct*) (*auto simp add*: *inorder_rot2 not_Leaf_if_not_single*)

lemma *inorder_insert*:

$\text{sorted}(\text{inorder } t) \implies \text{inorder}(\text{insert } x \ t) = \text{ins_list } x \ (\text{inorder } t)$

by (*induction* *t*)

(*auto simp*: *ins_list_simps inorder_rotateL inorder_rotateR not_Leaf_if_not_balanced1*)

lemma *split_minD*:

$\text{split_min } t = (x, t') \implies t \neq \text{Leaf} \implies x \ \# \ \text{inorder } t' = \text{inorder } t$

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*:

$\text{del_max } t = (x, t') \implies t \neq \text{Leaf} \implies \text{inorder } t' \ @ \ [x] = \text{inorder } t$

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*:

$\text{inorder}(\text{combine } l \ r) = \text{inorder } l \ @ \ \text{inorder } r$

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*:

$\text{sorted}(\text{inorder } t) \implies \text{inorder}(\text{delete } x \ t) = \text{del_list } x \ (\text{inorder } t)$

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]*:

$B \neq \text{Leaf} \implies \text{size}(\text{rot2 } A \ a \ B \ b \ C) = \text{size } A + \text{size } B + \text{size } C + 2$

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

lemma *size_rotateR[simp]*:
 $l \neq \text{Leaf} \implies \text{size}(\text{rotateR } l \ a \ r) = \text{size } l + \text{size } r + 1$
by(*induction l a r rule: rotateR.induct*)
(*auto simp: not_Leaf_if_not_single simp del: rot2.simps*)

lemma *size_rotateL[simp]*:
 $r \neq \text{Leaf} \implies \text{size}(\text{rotateL } l \ a \ r) = \text{size } l + \text{size } r + 1$
by(*induction l a r rule: rotateL.induct*)
(*auto simp: not_Leaf_if_not_single simp del: rot2.simps*)

lemma *size_length*: $\text{size } t = \text{length } (\text{inorder } t)$
by (*induction t rule: inorder.induct*) *auto*

lemma *size_insert*: $\text{size } (\text{insert } x \ t) = (\text{if } \text{isin } t \ x \ \text{then } \text{size } t \ \text{else } \text{Suc } (\text{size } t))$
by (*induction t rule: tree2_induct*) (*auto simp: not_Leaf_if_not_balanced1*)

3.3.2 Deletion

lemma *size_delete_if_isin*: $\text{isin } t \ x \implies \text{size } t = \text{Suc } (\text{size}(\text{delete } x \ t))$
proof (*induction t rule: tree2_induct*)
 case (*Node _ a _*)
 thus *?case*
 proof (*cases cmp x a*)
 case *LT* **thus** *?thesis using Node.prem1 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.prem2 by (simp add: Node.IH(2) not_Leaf_if_not_balanced1)*
 qed
qed (*auto*)

lemma *delete_id_if_wbt_notin*: $\text{wbt } t \implies \neg \text{isin } t \ x \implies \text{delete } x \ t = t$
by (*induction t*) *auto*

lemma *size_split_min*: $t \neq \text{Leaf} \implies \text{size } t = \text{Suc } (\text{size } (\text{snd } (\text{split_min } t)))$
by(*induction t*) (*auto simp: not_Leaf_if_not_balanced1 split: if_splits prod.splits*)

lemma *size_del_max*: $t \neq \text{Leaf} \implies \text{size } t = \text{Suc}(\text{size}(\text{snd}(\text{del_max } t)))$
by(*induction t*) (*auto simp: not_Leaf_if_not_balanced1 split: if_splits prod.splits*)

3.4 Auxiliary Definitions

fun *balanced1_arith* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**
*balanced1_arith a b = ($\Delta 1 * (a + 1) \geq \Delta 2 * (b + 1)$)*

fun *balanced2_arith* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**
balanced2_arith a b = (balanced1_arith a b \wedge balanced1_arith b a)


```

fun singly_balanced_arith :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bool where
singly_balanced_arith x y w = (balanced2_arith x y  $\wedge$  balanced2_arith (x+y+1) w)

fun doubly_balanced_arith :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bool where
doubly_balanced_arith x y z w =
  (balanced2_arith x y  $\wedge$  balanced2_arith z w  $\wedge$  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  $\implies$  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.prem 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)
      apply (auto) apply((thin_tac _ = _)+, smt)? done
    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 l'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.prem(2))
  thus ?thesis using delBalanceR by auto
qed
qed (blast)

lemma wbt_delete: wbt t  $\implies$  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.prem 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.prem 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.prem by simp
      moreover have size r = size ?r' + 1
        using GT Node.prem isin size_delete_if_isin by auto
      ultimately have wbt (balanceR l a ?r')
        by (meson Node.prem 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  $\neq$  Leaf
        hence ?thesis using Node.prem by (auto simp: neq_Leaf2_iff)
      }
      moreover
      {
        assume l  $\neq$  Leaf r = Leaf
        hence ?thesis using Node.prem by (auto simp: neq_Leaf2_iff)
      }
    }
  }

```

```

}
moreover
{
  assume lrNotLeaf:  $l \neq \text{Leaf } r \neq \text{Leaf}$ 
  then obtain kl kr ll al rl lr ar rr
    where [simp]:  $l = (\text{Node } ll \ (al, kl) \ rl) \ r = (\text{Node } lr \ (ar, kr) \ rr)$ 
    by (meson neq_Leaf2_iff)
  have ?thesis
  proof (cases size l > size r)
    case True
      obtain lMax l' where letMax:  $\text{del\_max } l = (lMax, l')$ 
        by (metis prod.exhaust)
      hence balanceLeft:  $\text{combine } l \ r = \text{balanceL } l' \ lMax \ r$ 
        using  $\langle \text{size } l > \text{size } r \rangle$  by (simp)
      have wbt l'
        using Node.premis wbt_del_max[OF lrNotLeaf(1)] letMax
        by (metis wbt.simps(2) snd_conv)
      moreover have  $\text{size } l = \text{size } l' + 1$ 
        using size_del_max[OF lrNotLeaf(1)] letMax by (simp)
      ultimately have  $\text{wbt}(\text{balanceL } l' \ lMax \ r)$ 
        using wbt_balanceL by (metis Node.premis)
      thus ?thesis using balanceLeft by simp
    next
      case False
        obtain rMin r' where letMin:  $\text{split\_min } r = (rMin, r')$ 
          by (metis prod.exhaust)
        hence balanceRight:  $\text{combine } l \ r = \text{balanceR } l \ rMin \ r'$ 
          using  $\langle \neg \text{size } l > \text{size } r \rangle$  by (simp)
        have wbt r'
          using Node.premis wbt_split_min[OF lrNotLeaf(2)] letMin
          by (metis wbt.simps(2) snd_conv)
        moreover have  $\text{size } r = \text{size } r' + 1$ 
          using size_split_min[OF lrNotLeaf(2)] letMin by simp
        ultimately have  $\text{wbt}(\text{balanceR } l \ rMin \ r')$ 
          using wbt_balanceR by (metis Node.premis)
        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 Computer 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.