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 α :: *real*
assumes *alpha_pos*: $0 < \alpha$ **and** *alpha_ub*: $\alpha \leq 1/2$
begin

fun *wbt* :: *'a tree* ⇒ *bool* **where**
wbt Leaf = *True* |
wbt (*Node l* _ *r*) = (*wbt l* ∧ *wbt r* ∧ (*let ratio* = *size1 l* / (*size1 l* + *size1 r*)
in $\alpha \leq$ *ratio* ∧ *ratio* $\leq 1 - \alpha$))

lemma *height_size1_exp*:
wbt t ⟹ *t* ≠ *Leaf* ⟹ 2 ≤ (1−α) ^ (*height t* − 1) ∗ *size1 t*
⟨*proof*⟩

lemma *height_size1_log*: **assumes** *wbt t t* ≠ *Leaf*
shows *height t* ≤ (*log* 2 (*size1 t*) − 1) / *log* 2 (1/(1−α)) + 1
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ⇒ *'a* ⇒ *'a wbt* ⇒ *'a wbt* **where**
N l a r = *Node l* (*a*, *size_wbt l* + *size_wbt r* + 1) *r*

Basic Rotations:

fun *rot*1*L* :: *'a wbt* ⇒ *'a* ⇒ *'a wbt* ⇒ *'a* ⇒ *'a wbt* ⇒ *'a wbt* **where**
*rot*1*L A a B b C* = *N* (*N A a B*) *b C*

fun *rot*1*R* :: *'a wbt* ⇒ *'a* ⇒ *'a wbt* ⇒ *'a* ⇒ *'a wbt* ⇒ *'a wbt* **where**
*rot*1*R A a B b C* = *N A a* (*N B b C*)

fun *rot*2 :: *'a wbt* ⇒ *'a* ⇒ *'a wbt* ⇒ *'a* ⇒ *'a wbt* ⇒ *'a wbt* **where**
*rot*2 *A a* (*Node B*1 (*b*,_) *B*2) *c C* = *N* (*N A a B*1) *b* (*N B*2 *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: Δ = Δ1/Δ2 and Γ = Γ1/Γ2.

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* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
valid_params Δ1 Δ2 Γ1 Γ2 = (
Δ1 ∗ 2 < Δ2 ∗ 9 — right: Δ < 4.5 ∧
Γ1 ∗ Δ2 + Γ2 ∗ Δ2 ≤ Γ2 ∗ Δ1 — left: Γ + 1 ≤ Δ ∧
Γ1 ∗ Δ1 ≥ Γ2 ∗ (Δ1 + Δ2) — lower: Γ ≥ (Δ + 1) / Δ ∧
— upper:
(5∗Δ2 ≤ 2∗Δ1 ∧ 1∗Δ1 < 3∗Δ2 ⟶ Γ1∗2 ≤ Γ2∗3)
— Γ ≤ 3/2 if 2.5 ≤ Δ < 3 ∧
(3∗Δ2 ≤ 1∗Δ1 ∧ 2∗Δ1 < 7∗Δ2 ⟶ Γ1∗2 ≤ Γ2∗4)
— Γ ≤ 4/2 if 3 ≤ Δ < 3.5 ∧
(7∗Δ2 ≤ 2∗Δ1 ∧ 1∗Δ1 < 4∗Δ2 ⟶ Γ1∗3 ≤ Γ2∗4)
— Γ ≤ 4/3 when 3.5 ≤ Δ < 4 ∧
(4∗Δ2 ≤ 1∗Δ1 ∧ 2∗Δ1 < 9∗Δ2 ⟶ Γ1∗3 ≤ Γ2∗5)
— Γ ≤ 5/3 when 4 ≤ Δ < 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 $\Delta 1$ $\Delta 2$ :: *nat* **and** $\Gamma 1$ $\Gamma 2$ :: *nat*
assumes *Delta_gr*1: $\Delta 1 > \Delta 2$ **and** *Gamma_gr*1: $\Gamma 1 > \Gamma 2$
begin

3.1.1 Balance Indicators

fun *balanced*1 :: *'a wbt* ⇒ *'a wbt* ⇒ *bool* **where**
*balanced*1 *t*1 *t*2 = ($\Delta 1$ ∗ (*size_wbt t*1 + 1) ≥ $\Delta 2$ ∗ (*size_wbt t*2 + 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 ∧ *balanced*1 *l r* ∧ *balanced*1 *r l* ∧ *wbt l* ∧ *wbt r*)

lemma *size_wbt_eq_size*[*simp*]: *wbt t* ⟹ *size_wbt t* = *size t*
⟨*proof*⟩

fun *single* :: *'a wbt* ⇒ *'a wbt* ⇒ *bool* **where**
*single t*1 *t*2 = ($\Gamma 1$ ∗ (*size_wbt t*2 + 1) > $\Gamma 2$ ∗ (*size_wbt t*1 + 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 rot*1*L A a B b C else rot*2 *A a B b C*)

fun *balanceL* :: *'a wbt* ⇒ *'a* ⇒ *'a wbt* ⇒ *'a wbt* **where**
balanceL l a r = (*if balanced*1 *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 rot*1*R A a B b C else rot*2 *A a B b C*)

fun *balanceR* :: *'a wbt* ⇒ *'a* ⇒ *'a wbt* ⇒ *'a wbt* **where**
balanceR l a r = (*if balanced*1 *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⟩

lemma not_Leaf_if_not_single:
  assumes ¬ single l r
  shows l ≠ Leaf
⟨proof⟩
```

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
```

⟨*proof*⟩

lemma *inorder_rotateL*:
r ≠ *Leaf* ⟹ *inorder*(*rotateL l a r*) = *inorder l* @ *a* # *inorder r*
⟨*proof*⟩

lemma *inorder_rotateR*:
l ≠ *Leaf* ⟹ *inorder*(*rotateR l a r*) = *inorder l* @ *a* # *inorder r*
⟨*proof*⟩

lemma *inorder_insert*:
sorted(*inorder t*) ⟹ *inorder*(*insert x t*) = *ins_list x* (*inorder t*)
⟨*proof*⟩

lemma *split_minD*:
split_min t = (*x*,*t*′) ⟹ *t* ≠ *Leaf* ⟹ *x* # *inorder t*′ = *inorder t*
⟨*proof*⟩

lemma *del_maxD*:
del_max t = (*x*,*t*′) ⟹ *t* ≠ *Leaf* ⟹ *inorder t*′ @ [*x*] = *inorder t*
⟨*proof*⟩

lemma *inorder_combine*:
inorder(*combine l r*) = *inorder l* @ *inorder r*
⟨*proof*⟩

lemma *inorder_delete*:
sorted(*inorder t*) ⟹ *inorder*(*delete x t*) = *del_list x* (*inorder t*)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *size_rotateR*[*simp*]:
l ≠ *Leaf* ⟹ *size*(*rotateR l a r*) = *size l* + *size r* + 1
⟨*proof*⟩

lemma *size_rotateL*[*simp*]:
r ≠ *Leaf* ⟹ *size*(*rotateL l a r*) = *size l* + *size r* + 1
⟨*proof*⟩

lemma *size_length*: *size t* = *length* (*inorder t*)
⟨*proof*⟩

lemma *size_insert*: *size* (*insert x t*) = (*if isin t x then size t else Suc* (*size t*))

⟨*proof*⟩

3.3.2 Deletion

lemma *size_delete_if_isin*: *isin t x* ⟹ *size t* = *Suc* (*size*(*delete x t*))
⟨*proof*⟩

lemma *delete_id_if_wbt_notin*: *wbt t* ⟹ ¬ *isin t x* ⟹ *delete x t* = *t*
⟨*proof*⟩

lemma *size_split_min*: *t* ≠ *Leaf* ⟹ *size t* = *Suc* (*size* (*snd* (*split_min t*)))
⟨*proof*⟩

lemma *size_del_max*: *t* ≠ *Leaf* ⟹ *size t* = *Suc*(*size*(*snd*(*del_max t*)))
⟨*proof*⟩

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**
Δ1 = 25 **and** Δ2 = 10 **and** Γ1 = 14 **and** Γ2 = 10

⟨*proof*⟩

lemma *wbt_insert*:
  wbt t ⟹ *wbt* (*insert x t*)
⟨*proof*⟩

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*⟩

lemma *wbt_balanceR*:
  assumes *wbt* (*Node l* (*a*, *n*) *r*) *wbt r′ size r* = *size r′* + 1
  shows *wbt* (*balanceR l a′ r′*)
⟨*proof*⟩

lemma *wbt_split_min*: *t* ≠ *Leaf* ⟹ *wbt t* ⟹ *wbt* (*snd* (*split_min t*))
⟨*proof*⟩

lemma *wbt_del_max*: *t* ≠ *Leaf* ⟹ *wbt t* ⟹ *wbt* (*snd* (*del_max t*))
⟨*proof*⟩

lemma *wbt_delete*: *wbt t* ⟹ *wbt* (*delete x t*)
⟨*proof*⟩

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*⟩

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.