

AVL Trees

Tobias Nipkow and Cornelia Pusch

December 12, 2009

Abstract

Two formalizations of AVL trees with room for extensions. The first formalization is monolithic and shorter, the second one in two stages, longer and a bit simpler. The final implementation is the same. If you are interested in developing this further, please contact <gerwin.klein@nicta.com.au>.

Contents

1	AVL Trees	2
1.1	Invariants and auxiliary functions	2
1.2	AVL interface and implementation	2
1.3	Correctness proof	3
1.3.1	Insertion maintains AVL balance	3
1.3.2	Correctness of insertion	4
1.3.3	Correctness of lookup	4
1.3.4	Insertion maintains order	5
2	AVL Trees in 2 Stages	5
2.1	Step 1: Pure binary and AVL trees	5
2.1.1	Auxiliary functions	5
2.1.2	AVL interface and simple implementation	6
2.1.3	Insertion maintains AVL balance	7
2.1.4	Correctness of insertion	7
2.1.5	Correctness of lookup	7
2.1.6	Insertion maintains order	8
2.2	Step 2: Binary and AVL trees with height information	8
2.2.1	Auxiliary functions	8
2.2.2	AVL interface and efficient implementation	8
2.2.3	Correctness proof	9

1 AVL Trees

```
theory AVL
imports Main
begin
```

This theory would be a nice candidate for structured Isar proof texts and for extensions (delete operation). At the moment only insertion is formalized.

```
datatype 'a tree = ET | MKT 'a "'a tree" "'a tree" nat
```

1.1 Invariants and auxiliary functions

```
primrec set_of :: "'a tree ⇒ 'a set"
where
"set_of ET = {}" |
"set_of (MKT n l r h) = insert n (set_of l ∪ set_of r)"
```

```
primrec height :: "'a tree ⇒ nat"
where
"height ET = 0" |
"height (MKT x l r h) = max (height l) (height r) + 1"
```

```
primrec avl :: "'a tree ⇒ bool"
where
"avl ET = True" |
"avl (MKT x l r h) =
((height l = height r ∨ height l = 1+height r ∨ height r = 1+height l) ∧
 h = max (height l) (height r) + 1 ∧ avl l ∧ avl r)"
```

```
primrec is_ord :: "('a::order) tree ⇒ bool"
where
"is_ord ET = True" |
"is_ord (MKT n l r h) =
((∀n' ∈ set_of l. n' < n) ∧ (∀n' ∈ set_of r. n < n')) ∧ is_ord l ∧ is_ord r)"
```

1.2 AVL interface and implementation

```
primrec is_in :: "('a::order) ⇒ 'a tree ⇒ bool"
where
"is_in k ET = False" |
"is_in k (MKT n l r h) = (if k = n then True else
                          if k < n then (is_in k l)
                          else (is_in k r))"
```

```
primrec ht :: "'a tree ⇒ nat"
where
"ht ET = 0" |
"ht (MKT x l r h) = h"
```

definition

```
mkt :: "'a ⇒ 'a tree ⇒ 'a tree ⇒ 'a tree" where
"mkt x l r = MKT x l r (max (ht l) (ht r) + 1)"
```

fun l_bal where

```
"l_bal(n, MKT ln ll lr h, r) =
  (if ht ll < ht lr
   then case lr of ET ⇒ ET (* impossible *)
           | MKT lrn lrl lrr lrh ⇒
             mkt lrn (mkt ln ll lrl) (mkt n lrr r)
   else mkt ln ll (mkt n lr r))"
```

fun r_bal where

```
"r_bal(n, l, MKT rn rl rr h) =
  (if ht rl > ht rr
   then case rl of ET ⇒ ET (* impossible *)
           | MKT rln rll rlr h ⇒ mkt rln (mkt n lrll) (mkt rn rlr rr)
   else mkt rn (mkt n l rl) rr)"
```

primrec insrt :: "'a::order ⇒ 'a tree ⇒ 'a tree"

where

```
"insrt x ET = MKT x ET ET 1" |
"insrt x (MKT n l r h) =
  (if x=n
   then MKT n l r h
   else if x<n
        then let l' = insrt x l; hl' = ht l'; hr = ht r
              in if hl' = 2+hr then l_bal(n,l',r)
                  else MKT n l' r (1 + max hl' hr)
        else let r' = insrt x r; hl = ht l; hr' = ht r'
              in if hr' = 2+hl then r_bal(n,l,r')
                  else MKT n l r' (1 + max hl hr'))"
```

1.3 Correctness proof

1.3.1 Insertion maintains AVL balance

```
declare Let_def [simp]
```

```
lemma [simp]: "avl t ⇒ ht t = height t"
<proof>
```

Lemmas about height after rotation

```
lemma height_l_bal:
```

```
"[ height l = height r + 2; avl l; avl r ]
 ⇒ height (l_bal(n,l,r)) = height r + 2 ∨
   height (l_bal(n,l,r)) = height r + 3"
<proof>
```

lemma *height_r_bal*:

```
"[[ height r = height l + 2; avl l; avl r ]]
  => height (r_bal(n,l,r)) = height l + 2 ∨
      height (r_bal(n,l,r)) = height l + 3"
⟨proof⟩
```

lemma [*simp*]: "height(mkt x l r) = max (height l) (height r) + 1"

⟨proof⟩

lemma *avl_mkt*: "[[avl l; avl r;

```
height l = height r ∨ height l = height r + 1 ∨ height r = height l + 1 ]]
```

```
=> avl(mkt x l r)"
```

⟨proof⟩

lemma *avl_l_bal*:

```
"[[ avl l; avl r; height l = height r + 2]]
  => avl (l_bal(n, l, r))"
```

⟨proof⟩

lemma *avl_r_bal*:

```
"[[ avl l; avl r; height r = height l + 2]] => avl (r_bal(n, l, r))"
```

⟨proof⟩

Insertion maintains the AVL property:

theorem *avl_insrt_aux*:

```
"avl t => avl(insrt x t) ∧
  (height (insrt x t) = height t ∨ height (insrt x t) = height t + 1)"
```

⟨proof⟩

lemmas *avl_insrt* = *avl_insrt_aux*[*THEN* *conjunct1*]

1.3.2 Correctness of insertion

lemma *set_of_l_bal*: "height l = height r + 2 =>

```
set_of (l_bal(x, l, r)) = insert x (set_of l ∪ set_of r)"
```

⟨proof⟩

lemma *set_of_r_bal*: "height r = height l + 2 =>

```
set_of (r_bal(x, l, r)) = insert x (set_of l ∪ set_of r)"
```

⟨proof⟩

Correctness of *insrt*:

theorem *set_of_insrt*:

```
"avl t => set_of(insrt x t) = insert x (set_of t)"
```

⟨proof⟩

1.3.3 Correctness of lookup

theorem *is_in_correct*: "is_ord t => is_in k t = (k : set_of t)"

⟨proof⟩

1.3.4 Insertion maintains order

```
lemma is_ord_l_bal:
  "[[ is_ord(MKT x l r h); height l = height r + 2 ] ] ==> is_ord(l_bal(x,l,r))"
  <proof>
```

```
lemma is_ord_r_bal:
  "[[ is_ord(MKT x l r h); height r = height l + 2 ] ] ==> is_ord(r_bal(x,l,r))"
  <proof>
```

If the order is linear, *insrt* maintains the order:

```
theorem is_ord_insrt:
  "[[ avl t; is_ord t ] ] ==> is_ord(insrt (x::'a::linorder) t)"
  <proof>
```

end

2 AVL Trees in 2 Stages

```
theory AVL2
imports Main
begin
```

This development of AVL trees leads to the same implementation as the monolithic one (in theory AVL) but via an intermediate abstraction: AVL trees where the height is recomputed rather than stored in the tree. This two-stage development is longer than the monolithic one but each individual step is simpler. It should really be viewed as a blueprint for the development of data structures where some of the fields contain redundant information (for efficiency reasons).

2.1 Step 1: Pure binary and AVL trees

The basic formulation of AVL trees builds on pure binary trees and recomputes all height information whenever it is required. This simplifies the correctness proofs.

```
datatype 'a tree0 = ET0 | MKT0 'a "'a tree0" "'a tree0"
```

2.1.1 Auxiliary functions

```
primrec height :: "'a tree0 => nat" where
  "height ET0 = 0"
  | "height (MKT0 n l r) = 1 + max (height l) (height r)"
```

```
primrec set_of :: "'a tree0 => 'a set" where
  "set_of ET0 = {}"
  | "set_of (MKT0 n l r) = insert n (set_of l ∪ set_of r)"
```

```

primrec is_ord :: "('a::preorder) tree0 ⇒ bool" where
  "is_ord ET0 = True"
| "is_ord (MKT0 n l r) =
  ((∀n'∈ set_of l. n' < n) ∧ (∀n'∈ set_of r. n < n') ∧ is_ord l ∧ is_ord r)"

```

```

primrec is_bal :: "'a tree0 ⇒ bool" where
  "is_bal ET0 = True"
| "is_bal (MKT0 n l r) =
  ((height l = height r ∨ height l = 1+height r ∨ height r = 1+height l) ∧
  is_bal l ∧ is_bal r)"

```

2.1.2 AVL interface and simple implementation

```

primrec is_in0 :: "('a::preorder) ⇒ 'a tree0 ⇒ bool" where
  "is_in0 k ET0 = False"
| "is_in0 k (MKT0 n l r) = (if k = n then True else
  if k < n then (is_in0 k l)
  else (is_in0 k r))"

```

```

primrec l_bal0 :: "'a ⇒ 'a tree0 ⇒ 'a tree0 ⇒ 'a tree0" where
  "l_bal0 n (MKT0 ln ll lr) r =
  (if height ll < height lr
  then case lr of ET0 ⇒ ET0 (* impossible *)
  | MKT0 lrn lrl lrr ⇒ MKT0 lrn (MKT0 ln ll lrl) (MKT0 n lrr r)
  else MKT0 ln ll (MKT0 n lr r))"

```

```

primrec r_bal0 :: "'a ⇒ 'a tree0 ⇒ 'a tree0 ⇒ 'a tree0" where
  "r_bal0 n l (MKT0 rn rl rr) =
  (if height rl > height rr
  then case rl of ET0 ⇒ ET0 (* impossible *)
  | MKT0 rln rll rlr ⇒ MKT0 rln (MKT0 n l rll) (MKT0 rn rlr rr)
  else MKT0 rn (MKT0 n l rl) rr)"

```

```

primrec insrt0 :: "'a::preorder ⇒ 'a tree0 ⇒ 'a tree0" where
  "insrt0 x ET0 = MKT0 x ET0 ET0"
| "insrt0 x (MKT0 n l r) =
  (if x=n
  then MKT0 n l r
  else if x < n
  then let l' = insrt0 x l
  in if height l' = 2+height r
  then l_bal0 n l' r
  else MKT0 n l' r
  else let r' = insrt0 x r
  in if height r' = 2+height l
  then r_bal0 n l r'
  else MKT0 n l r'"

```

2.1.3 Insertion maintains AVL balance

lemma *height_l_bal*:

```
"height l = height r + 2
  => height (l_bal_0 n l r) = height r + 2 ∨
     height (l_bal_0 n l r) = height r + 3"
⟨proof⟩
```

lemma *height_r_bal*:

```
"height r = height l + 2
  => height (r_bal_0 n l r) = height l + 2 ∨
     height (r_bal_0 n l r) = height l + 3"
⟨proof⟩
```

lemma *height_insrt*:

```
"is_bal t
  => height (insrt_0 x t) = height t ∨ height (insrt_0 x t) = height t + 1"
⟨proof⟩
```

lemma *is_bal_l_bal*:

```
"is_bal l => is_bal r => height l = height r + 2 => is_bal (l_bal_0 n l r)"
⟨proof⟩
```

lemma *is_bal_r_bal*:

```
"is_bal l => is_bal r => height r = height l + 2 => is_bal (r_bal_0 n l r)"
⟨proof⟩
```

theorem *is_bal_insrt*:

```
"is_bal t => is_bal (insrt_0 x t)"
⟨proof⟩
```

2.1.4 Correctness of insertion

lemma *set_of_l_bal*: "height l = height r + 2 =>

```
set_of (l_bal_0 x l r) = insert x (set_of l ∪ set_of r)"
⟨proof⟩
```

lemma *set_of_r_bal*: "height r = height l + 2 =>

```
set_of (r_bal_0 x l r) = insert x (set_of l ∪ set_of r)"
⟨proof⟩
```

theorem *set_of_insrt*:

```
"set_of (insrt_0 x t) = insert x (set_of t)"
⟨proof⟩
```

2.1.5 Correctness of lookup

theorem *is_in_correct*: "is_ord t => is_in_0 k t = (k : set_of t)"

```
⟨proof⟩
```

2.1.6 Insertion maintains order

```
lemma is_ord_l_bal:
  "is_ord (MKT0 x l r)  $\implies$  height l = Suc (Suc (height r))  $\implies$ 
  is_ord (l_bal0 x l r)"
  <proof>
```

```
lemma is_ord_r_bal:
  "is_ord (MKT0 x l r)  $\implies$  height r = height l + 2  $\implies$ 
  is_ord (r_bal0 x l r)"
  <proof>
```

If the order is linear, insrt_0 maintains the order:

```
theorem is_ord_insrt:
  "is_ord t  $\implies$  is_ord (insrt0 (x::'a::linorder) t)"
  <proof>
```

2.2 Step 2: Binary and AVL trees with height information

```
datatype 'a tree = ET | MKT 'a "'a tree" "'a tree" nat
```

2.2.1 Auxiliary functions

```
primrec erase :: "'a tree  $\Rightarrow$  'a tree0" where
  "erase ET = ET0"
  | "erase (MKT x l r h) = MKT0 x (erase l) (erase r)"
```

```
primrec hinv :: "'a tree  $\Rightarrow$  bool" where
  "hinv ET  $\longleftrightarrow$  True"
  | "hinv (MKT x l r h)  $\longleftrightarrow$  h = 1 + max (height (erase l)) (height (erase r))
     $\wedge$  hinv l  $\wedge$  hinv r"
```

```
definition avl :: "'a tree  $\Rightarrow$  bool" where
  "avl t  $\longleftrightarrow$  is_bal (erase t)  $\wedge$  hinv t"
```

2.2.2 AVL interface and efficient implementation

```
primrec is_in :: "('a::preorder)  $\Rightarrow$  'a tree  $\Rightarrow$  bool" where
  "is_in k ET  $\longleftrightarrow$  False"
  | "is_in k (MKT n l r h)  $\longleftrightarrow$  (if k = n then True else
    if k < n then (is_in k l)
    else (is_in k r))"
```

```
primrec ht :: "'a tree  $\Rightarrow$  nat" where
  "ht ET = 0"
  | "ht (MKT x l r h) = h"
```

```
definition mkt :: "'a  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree" where
  "mkt x l r = MKT x l r (max (ht l) (ht r) + 1)"
```

primrec l_bal :: "'a ⇒ 'a tree ⇒ 'a tree ⇒ 'a tree" where

```
"l_bal n (MKT ln ll lr h) r =
  (if ht ll < ht lr
   then case lr of ET ⇒ ET (* impossible *)
             | MKT lrn lrl lrr lrh ⇒
                 mkt lrn (mkt ln ll lrl) (mkt n lrr r)
   else mkt ln ll (mkt n lr r))"
```

primrec r_bal :: "'a ⇒ 'a tree ⇒ 'a tree ⇒ 'a tree" where

```
"r_bal n l (MKT rn rl rr h) =
  (if ht rl > ht rr
   then case rl of ET ⇒ ET (* impossible *)
             | MKT rln rll rlr h ⇒ mkt rln (mkt n l rll) (mkt rn rlr rr)
   else mkt rn (mkt n l rl) rr)"
```

primrec insrt :: "'a::preorder ⇒ 'a tree ⇒ 'a tree" where

```
"insrt x ET = MKT x ET ET 1"
| "insrt x (MKT n l r h) =
  (if x=n
   then MKT n l r h
   else if x<n
        then let l' = insrt x l; hl' = ht l'; hr = ht r
              in if hl' = 2+hr then l_bal n l' r
                 else MKT n l' r (1 + max hl' hr)
        else let r' = insrt x r; hl = ht l; hr' = ht r'
              in if hr' = 2+hl then r_bal n l r'
                 else MKT n l r' (1 + max hl hr'))"
```

2.2.3 Correctness proof

The auxiliary functions are implemented correctly:

lemma height_hinv: "hinv t ⇒ ht t = height (erase t)"

<proof>

lemma erase_mkt: "erase (mkt n l r) = MKT₀ n (erase l) (erase r)"

<proof>

lemma erase_l_bal:

```
"hinv l ⇒ hinv r ⇒ height (erase l) = height(erase r) + 2 ⇒
erase (l_bal n l r) = l_bal0 n (erase l) (erase r)"
```

<proof>

lemma erase_r_bal:

```
"hinv l ⇒ hinv r ⇒ height(erase r) = height(erase l) + 2 ⇒
erase (r_bal n l r) = r_bal0 n (erase l) (erase r)"
```

<proof>

Function *insrt* maintains the invariant:

lemma hinv_mkt: "hinv l ⇒ hinv r ⇒ hinv (mkt x l r)"

<proof>

lemma *hinv_l_bal*:

"*hinv l* \implies *hinv r* \implies *height(erase l)* = *height(erase r)* + 2 \implies
hinv (l_bal n l r)"

<proof>

lemma *hinv_r_bal*:

"*hinv l* \implies *hinv r* \implies *height(erase r)* = *height(erase l)* + 2 \implies
hinv (r_bal n l r)"

<proof>

theorem *hinv_insrt*: "*hinv t* \implies *hinv (insrt x t)*"

<proof>

Function *insrt* implements *insrt₀*:

lemma *erase_insrt*: "*hinv t* \implies *erase (insrt x t)* = *insrt₀ x (erase t)*"

<proof>

Function *insrt* meets its spec:

corollary "*avl t* \implies *set_of (erase (insrt x t))* = *insert x (set_of (erase t))*"

<proof>

Function *insrt* preserves the invariants:

corollary "*avl t* \implies *avl (insrt x t)*"

<proof>

corollary

"*avl t* \implies *is_ord (erase t)* \implies *is_ord (erase (insrt (x::'a::linorder) t))*"

<proof>

Function *is_in* implements *is_in*:

theorem *is_in*: "*is_in x t* = *is_in₀ x (erase t)*"

<proof>

Function *is_in* meets its spec:

corollary "*is_ord (erase t)* \implies *is_in x t* \iff $x \in$ *set_of (erase t)*"

<proof>

end