

# 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>](mailto:gerwin.klein@nicta.com.au).

## Contents

<b>1</b>	<b>AVL Trees</b>	<b>2</b>
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 . . . . .	5
1.3.3	Correctness of lookup . . . . .	5
1.3.4	Insertion maintains order . . . . .	5
<b>2</b>	<b>AVL Trees in 2 Stages</b>	<b>6</b>
2.1	Step 1: Pure binary and AVL trees . . . . .	6
2.1.1	Auxiliary functions . . . . .	6
2.1.2	AVL interface and simple implementation . . . . .	7
2.1.3	Insertion maintains AVL balance . . . . .	8
2.1.4	Correctness of insertion . . . . .	9
2.1.5	Correctness of lookup . . . . .	9
2.1.6	Insertion maintains order . . . . .	10
2.2	Step 2: Binary and AVL trees with height information . . . . .	10
2.2.1	Auxiliary functions . . . . .	10
2.2.2	AVL interface and efficient implementation . . . . .	10
2.2.3	Correctness proof . . . . .	11

# 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  $\Rightarrow$  'a set"
where
"set_of ET = {}" |
"set_of (MKT n l r h) = insert n (set_of l  $\cup$  set_of r)"
```

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

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

```
primrec is_ord :: "('a::order) tree  $\Rightarrow$  bool"
where
"is_ord ET = True" |
"is_ord (MKT n l r h) =
(( $\forall n' \in$  set_of l.  $n' < n$ )  $\wedge$  ( $\forall n' \in$  set_of r.  $n < n'$ )  $\wedge$  is_ord l  $\wedge$  is_ord r)"
```

## 1.2 AVL interface and implementation

```
primrec is_in :: "('a::order)  $\Rightarrow$  'a tree  $\Rightarrow$  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  $\Rightarrow$  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"
by (induct t) simp_all
```

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"
```

```
apply (cases l)
```

```
apply (auto simp add:max_def mkt_def split:tree.split split_if_asm)
```

done

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"
```

apply(cases r)

apply (auto simp add:max\_def mkt\_def split:tree.split split\_if\_asm)

done

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

by (simp add: mkt\_def)

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)"

by (auto simp add:max\_def mkt\_def)

lemma avl\_l\_bal:

```
"[ avl l; avl r; height l = height r + 2 ]
```

```
  ⇒ avl (l_bal(n, l, r))"
```

apply(cases l)

apply (auto simp:avl\_mkt max\_def)

apply (auto simp:avl\_mkt max\_def split:tree.split)

done

lemma avl\_r\_bal:

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

apply(cases r)

apply (auto simp:avl\_mkt max\_def)

apply (auto simp:avl\_mkt max\_def split:tree.split)

done

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)"
```

apply (induct "t")

apply simp

apply (rename\_tac n t1 t2 h)

apply (case\_tac "x=n")

apply simp

apply (case\_tac "x<n")

apply (case\_tac "height (insrt x t1) = height t2 + 2")

apply(rule conjI)

apply (simp add:avl\_l\_bal)

apply (frule\_tac n = n in height\_l\_bal) apply simp apply simp

```

  apply (simp add: max_def) apply arith
  apply(rule conjI)
  apply fastsimp
  apply (simp add: max_def) apply arith
  apply (case_tac "height (insrt x t2) = height t1 + 2")
  apply(rule conjI)
  apply (simp add:avl_r_bal)
  apply (frule_tac n = n in height_r_bal) apply simp apply simp
  apply (simp add: max_def) apply arith
  apply(rule conjI)
  apply fastsimp
  apply (simp add: max_def) apply arith
done

```

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

### 1.3.2 Correctness of insertion

```

lemma set_of_l_bal: "height l = height r + 2  $\implies$ 
  set_of (l_bal(x, l, r)) = insert x (set_of l  $\cup$  set_of r)"
  apply(cases l)
  apply (auto simp:max_def mkt_def split:tree.splits)
done

```

```

lemma set_of_r_bal: "height r = height l + 2  $\implies$ 
  set_of (r_bal(x, l, r)) = insert x (set_of l  $\cup$  set_of r)"
  apply(cases r)
  apply (auto simp:max_def mkt_def split:tree.splits)
done

```

Correctness of *insrt*:

```

theorem set_of_insrt:
  "avl t  $\implies$  set_of(insrt x t) = insert x (set_of t)"
  apply (induct t)
  apply simp
  apply(force simp: avl_insrt set_of_l_bal set_of_r_bal)
done

```

### 1.3.3 Correctness of lookup

```

theorem is_in_correct: "is_ord t  $\implies$  is_in k t = (k : set_of t)"
  by (induct "t") auto

```

### 1.3.4 Insertion maintains order

```

lemma is_ord_l_bal:
  "[[ is_ord(MKT x l r h); height l = height r + 2 ]  $\implies$  is_ord(l_bal(x,l,r))"
  apply (cases l)
  apply (auto simp: mkt_def split:tree.splits intro: order_less_trans)
done

```

```

lemma is_ord_r_bal:
  "[ is_ord(MKT x l r h); height r = height l + 2 ] ==> is_ord(r_bal(x,l,r))"
apply (cases r)
apply (auto simp: mkt_def split:tree.splits intro: order_less_trans)
done

```

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)"
apply (induct t)
  apply simp
apply (simp add:is_ord_l_bal is_ord_r_bal avl_insrt set_of_insrt
              linorder_not_less order_neq_le_trans)
done

```

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  $MKT_0$  n l r')"

### 2.1.3 Insertion maintains AVL balance

lemma height\_l\_bal:

"height l = height r + 2  
 $\implies$  height (l\_bal<sub>0</sub> n l r) = height r + 2  $\vee$   
height (l\_bal<sub>0</sub> n l r) = height r + 3"  
by (cases l) (auto split: tree<sub>0</sub>.split split\_if\_asm)

lemma height\_r\_bal:

"height r = height l + 2  
 $\implies$  height (r\_bal<sub>0</sub> n l r) = height l + 2  $\vee$   
height (r\_bal<sub>0</sub> n l r) = height l + 3"  
by (cases r) (auto split: tree<sub>0</sub>.split split\_if\_asm)

lemma height\_insrt:

"is\_bal t  
 $\implies$  height (insrt<sub>0</sub> x t) = height t  $\vee$  height (insrt<sub>0</sub> x t) = height t + 1"

proof (induct t)

case ET<sub>0</sub> show ?case by simp

next

case (MKT<sub>0</sub> n t1 t2) then show ?case proof (cases "x < n")

case True show ?thesis

proof (cases "height (insrt<sub>0</sub> x t1) = height t2 + 2")

case True with height\_l\_bal [of \_ \_ n]

have "height (l\_bal<sub>0</sub> n (insrt<sub>0</sub> x t1) t2) =

height t2 + 2  $\vee$  height (l\_bal<sub>0</sub> n (insrt<sub>0</sub> x t1) t2) = height t2 + 3" by simp

with 'x < n' MKT<sub>0</sub> show ?thesis by auto

next

case False with 'x < n' MKT<sub>0</sub> show ?thesis by auto

qed

next

case False show ?thesis

proof (cases "height (insrt<sub>0</sub> x t2) = height t1 + 2")

case True with height\_r\_bal [of \_ \_ n]

have "height (r\_bal<sub>0</sub> n t1 (insrt<sub>0</sub> x t2)) = height t1 + 2  $\vee$

height (r\_bal<sub>0</sub> n t1 (insrt<sub>0</sub> x t2)) = height t1 + 3" by simp

with ' $\neg$  x < n' MKT<sub>0</sub> show ?thesis by auto

next

case False with ' $\neg$  x < n' MKT<sub>0</sub> show ?thesis by auto

qed

qed

qed

lemma is\_bal\_l\_bal:

"is\_bal l  $\implies$  is\_bal r  $\implies$  height l = height r + 2  $\implies$  is\_bal (l\_bal<sub>0</sub> n l r)"

by (cases l) (auto, auto split: tree<sub>0</sub>.split) — separating the two auto's is just for speed

```

lemma is_bal_r_bal:
  "is_bal l  $\implies$  is_bal r  $\implies$  height r = height l + 2  $\implies$  is_bal (r_bal_0 n l r)"
  by (cases r) (auto, auto split: tree_0.split) — separating the two auto's is just for speed

```

```

theorem is_bal_insrt:
  "is_bal t  $\implies$  is_bal(insrt_0 x t)"
proof (induct t)
  case ET_0 show ?case by simp
next
  case (MKT_0 n t1 t2) show ?case proof (cases "x < n")
    case True show ?thesis
      proof (cases "height (insrt_0 x t1) = height t2 + 2")
        case True with 'x < n' MKT_0 show ?thesis
          by (simp add: is_bal_l_bal)
        next
          case False with 'x < n' MKT_0 show ?thesis
            using height_insrt [of t1 x] by auto
      qed
    next
      case False show ?thesis
        proof (cases "height (insrt_0 x t2) = height t1 + 2")
          case True with ' $\neg$  x < n' MKT_0 show ?thesis
            by (simp add: is_bal_r_bal)
          next
            case False with ' $\neg$  x < n' MKT_0 show ?thesis
              using height_insrt [of t2 x] by auto
        qed
      qed
    qed
  qed
qed

```

#### 2.1.4 Correctness of insertion

```

lemma set_of_l_bal: "height l = height r + 2  $\implies$ 
  set_of (l_bal_0 x l r) = insert x (set_of l  $\cup$  set_of r)"
  by (cases l) (auto split: tree_0.splits)

```

```

lemma set_of_r_bal: "height r = height l + 2  $\implies$ 
  set_of (r_bal_0 x l r) = insert x (set_of l  $\cup$  set_of r)"
  by (cases r) (auto split: tree_0.splits)

```

```

theorem set_of_insrt:
  "set_of (insrt_0 x t) = insert x (set_of t)"
  by (induct t) (auto simp add: set_of_l_bal set_of_r_bal Let_def)

```

#### 2.1.5 Correctness of lookup

```

theorem is_in_correct: "is_ord t  $\implies$  is_in_0 k t = (k : set_of t)"
  by (induct t) (auto simp add: less_le_not_le)

```

### 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)"
  by (cases l) (auto split: tree0.splits intro: order_less_trans)
```

```
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)"
  by (cases r) (auto split: tree0.splits intro: order_less_trans)
```

If the order is linear,  $\text{insrt}_0$  maintains the order:

```
theorem is_ord_insrt:
  "is_ord t  $\implies$  is_ord (insrt0 (x::'a::linorder) t)"
  by (induct t) (simp_all add: is_ord_l_bal is_ord_r_bal set_of_insrt
    linorder_not_less order_neq_le_trans Let_def)
```

## 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)"
  by (induct t) simp_all

```

```

lemma erase_mkt: "erase (mkt n l r) = MKT0 n (erase l) (erase r)"
  by (simp add: mkt_def)

```

```

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)"
  by (cases l) (simp_all add: height_hinv erase_mkt split: tree.split)

```

```

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)"
  by (cases r) (simp_all add: height_hinv erase_mkt split: tree.split)

```

Function `insrt` maintains the invariant:

**lemma** *hinv\_mkt*: "hinv l  $\implies$  hinv r  $\implies$  hinv (mkt x l r)"  
 by (simp add: height\_hinv mkt\_def)

**lemma** *hinv\_l\_bal*:  
 "hinv l  $\implies$  hinv r  $\implies$  height(erase l) = height(erase r) + 2  $\implies$   
 hinv (l\_bal n l r)"  
 by (cases l) (auto simp add: hinv\_mkt split: tree.splits)

**lemma** *hinv\_r\_bal*:  
 "hinv l  $\implies$  hinv r  $\implies$  height(erase r) = height(erase l) + 2  $\implies$   
 hinv (r\_bal n l r)"  
 by (cases r) (auto simp add: hinv\_mkt split: tree.splits)

**theorem** *hinv\_insrt*: "hinv t  $\implies$  hinv (insrt x t)"  
 by (induct t) (simp\_all add: Let\_def height\_hinv hinv\_l\_bal hinv\_r\_bal)

Function *insrt* implements *insrt<sub>0</sub>*:

**lemma** *erase\_insrt*: "hinv t  $\implies$  erase (insrt x t) = *insrt<sub>0</sub>* x (erase t)"  
 by (induct t) (simp\_all add: Let\_def hinv\_insrt height\_hinv erase\_l\_bal erase\_r\_bal)

Function *insrt* meets its spec:

**corollary** "avl t  $\implies$  set\_of (erase (insrt x t)) = insert x (set\_of (erase t))"  
 by (simp add: avl\_def erase\_insrt set\_of\_insrt)

Function *insrt* preserves the invariants:

**corollary** "avl t  $\implies$  avl (insrt x t)"  
 by (simp add: hinv\_insrt avl\_def erase\_insrt is\_bal\_insrt)

**corollary**  
 "avl t  $\implies$  is\_ord (erase t)  $\implies$  is\_ord (erase (insrt (x::'a::linorder) t))"  
 by (simp add: avl\_def erase\_insrt is\_ord\_insrt)

Function *is\_in* implements *is\_in*:

**theorem** *is\_in*: "is\_in x t = is\_in<sub>0</sub> x (erase t)"  
 by (induct t) simp\_all

Function *is\_in* meets its spec:

**corollary** "is\_ord (erase t)  $\implies$  is\_in x t  $\iff$  x  $\in$  set\_of (erase t)"  
 by (simp add: is\_in is\_in\_correct)

end