AVL Trees

Tobias Nipkow and Cornelia Pusch

March 17, 2025

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

theory *AVL*
imports *Main*
begin

This is a monolithic formalization of AVL trees.

1.1 AVL tree type definition

```
datatype (set_of: 'a) tree = ET |  MKT 'a "'a tree" "'a tree" nat
```

1.2 Invariants and auxiliary functions

```
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 = height r + 1 ∨ height r = height l + 1) ∧
  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)"
```

1.3 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 mkt_bal_l where
"mkt_bal_l n l r = (
  if ht l = ht r + 2 then (case l of
    MKT ln ll lr _ ⇒ (if ht ll < ht lr
    then case lr of
```

```
      MKT lrn lrl lrr _ ⇒ mkt lrn (mkt ln ll lrl) (mkt n lrr r)
    else mkt ln ll (mkt n lr r)))
  else mkt n l r
)"
```

fun *mkt_bal_r* **where**

```
"mkt_bal_r n l r = (
  if ht r = ht l + 2 then (case r of
    MKT rn rl rr _ ⇒ (if ht rl > ht rr
    then case rl of
      MKT rln rll rlr _ ⇒ mkt rln (mkt n l rll) (mkt rn rlr rr)
    else mkt rn (mkt n l rl) rr))
  else mkt n l r
)"
```

primrec *insert :: "'a::order ⇒ 'a tree ⇒ 'a tree"* **where**

```
"insert x ET = MKT x ET ET 1" |
"insert x (MKT n l r h) =
   (if x=n
    then MKT n l r h
    else if x<n
      then mkt_bal_l n (insert x l) r
      else mkt_bal_r n l (insert x r))"
```

fun *delete_max* **where**

```
"delete_max (MKT n l ET h) = (n,l)" |
"delete_max (MKT n l r h) = (
  let (n',r') = delete_max r in
  (n',mkt_bal_l n l r'))"
```

lemmas *delete_max_induct = delete_max.induct[case_names ET MKT]*

fun *delete_root* **where**

```
"delete_root (MKT n ET r h) = r" |
"delete_root (MKT n l ET h) = l" |
"delete_root (MKT n l r h) =
  (let (new_n, l') = delete_max l in
      mkt_bal_r new_n l' r
  )"
```

lemmas *delete_root_cases = delete_root.cases[case_names ET_t MKT_ET MKT_MKT]*

primrec *delete :: "'a::order ⇒ 'a tree ⇒ 'a tree"* **where**

```
"delete _ ET = ET" |
"delete x (MKT n l r h) = (
   if x = n then delete_root (MKT n l r h)
   else if x < n then
        let l' = delete x l in
        mkt_bal_r n l' r
```

```
  else
       let r’ = delete x r in
       mkt_bal_l n l r’
  )"
```

1.4 Correctness proof

1.4.1 Insertion maintains AVL balance

declare *Let_def [simp]*

lemma *[simp]: "avl t ⟹ ht t = height t"*
⟨*proof*⟩

lemma *height_mkt_bal_l:*
```
  "⟦ height l = height r + 2; avl l; avl r ⟧ ⟹
   height (mkt_bal_l n l r) = height r + 2 ∨
   height (mkt_bal_l n l r) = height r + 3"
```
⟨*proof*⟩

lemma *height_mkt_bal_r:*
```
  "⟦ height r = height l + 2; avl l; avl r ⟧ ⟹
   height (mkt_bal_r n l r) = height l + 2 ∨
   height (mkt_bal_r 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 *height_mkt_bal_l2:*
```
  "⟦ avl l; avl r; height l ≠ height r + 2 ⟧ ⟹
   height (mkt_bal_l n l r) = (1 + max (height l) (height r))"
```
⟨*proof*⟩

lemma *height_mkt_bal_r2:*
```
  "⟦ avl l;  avl r;  height r ≠ height l + 2 ⟧ ⟹
   height (mkt_bal_r n l r) = (1 + max (height l) (height r))"
```
⟨*proof*⟩

lemma *avl_mkt_bal_l:*
  assumes *"avl l" "avl r"* **and** *"height l = height r ∨ height l = height r + 1*
    ∨ height r = height l + 1 ∨ height l = height r + 2"
  shows *"avl(mkt_bal_l n l r)"*
⟨*proof*⟩

lemma *avl_mkt_bal_r:*
  assumes *"avl l"* **and** *"avl r"* **and** *"height l = height r ∨ height l = height r + 1 ∨ height r = height l + 1 ∨ height r = height l + 2"*
  shows *"avl(mkt_bal_r n l r)"*
⟨*proof*⟩

Insertion maintains the AVL property:

theorem *avl_insert_aux:*
  assumes *"avl t"*
  shows *"avl(insert x t)"*
  "(height (insert x t) = height t ∨ height (insert x t) = height t + 1)"
⟨*proof*⟩

lemmas *avl_insert = avl_insert_aux(1)*

1.4.2 Deletion maintains AVL balance

lemma *avl_delete_max:*
  assumes *"avl x"* **and** *"x ≠ ET"*
  shows *"avl (snd (delete_max x))"* *"height x = height(snd (delete_max x)) ∨ height x = height(snd (delete_max x)) + 1"*
⟨*proof*⟩

lemma *avl_delete_root:*
  assumes *"avl t"* **and** *"t ≠ ET"*
  shows *"avl(delete_root t)"*
⟨*proof*⟩

lemma *height_delete_root:*
  assumes *"avl t"* **and** *"t ≠ ET"*
  shows *"height t = height(delete_root t) ∨ height t = height(delete_root t) + 1"*
⟨*proof*⟩

Deletion maintains the AVL property:

theorem *avl_delete_aux:*
  assumes *"avl t"*
  shows *"avl(delete x t)"* **and** *"height t = (height (delete x t)) ∨ height t = height (delete x t) + 1"*
⟨*proof*⟩

lemmas *avl_delete = avl_delete_aux(1)*

1.4.3 Correctness of insertion

lemma *set_of_mkt_bal_l:*
  "⟦ avl l; avl r ⟧ ⟹ set_of (mkt_bal_l n l r) = Set.insert n (set_of l ∪ set_of r)"
⟨*proof*⟩

lemma *set_of_mkt_bal_r:*
"⟦ avl l; avl r ⟧ ⟹
set_of (mkt_bal_r n l r) = Set.insert n (set_of l ∪ set_of r)"
⟨*proof*⟩

Correctness of *AVL.insert*:

theorem *set_of_insert:*
"avl t ⟹ set_of(insert x t) = Set.insert x (set_of t)"
⟨*proof*⟩

1.4.4 Correctness of deletion

fun *rightmost_item :: "'a tree ⇒ 'a"* **where**
"rightmost_item (MKT n l ET h) = n" |
"rightmost_item (MKT n l r h) = rightmost_item r"

lemma *avl_dist:*
"⟦ avl(MKT n l r h); is_ord(MKT n l r h); x ∈ set_of l ⟧ ⟹
x ∉ set_of r"
⟨*proof*⟩

lemma *avl_dist2:*
"⟦ avl(MKT n l r h); is_ord(MKT n l r h); x ∈ set_of l ∨ x ∈ set_of r ⟧ ⟹
x ≠ n"
⟨*proof*⟩

lemma *ritem_in_rset:* "r ≠ ET ⟹ rightmost_item r ∈ set_of r"
⟨*proof*⟩

lemma *ritem_greatest_in_rset:*
"⟦ r ≠ ET; is_ord r ⟧ ⟹
∀x. x ∈ set_of r ⟶ x ≠ rightmost_item r ⟶ x < rightmost_item r"
⟨*proof*⟩

lemma *ritem_not_in_ltree:*
"⟦ avl(MKT n l r h); is_ord(MKT n l r h); r ≠ ET ⟧ ⟹
rightmost_item r ∉ set_of l"
⟨*proof*⟩

lemma *set_of_delete_max:*
"⟦ avl t; is_ord t; t≠ET ⟧ ⟹
set_of (snd(delete_max t)) = (set_of t) - {rightmost_item t}"
⟨*proof*⟩

lemma *fst_delete_max_eq_ritem:*
"t≠ET ⟹ fst(delete_max t) = rightmost_item t"
⟨*proof*⟩

lemma *set_of_delete_root:*
  assumes *"t = MKT n l r h"* **and** *"avl t"* **and** *"is_ord t"*
  shows *"set_of (delete_root t) = (set_of t) - {n}"*
⟨*proof*⟩

Correctness of *delete*:

theorem *set_of_delete:*
  "⟦ avl t; is_ord t ⟧ ⟹ set_of (delete x t) = (set_of t) - {x}"
⟨*proof*⟩

1.4.5 Correctness of lookup

theorem *is_in_correct: "is_ord t ⟹ is_in k t = (k : set_of t)"*
⟨*proof*⟩

1.4.6 Insertion maintains order

lemma *is_ord_mkt_bal_l:*
  "is_ord(MKT n l r h) ⟹ is_ord (mkt_bal_l n l r)"
⟨*proof*⟩

lemma *is_ord_mkt_bal_r: "is_ord(MKT n l r h) ⟹ is_ord (mkt_bal_r n l r)"*
⟨*proof*⟩

If the order is linear, *AVL.insert* maintains the order:

theorem *is_ord_insert:*
  "⟦ avl t; is_ord t ⟧ ⟹ is_ord(insert (x::'a::linorder) t)"
⟨*proof*⟩

1.4.7 Deletion maintains order

lemma *is_ord_delete_max:*
  "⟦ avl t; is_ord t; t≠ET ⟧ ⟹ is_ord(snd(delete_max t))"
⟨*proof*⟩

lemma *is_ord_delete_root:*
  assumes *"avl t"* **and** *"is_ord t"* **and** *"t ≠ ET"*
  shows *"is_ord (delete_root t)"*
⟨*proof*⟩

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

theorem *is_ord_delete:*
  "⟦ avl t; is_ord t ⟧ ⟹ is_ord (delete x 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 theorey AVL) but via an intermediate abstraction: AVL trees where the height is recomputed rather than stored in the tree. This two-stage devlopment 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 (set_of: '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 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_bal0 n l r) = height r + 2 ∨
      height (l_bal0 n l r)  = height r + 3"
  ⟨proof⟩
```

```
lemma height_r_bal:
 "height r = height l + 2
  ⟹ height (r_bal0 n l r) = height l + 2 ∨
      height (r_bal0 n l r) = height l + 3"
  ⟨proof⟩
```

```
lemma height_insrt:
 "is_bal t
  ⟹ height (insrt0 x t) = height t ∨ height (insrt0 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_bal0 n l r)"
  ⟨proof⟩
```

lemma *is_bal_r_bal:*
  "is_bal l $\Longrightarrow$ is_bal r $\Longrightarrow$ height r = height l + 2 $\Longrightarrow$ is_bal (r_bal$_0$ n l r)"
  ⟨*proof*⟩

theorem *is_bal_insrt:*
  "is_bal t $\Longrightarrow$ is_bal(insrt$_0$ x t)"
⟨*proof*⟩

2.1.4 Correctness of insertion

lemma *set_of_l_bal:* "height l = height r + 2 $\Longrightarrow$
  set_of (l_bal$_0$ x l r) = insert x (set_of l $\cup$ set_of r)"
  ⟨*proof*⟩

lemma *set_of_r_bal:* "height r = height l + 2 $\Longrightarrow$
  set_of (r_bal$_0$ x l r) = insert x (set_of l $\cup$ 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 $\Longrightarrow$ is_in$_0$ k t = (k : set_of t)"
  ⟨*proof*⟩

2.1.6 Insertion maintains order

lemma *is_ord_l_bal:*
 "is_ord (MKT$_0$ x l r) $\Longrightarrow$ height l = Suc (Suc (height r)) $\Longrightarrow$
  is_ord (l_bal$_0$ x l r)"
  ⟨*proof*⟩

lemma *is_ord_r_bal:*
 "is_ord (MKT$_0$ x l r) $\Longrightarrow$ height r = height l + 2 $\Longrightarrow$
  is_ord (r_bal$_0$ x l r)"
  ⟨*proof*⟩

If the order is linear, *insrt$_0$* maintains the order:

theorem *is_ord_insrt:*
 "is_ord t $\Longrightarrow$ is_ord (insrt$_0$ (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 ⇒ 'a tree0" where
  "erase ET = ET0"
  | "erase (MKT x l r h) = MKT0 x (erase l) (erase r)"

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

definition avl :: "'a tree ⇒ bool" where
  "avl t ⟷ is_bal (erase t) ∧ hinv t"
```

2.2.2 AVL interface and efficient implementation

```
primrec is_in :: "('a::preorder) ⇒ '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)"

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) = MKT_0 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_bal_0 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_bal_0 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 ⟹ hinv r ⟹ height(erase l) = height(erase r) + 2 ⟹
hinv (l_bal n l r)"
⟨*proof*⟩

lemma *hinv_r_bal:*
"hinv l ⟹ hinv r ⟹ height(erase r) = height(erase l) + 2 ⟹
hinv (r_bal n l r)"
⟨*proof*⟩

theorem *hinv_insrt: "hinv t ⟹ hinv (insrt x t)"*
⟨*proof*⟩

Function *insrt* implements $insrt_0$:

lemma *erase_insrt: "hinv t ⟹ erase (insrt x t) = $insrt_0$ x (erase t)"*
⟨*proof*⟩

Function *insrt* meets its spec:

corollary *"avl t $\Longrightarrow$ set_of (erase (insrt x t)) = insert x (set_of (erase t))"*
⟨*proof*⟩

Function *insrt* preserves the invariants:

corollary *"avl t $\Longrightarrow$ avl (insrt x t)"*
⟨*proof*⟩

corollary
"avl t $\Longrightarrow$ is_ord (erase t) $\Longrightarrow$ is_ord (erase (insrt (x::'a::linorder) t))"
⟨*proof*⟩

Function *is_in* implements *is_in*:

theorem *is_in: "is_in x t = is_in$_0$ x (erase t)"*
⟨*proof*⟩

Function *is_in* meets its spec:

corollary *"is_ord (erase t) $\Longrightarrow$ is_in x t $\longleftrightarrow$ x $\in$ set_of (erase t)"*
⟨*proof*⟩

end