AVL Trees

Tobias Nipkow and Cornelia Pusch

February 23, 2021

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"
by (induct t) simp_all

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"
by (cases l) (auto simp:mkt_def split:tree.split)

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"
by (cases r) (auto simp add:mkt_def split:tree.split)

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 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))"
by (cases l, cases r) simp_all

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))"
by (cases l, cases r) simp_all

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(cases l)
```

```
  case ET
  with assms show ?thesis by (simp add: mkt_def)
next
  case (MKT ln ll lr lh)
  with assms show ?thesis
  proof(cases "height l = height r + 2")
    case True
      from True MKT assms show ?thesis by (auto intro!: avl_mkt split: tree.split) arith+
  next
    case False
      with assms show ?thesis by (simp add: avl_mkt)
  qed
qed

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(cases r)
  case ET
  with assms show ?thesis by (simp add: mkt_def)
next
  case (MKT rn rl rr rh)
  with assms show ?thesis
  proof(cases "height r = height l + 2")
    case True
      from True MKT assms show ?thesis by (auto intro!: avl_mkt split: tree.split) arith+
  next
    case False
      with assms show ?thesis by (simp add: avl_mkt)
  qed
qed
```

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)"
using assms
proof (induction t)
  case (MKT n l r h)
  case 1
  with MKT show ?case
  proof(cases "x = n")
    case True
    with MKT 1 show ?thesis by simp
  next
    case False
    with MKT 1 show ?thesis
```

```
    proof(cases "x<n")
      case True
      with MKT 1 show ?thesis by (auto simp add:avl_mkt_bal_l simp del:mkt_bal_l.simps)
    next
      case False
      with MKT 1 ‹x≠n› show ?thesis by (auto simp add:avl_mkt_bal_r simp del:mkt_bal_r.simps)
    qed
  qed
  case 2
  from 2 MKT show ?case
  proof(cases "x = n")
    case True
    with MKT 1 show ?thesis by simp
  next
    case False
    with MKT 1 show ?thesis
     proof(cases "x<n")
      case True
      with MKT 2 show ?thesis
      proof(cases "height (AVL.insert x l) = height r + 2")
        case False with MKT 2 ‹x < n› show ?thesis by (auto simp del: mkt_bal_l.simps
simp: height_mkt_bal_l2)
      next
        case True
        then consider (a) "height (mkt_bal_l n (AVL.insert x l) r) = height r + 2"
          | (b) "height (mkt_bal_l n (AVL.insert x l) r) = height r + 3"
          using MKT 2 by (atomize_elim, intro height_mkt_bal_l) simp_all
        then show ?thesis
        proof cases
          case a
          with 2 ‹x < n› show ?thesis by (auto simp del: mkt_bal_l.simps)
        next
          case b
          with True 1 MKT(2) ‹x < n› show ?thesis by (simp del: mkt_bal_l.simps) arith
        qed
      qed
    next
      case False
      with MKT 2 show ?thesis
      proof(cases "height (AVL.insert x r) = height l + 2")
        case False with MKT 2 ‹¬x < n› show ?thesis by (auto simp del: mkt_bal_r.simps
simp: height_mkt_bal_r2)
      next
        case True
        then consider (a) "height (mkt_bal_r n l (AVL.insert x r)) = height l + 2"
          | (b) "height (mkt_bal_r n l (AVL.insert x r)) = height l + 3"
          using MKT 2 by (atomize_elim, intro height_mkt_bal_r) simp_all
        then show ?thesis
        proof cases
```

```
          case a
          with 2 ‹¬x < n› show ?thesis by (auto simp del: mkt_bal_r.simps)
        next
          case b
          with True 1 MKT(4) ‹¬x < n› show ?thesis by (simp del: mkt_bal_r.simps) arith
        qed
      qed
    qed
  qed
qed simp_all

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"
using assms
proof (induct x rule: delete_max_induct)
  case (MKT n l rn rl rr rh h)
  case 1
  with MKT have "avl l" "avl (snd (delete_max (MKT rn rl rr rh)))" by auto
  with 1 MKT have "avl (mkt_bal_l n l (snd (delete_max (MKT rn rl rr rh))))"
    by (intro avl_mkt_bal_l) fastforce+
  then show ?case
    by (auto simp: height_mkt_bal_l height_mkt_bal_l2
      linorder_class.max.absorb1 linorder_class.max.absorb2
      split:prod.split simp del:mkt_bal_l.simps)
next
  case (MKT n l rn rl rr rh h)
  case 2
  let ?r = "MKT rn rl rr rh"
  let ?r' = "snd (delete_max ?r)"
  from ‹avl x› MKT 2 have "avl l" and "avl ?r" by simp_all
  then show ?case using MKT 2 height_mkt_bal_l[of l ?r' n] height_mkt_bal_l2[of l ?r'
n]
    apply (auto split:prod.splits simp del:avl.simps mkt_bal_l.simps) by arith+
qed auto

lemma avl_delete_root:
  assumes "avl t" and "t ≠ ET"
  shows "avl(delete_root t)"
using assms
proof (cases t rule:delete_root_cases)
  case (MKT_MKT n ln ll lr lh rn rl rr rh h)
  let ?l = "MKT ln ll lr lh"
  let ?r = "MKT rn rl rr rh"
```

```
  let ?l' = "snd (delete_max ?l)"
  from ‹avl t› and MKT_MKT have "avl ?r" by simp
  from ‹avl t› and MKT_MKT have "avl ?l" by simp
  then have "avl(?l')" "height ?l = height(?l') ∨
         height ?l = height(?l') + 1" by (rule avl_delete_max,simp)+
  with ‹avl t› MKT_MKT have "height ?l' = height ?r ∨ height ?l' = height ?r + 1
          ∨ height ?r = height ?l' + 1 ∨ height ?r = height ?l' + 2" by fastforce
  with ‹avl ?l'› ‹avl ?r› have "avl(mkt_bal_r (fst(delete_max ?l)) ?l' ?r)"
    by (rule avl_mkt_bal_r)
  with MKT_MKT show ?thesis by (auto split:prod.splits simp del:mkt_bal_r.simps)
qed simp_all

lemma height_delete_root:
  assumes "avl t" and "t ≠ ET"
  shows "height t = height(delete_root t) ∨ height t = height(delete_root t) + 1"
using assms
proof (cases t rule: delete_root_cases)
  case (MKT_MKT n ln ll lr lh rn rl rr rh h)
  let ?l = "MKT ln ll lr lh"
  let ?r = "MKT rn rl rr rh"
  let ?l' = "snd (delete_max ?l)"
  let ?t' = "mkt_bal_r (fst(delete_max ?l)) ?l' ?r"
  from ‹avl t› and MKT_MKT have "avl ?r" by simp
  from ‹avl t› and MKT_MKT have "avl ?l" by simp
  then have "avl(?l')"  by (rule avl_delete_max,simp)
  have l'_height: "height ?l = height ?l' ∨ height ?l = height ?l' + 1" using ‹avl ?l›
by (intro avl_delete_max) auto
  have t_height: "height t = 1 + max (height ?l) (height ?r)" using ‹avl t› MKT_MKT by
simp
  have "height t = height ?t' ∨ height t = height ?t' + 1" using  ‹avl t› MKT_MKT
  proof(cases "height ?r = height ?l' + 2")
    case False
    show ?thesis using l'_height t_height False by (subst  height_mkt_bal_r2[OF ‹avl ?l'›
‹avl ?r› False])+ arith
  next
    case True
    show ?thesis
    proof(cases rule: disjE[OF height_mkt_bal_r[OF True ‹avl ?l'› ‹avl ?r›, of "fst (delete_max
?l)"]])
      case 1
      then show ?thesis using l'_height t_height True by arith
    next
      case 2
      then show ?thesis using l'_height t_height True by arith
    qed
  qed
  thus ?thesis using MKT_MKT by (auto split:prod.splits simp del:mkt_bal_r.simps)
qed simp_all
```

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"
using assms
proof (induct t)
  case (MKT n l r h)
  case 1
  with MKT show ?case
  proof(cases "x = n")
    case True
    with MKT 1 show ?thesis by (auto simp:avl_delete_root)
  next
    case False
    with MKT 1 show ?thesis
    proof(cases "x<n")
      case True
      with MKT 1 show ?thesis by (auto simp add:avl_mkt_bal_r simp del:mkt_bal_r.simps)
    next
      case False
      with MKT 1 ‹x≠n› show ?thesis by (auto simp add:avl_mkt_bal_l simp del:mkt_bal_l.simps)
    qed
  qed
  case 2
  with MKT show ?case
  proof(cases "x = n")
    case True
    with 1 have "height (MKT n l r h) = height(delete_root (MKT n l r h))
      ∨ height (MKT n l r h) = height(delete_root (MKT n l r h)) + 1"
      by (subst height_delete_root,simp_all)
    with True show ?thesis by simp
  next
    case False
    with MKT 1 show ?thesis
     proof(cases "x<n")
      case True
      show ?thesis
      proof(cases "height r = height (delete x l) + 2")
        case False with MKT 1 ‹x < n› show ?thesis by auto
      next
        case True
        then consider (a) "height (mkt_bal_r n (delete x l) r) = height (delete x l) +
2"
          | (b) "height (mkt_bal_r n (delete x l) r) = height (delete x l) + 3"
          using MKT 2 by (atomize_elim, intro height_mkt_bal_r) auto
        then show ?thesis
        proof cases
          case a
          with ‹x < n› MKT 2 show ?thesis by auto
```

```
        next
          case b
          with ‹x < n› MKT 2 show ?thesis by auto
        qed
      qed
    next
      case False
      show ?thesis
      proof(cases "height l = height (delete x r) + 2")
        case False with MKT 1 ‹¬x < n› ‹x ≠ n› show ?thesis by auto
      next
        case True
        then consider (a) "height (mkt_bal_l n l (delete x r)) = height (delete x r) +
2"
          | (b) "height (mkt_bal_l n l (delete x r)) = height (delete x r) + 3"
          using MKT 2 by (atomize_elim, intro height_mkt_bal_l) auto
        then show ?thesis
        proof cases
          case a
          with ‹¬x < n› ‹x ≠ n› MKT 2 show ?thesis by auto
        next
          case b
          with ‹¬x < n› ‹x ≠ n› MKT 2 show ?thesis by auto
        qed
      qed
    qed
  qed
qed simp_all

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)"
by (auto simp: mkt_def split:tree.splits)

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)"
by (auto simp: mkt_def split:tree.splits)
```

Correctness of `AVL.insert`:

```
theorem set_of_insert:
  "avl t ⟹ set_of(insert x t) = Set.insert x (set_of t)"
by (induct t)
   (auto simp: avl_insert set_of_mkt_bal_l set_of_mkt_bal_r simp del:mkt_bal_l.simps mkt_bal_r.simps
```

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"
by fastforce

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"
by auto

lemma ritem_in_rset: "r ≠ ET ⟹ rightmost_item r ∈ set_of r"
by(induct r rule:rightmost_item.induct) auto

lemma ritem_greatest_in_rset:
  "⟦ r ≠ ET; is_ord r ⟧ ⟹
  ∀x.  x ∈ set_of r ⟶ x ≠ rightmost_item r ⟶ x < rightmost_item r"
proof(induct r rule:rightmost_item.induct)
  case (2 n l rn rl rr rh h)
  show ?case (is "∀x. ?P x")
  proof
    fix x
    from 2 have "is_ord (MKT rn rl rr rh)" by auto
    moreover from 2 have "n < rightmost_item (MKT rn rl rr rh)"
      by (metis is_ord.simps(2) ritem_in_rset tree.simps(2))
    moreover from 2 have "x ∈ set_of l ⟶ x < rightmost_item (MKT rn rl rr rh)"
      by (metis calculation(2) is_ord.simps(2) xt1(10))
    ultimately show "?P x" using 2 by simp
  qed
qed auto

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"
by (metis avl_dist ritem_in_rset)

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 (induct t rule: delete_max_induct)
  case (MKT n l rn rl rr rh h)
  let ?r = "MKT rn rl rr rh"
  from MKT have "avl l" and "avl ?r" by simp_all
  let ?t' = "mkt_bal_l n l (snd (delete_max ?r))"
  from MKT have "avl (snd(delete_max ?r))" by (auto simp add: avl_delete_max)
```

```
  with MKT ritem_not_in_ltree[of n l ?r h]
  have "set_of ?t' = (set_of l) ∪ (set_of ?r) - {rightmost_item ?r} ∪ {n}"
    by (auto simp add:set_of_mkt_bal_l simp del: mkt_bal_l.simps)
  moreover have "n ∉ {rightmost_item ?r}"
    by (metis MKT(2) MKT(3) avl_dist2 ritem_in_rset singletonE tree.simps(3))
  ultimately show ?case
    by (auto simp add:insert_Diff_if split:prod.splits simp del: mkt_bal_l.simps)
qed auto

lemma fst_delete_max_eq_ritem:
  "t≠ET ⟹ fst(delete_max t) = rightmost_item t"
by (induct t rule:rightmost_item.induct) (auto split:prod.splits)

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}"
using assms
proof(cases t rule:delete_root_cases)
  case(MKT_MKT n ln ll lr lh rn rl rr rh h)
  let ?t' = "mkt_bal_r (fst (delete_max l)) (snd (delete_max l)) r"
  from assms MKT_MKT have "avl l" and "avl r" and "is_ord l" and "l≠ET" by auto
  moreover from MKT_MKT assms have "avl (snd(delete_max l))"
    by (auto simp add: avl_delete_max)
  ultimately have "set_of ?t' = (set_of l) ∪ (set_of r)"
    by (fastforce simp add: Set.insert_Diff ritem_in_rset fst_delete_max_eq_ritem
       set_of_delete_max set_of_mkt_bal_r  simp del: mkt_bal_r.simps)
  moreover from MKT_MKT assms(1) have "set_of (delete_root t) = set_of ?t'"
    by (simp split:prod.split del:mkt_bal_r.simps)
  moreover from MKT_MKT assms have "(set_of t) - {n} = set_of l ∪ set_of r"
    by (metis Diff_insert_absorb UnE avl_dist2 tree.set(2) tree.inject)
  ultimately show ?thesis using MKT_MKT assms(1)
    by (simp del: delete_root.simps)
qed auto
```

Correctness of `delete`:

```
theorem set_of_delete:
  "⟦ avl t; is_ord t ⟧ ⟹ set_of (delete x t) = (set_of t) - {x}"
proof (induct t)
  case (MKT n l r h)
  then show ?case
  proof(cases "x = n")
    case True
    with MKT set_of_delete_root[of "MKT n l r h"] show ?thesis by simp
  next
    case False
    with MKT show ?thesis
    proof(cases "x<n")
      case True
      with True MKT  show ?thesis
```

```
      by (force simp: avl_delete set_of_mkt_bal_r[of "(delete x l)" r n] simp del:mkt_bal_r.simps
    next
      case False
      with False MKT ‹x≠n› show ?thesis
        by (force simp: avl_delete set_of_mkt_bal_l[of l "(delete x r)" n] simp del:mkt_bal_l.simps
    qed
  qed
qed simp
```

1.4.5 Correctness of lookup

```
theorem is_in_correct: "is_ord t ⟹ is_in k t = (k : set_of t)"
by (induct t) auto
```

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)"
by (cases l) (auto simp: mkt_def split:tree.splits intro: order_less_trans)

lemma is_ord_mkt_bal_r: "is_ord(MKT n l r h) ⟹ is_ord (mkt_bal_r n l r)"
by (cases r) (auto simp: mkt_def split:tree.splits intro: order_less_trans)
```

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)"
by (induct t) (simp_all add:is_ord_mkt_bal_l is_ord_mkt_bal_r avl_insert set_of_insert
               linorder_not_less order_neq_le_trans del:mkt_bal_l.simps mkt_bal_r.simps)
```

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(induct t rule:delete_max_induct)
  case(MKT n l rn rl rr rh h)
  let ?r = "MKT rn rl rr rh"
  let ?r' = "snd(delete_max ?r)"
  from MKT have "∀h. is_ord(MKT n l ?r' h)" by (auto simp: set_of_delete_max)
  moreover from MKT have "avl(?r')" by (auto simp: avl_delete_max)
  moreover note MKT is_ord_mkt_bal_l[of n l ?r']
  ultimately show ?case by (auto split:prod.splits simp del:is_ord.simps mkt_bal_l.simps)
qed auto

lemma is_ord_delete_root:
  assumes "avl t" and "is_ord t" and "t ≠ ET"
  shows "is_ord (delete_root t)"
using assms
proof(cases t rule:delete_root_cases)
  case(MKT_MKT n ln ll lr lh rn rl rr rh h)
  let ?l = "MKT ln ll lr lh"
```

```
  let ?r = "MKT rn rl rr rh"
  let ?l' = "snd (delete_max ?l)"
  let ?n' = "fst (delete_max ?l)"
  from assms MKT_MKT have "∀h. is_ord(MKT ?n' ?l' ?r h)"
  proof -
    from assms MKT_MKT have "is_ord ?l'" by (auto simp add: is_ord_delete_max)
    moreover from assms MKT_MKT have "is_ord ?r" by auto
    moreover from assms MKT_MKT have "∀x. x ∈ set_of ?r ⟶ ?n' < x"
      by (metis fst_delete_max_eq_ritem is_ord.simps(2) order_less_trans ritem_in_rset

          tree.simps(3))
    moreover from assms MKT_MKT ritem_greatest_in_rset have "∀x. x ∈ set_of ?l' ⟶
x < ?n'"
      by (metis Diff_iff avl.simps(2) fst_delete_max_eq_ritem is_ord.simps(2)
          set_of_delete_max singleton_iff tree.simps(3))
    ultimately show ?thesis by auto
  qed
  moreover from assms MKT_MKT have "avl ?r" by simp
  moreover from assms MKT_MKT have "avl ?l'"  by (simp add: avl_delete_max)
  moreover note MKT_MKT is_ord_mkt_bal_r[of  ?n' ?l' ?r]
  ultimately show ?thesis by (auto simp del:mkt_bal_r.simps is_ord.simps split:prod.splits)
qed simp_all
```

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

```
theorem is_ord_delete:
  "⟦ avl t; is_ord t ⟧ ⟹ is_ord (delete x t)"
proof (induct t)
  case (MKT n l r h)
  then show ?case
  proof(cases "x = n")
    case True
    with MKT is_ord_delete_root[of "MKT n l r h"] show ?thesis by simp
  next
    case False
    with MKT show ?thesis
    proof(cases "x<n")
      case True
      with True MKT have "∀h. is_ord (MKT n (delete x l) r h)" by (auto simp:set_of_delete)
      with True MKT is_ord_mkt_bal_r[of n "(delete x l)" r]  show ?thesis
        by (auto simp add: avl_delete)
    next
      case False
      with False MKT have "∀h. is_ord (MKT n l (delete x r) h)" by (auto simp:set_of_delete)
      with False MKT is_ord_mkt_bal_l[of n l "(delete x r)"] ‹x≠n› show ?thesis by (simp
add: avl_delete)
    qed
  qed
qed simp
```

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"
  by (cases l) (auto split: tree0.split if_split_asm)

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"
  by (cases r) (auto split: tree0.split if_split_asm)

lemma height_insrt:
 "is_bal t
  ⟹ height (insrt0 x t) = height t ∨ height (insrt0 x t) = height t + 1"
proof (induct t)
  case ET0 show ?case by simp
next
```

```
  case (MKT0 n t1 t2) then show ?case proof (cases "x < n")
    case True show ?thesis
    proof (cases "height (insrt0 x t1) = height t2 + 2")
      case True with height_l_bal [of _ _ n]
      have "height (l_bal0 n (insrt0 x t1) t2) =
        height t2 + 2 ∨ height (l_bal0 n (insrt0 x t1) t2) = height t2 + 3" by simp
      with ‹x < n› MKT0 show ?thesis by auto
    next
      case False with ‹x < n› MKT0 show ?thesis by auto
    qed
  next
    case False show ?thesis
    proof (cases "height (insrt0 x t2) = height t1 + 2")
      case True with height_r_bal [of _ _ n]
      have "height (r_bal0 n t1 (insrt0 x t2)) = height t1 + 2 ∨
        height (r_bal0 n t1 (insrt0 x t2)) = height t1 + 3" by simp
      with ‹¬ x < n› MKT0 show ?thesis by auto
    next
      case False with ‹¬ x < n› MKT0 show ?thesis by auto
    qed
  qed
qed

lemma is_bal_l_bal:
  "is_bal l ⟹ is_bal r ⟹ height l = height r + 2 ⟹ is_bal (l_bal0 n l r)"
  by (cases l) (auto, auto split: tree0.split) — separating the two auto's is just for speed

lemma is_bal_r_bal:
  "is_bal l ⟹ is_bal r ⟹ height r = height l + 2 ⟹ is_bal (r_bal0 n l r)"
  by (cases r) (auto, auto split: tree0.split) — separating the two auto's is just for speed

theorem is_bal_insrt:
  "is_bal t ⟹ is_bal(insrt0 x t)"
proof (induct t)
  case ET0 show ?case by simp
next
  case (MKT0 n t1 t2) show ?case proof (cases "x < n")
    case True show ?thesis
    proof (cases "height (insrt0 x t1) = height t2 + 2")
      case True with ‹x < n› MKT0 show ?thesis
        by (simp add: is_bal_l_bal)
    next
      case False with ‹x < n› MKT0 show ?thesis
        using height_insrt [of t1 x] by auto
    qed
  next
    case False show ?thesis
    proof (cases "height (insrt0 x t2) = height t1 + 2")
      case True with ‹¬ x < n› MKT0 show ?thesis
```

```
        by (simp add: is_bal_r_bal)
    next
      case False with ‹¬ x < n› MKT0 show ?thesis
        using height_insrt [of t2 x] by auto
    qed
  qed
qed
```

2.1.4 Correctness of insertion

```
lemma set_of_l_bal: "height l = height r + 2 ⟹
  set_of (l_bal0 x l r) = insert x (set_of l ∪ set_of r)"
  by (cases l) (auto split: tree0.splits)

lemma set_of_r_bal: "height r = height l + 2 ⟹
  set_of (r_bal0 x l r) = insert x (set_of l ∪ set_of r)"
  by (cases r) (auto split: tree0.splits)

theorem set_of_insrt:
  "set_of (insrt0 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 ⟹ is_in0 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) ⟹ height l = Suc (Suc (height r)) ⟹
  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) ⟹ height r = height l + 2 ⟹
  is_ord (r_bal0 x l r)"
  by (cases r) (auto split:tree0.splits intro: order_less_trans)
```

If the order is linear, $insrt_0$ maintains the order:

```
theorem is_ord_insrt:
 "is_ord t ⟹ 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 ⇒ '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)"
  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 ⟹ hinv r ⟹ hinv (mkt x l r)"
  by (simp add: height_hinv mkt_def)

lemma hinv_l_bal:
 "hinv l ⟹ hinv r ⟹ height(erase l) = height(erase r) + 2 ⟹
  hinv (l_bal n l r)"
  by (cases l) (auto simp add: hinv_mkt split: tree.splits)

lemma hinv_r_bal:
 "hinv l ⟹ hinv r ⟹ height(erase r) = height(erase l) + 2 ⟹
  hinv (r_bal n l r)"
  by (cases r) (auto simp add: hinv_mkt split: tree.splits)

theorem hinv_insrt: "hinv t ⟹ hinv (insrt x t)"
  by (induct t) (simp_all add: Let_def height_hinv hinv_l_bal hinv_r_bal)
```

Function `insrt` implements $insrt_0$:

```
lemma erase_insrt: "hinv t ⟹ erase (insrt x t) = insrt₀ 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 ⟹ 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 ⟹ avl (insrt x t)"*
  by *(simp add: hinv_insrt avl_def erase_insrt is_bal_insrt)*

corollary
  "avl t ⟹ is_ord (erase t) ⟹ 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_0 *x (erase t)"*
  by *(induct t) simp_all*

Function *is_in* meets its spec:

corollary *"is_ord (erase t) ⟹ is_in x t ⟷ x ∈ set_of (erase t)"*
  by *(simp add:is_in is_in_correct)*

end