Verified AVL Trees in Lean 4

Alexander Gillon (agillon@andrew.cmu.edu)
Department of Philosophy, Carnegie Mellon University

Supervised by Jeremy Avigad Ph.D.
Departments of Philosophy/Mathematics, Carnegie Mellon University

Submitted in fulfillment of the requirements for the degree of
Bachelor in Science in Logic and Computation

April 30, 2024

0.1 Assumed Knowledge

This paper assumes understanding of the following:

- Functional programming
- Algebraic data types
- Binary search trees

Cursory knowledge about interactive theorem provers, or the Lean programming language, would also be helpful.

0.2 Typesetting Code

Please note that embedded code snippets in the first few pages are inserted as images, as a workaround of some typesetting issues. A full, text-based version of all code is available in Appendix A.

1 Introduction

1.1 Software Verification and Lean

Software verification is the process by which a computer program can be proven correct, relative to a formal specification. Provably correct software confers numerous benefits, the most important of which is that software is guaranteed to be defect-free when used [†]. As computer software becomes increasingly complex, there is an increasing demand for software to be formally verified, and the software verification movement has been gaining traction in recent years. Verification is of particular importance to safety-critical software applications (such as software in the automotive and medical industries), but there also also benefits to a wider variety of important, but not safety-critical software.

Lean is a functional programming language and interactive theorem prover. It allows software developers to write programs and mathematical proofs - these can be combined to formally verify programs by writing them in Lean, and writing proofs about them which prove that these programs satisfy a certain specification. Originally launched in 2013, the language has gained traction in recent years in the mathematical community, and continues to grow in usage. Lean 4 is the latest version of Lean, as of the time of writing.

1.2 AVL Trees

AVL trees are a data structure introduced in 1962 by inventors **A**delson-**V**elsky and **L**andis [‡], and are a specialized form of binary search trees. They implement an abstract mapping from data of a key type to data of a value type, and (in this paper) support insertion of key-value pairs and lookup of a value from a key. The strength of AVL trees is that they are self-balancing - this allows insertions and lookups to be performed in $O(\log n)$ time, where n is the number of elements in the mapping. AVL trees are used in a number of applications where large mappings are needed, and where insertion and lookup time need to be quick.

[†] On the condition that a number of other components in the software system are defect free, including the software verifier itself and the hardware on which the software runs.

[‡] Adelson-Velsky, Georgy; Landis, Evgenii (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences (Russian). 146: 263-266.

2 AVL Trees in Lean

This paper implements a formally verified AVL tree in Lean. That is, the implementation is proven to satisfy and maintain the AVL invariants across operations. It was desired that a number of auxiliary proofs would verify that the implementation correctly functions as an abstract mapping - due to time constraints, all proofs required to demonstrate this could not be completed. We state the desired proofs, and what has been completed among these proofs.

2.1 Tree Nodes

In order to define an AVL tree in Lean, we first define the core inductive datatype, which is a node of the tree:

```
inductive AVLNode (α : Type) (β : Type) (cmp : ComparisonFunction α) : Type
| Leaf : AVLNode α β cmp
| Node (height : ℤ) (key : α) (value : β) (left : AVLNode α β cmp) (right : AVLNode α β cmp) : AVLNode α β cmp
deriving Repr, DecidableEq, Inhabited
```

The only non-standard aspect of this definition is that the height of each node is kept stored at the node. This is a caching maneuver, and avoids having to re-calculate heights from the bottom up every time a tree changes. This definition relies on a user-supplied comparison function, which has type `ComparisonFunction` α, which is a trinary comparison function on keys of type α. The relevant definitions follow:

```
inductive Order : Type
| LESS    : Order
| EQUAL   : Order
| GREATER : Order
deriving Repr, DecidableEq, Inhabited

def XOR3 (a b c : Prop) : Prop := (a ∧ ¬ b ∧ ¬ c) ∨ (¬ a ∧ b ∧ ¬ c) ∨ (¬ a ∧ ¬ b ∧ c)

structure ComparisonFunction (α : Type) where
  cmp (k₁ : α) (k₂ : α) : Order

  cmpEq (k₁ : α) (k₂ : α)
    : cmp k₁ k₂ = Order.EQUAL ↔ k₁ = k₂
  cmpK (k : α)
    : cmp k k ≠ Order.LESS ∧ cmp k k = Order.EQUAL ∧ cmp k k ≠ Order.GREATER
  cmpXor (k₁ : α) (k₂ : α)
    : XOR3 (cmp k₁ k₂ = Order.LESS) (cmp k₁ k₂ = Order.EQUAL) (cmp k₁ k₂ = Order.GREATER)
  cmpTransitiveLess {k₁ k₂ k₃ : α} (h1 : cmp k₁ k₂ = Order.LESS) (h2 : cmp k₂ k₃ = Order.LESS)
    : cmp k₁ k₃ = Order.LESS
  cmpTransitiveGreater {k₁ k₂ k₃ : α} (h1 : cmp k₁ k₂ = Order.GREATER) (h2 : cmp k₂ k₃ = Order.GREATER)
    : cmp k₁ k₃ = Order.GREATER
  cmpLessToGreater {k₁ k₂ : α} (h : cmp k₁ k₂ = Order.LESS)
    : cmp k₂ k₁ = Order.GREATER
  cmpGreaterToLess {k₁ k₂ : α} (h : cmp k₁ k₂ = Order.GREATER)
    : cmp k₂ k₁ = Order.LESS
```

A comparison function is therefore the comparison function itself (`cmp`), along with a number of proofs that demonstrate that the supplied comparison function 'behaves nicely'. Each proof is clearly necessary of any function that must operate as a comparison function - this specific list of proofs is what happened to be needed for the underlying implementation.

2.2 AVL Trees

AVL trees satisfy two main invariants:

1. The binary search tree invariant: the key at a node is strictly greater than all keys in the left child of that node, and strictly less than all keys in the right child of that node.
2. The balance invariant: the height of the left and right children of any node differs by at most 1.

Due to the way in which we earlier defined an AVL node, we impose an additional constraint on AVL trees:

3. The height stored at each node is correct.

We therefore give the following definitions on AVL nodes, which formally capture these invariants. A valid AVL tree is therefore an `AVLNode n` for which `n.isAVLTree` is true.

```
def height {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : ℤ := match T with
  | Leaf => 0
  | (Node _ _ _ left right) => 1 + max left.height right.height

def balanceFactor {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : ℤ := match T with
  | Leaf => 0
  | (Node _ _ _ left right) => right.height - left.height

def allLess {α β : Type} {cmp : ComparisonFunction α} (k : α) (node : AVLNode α β cmp) : Prop := match node with
  | Leaf => true
  | (Node _ k' _ l r) => cmp.cmp k' k = Order.LESS ∧ allLess k l ∧ allLess k r

def allGreater {α β : Type} {cmp : ComparisonFunction α} (k : α) (node : AVLNode α β cmp) : Prop := match node with
  | Leaf => true
  | (Node _ k' _ l r) => cmp.cmp k' k = Order.GREATER ∧ allGreater k l ∧ allGreater k r

def isOrdered {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : Prop := match T with
  | Leaf => true
  | (Node _ k _ left right) => (allLess k left ∧ allGreater k right) ∧ left.isOrdered ∧ right.isOrdered

def heightCorrect {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : Prop := match T with
  | Leaf => true
  | (Node h _ _ left right) => (h = T.height) ∧ left.heightCorrect ∧ right.heightCorrect

def isBalanced {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : Bool := match T with
  | Leaf => true
  | (Node _ _ _ left right) => (T.balanceFactor = -1 ∨ T.balanceFactor = 0 ∨ T.balanceFactor = 1)
                               ∧ left.isBalanced ∧ right.isBalanced

def isAVLTree {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : Prop :=
  T.isOrdered ∧ T.heightCorrect ∧ T.isBalanced
```

We encapsulate this as a product type in Lean, as follows:

```
structure AVLTree (α : Type) (β : Type) (cmp : ComparisonFunction α) where
  root : AVLNode α β cmp
  wellFormed : root.isAVLTree
deriving Repr, DecidableEq
```

This allows an `AVLTree` to be passed around as one parameter. A user of an `AVLTree` has a formal guarantee that the `root` member is actually a well-formed AVL tree, via the `wellFormed` member.

2.3 Operations on AVL Trees

We provide the following interface for the creation and modification of AVL trees:

```
empty (α β : Type) (cmp : ComparisonFunction α) : AVLTree α β cmp
insert {α β : Type} {cmp : ComparisonFunction α} (T : AVLTree α β cmp) (key : α) (value : β) : AVLTree α β cmp
lookup {α β : Type} {cmp : ComparisonFunction α} (T : AVLTree α β cmp) (key : α) : Option β
```

`empty` allows for the creation of an empty AVL tree, `insert` allows insertion of a key-value pair into an existing AVL tree, and `lookup` allows for the value corresponding to a certain key to be retrieved (returning an option type to handle when the key is not present in the tree). A note: the intention is that `insert` is overwriting - that is, if a key already exists in the tree, and is inserted again, the previous value in the tree is overwritten.

However, by type alone, there is no guarantee that any implementation of the above functions is truly well-formed. For example, nothing in the signature of `empty` guarantees that the tree returned is actually empty, nor does the signature of `insert` guarantee that the key-value pair was actually inserted into the tree. In fact, from types alone, an implementation of `insert` which always returns an empty tree is compliant.

We therefore need some additional theorems which prove that our implementation is sound. We offer the following definitions:

```
AVLTreeInsertionProducesValue {α β : Type} {cmp : ComparisonFunction α}
  (T : AVLTree α β cmp) (k : α) (v : β)
  : (T.insert k v).lookup k = some v

AVLTreeInsertionDoesNotModifyOtherValues {α β : Type} {cmp : ComparisonFunction α}
  (T : AVLTree α β cmp) (k : α) (v : β) (k' : α) (result : Option β) (hk : k ≠ k')
  : T.lookup k' = result → (T.insert k v).lookup k' = result

EmptyAVLTreeLookup {α β : Type} {cmp : ComparisonFunction α} (k : α)
  : (empty α β cmp).lookup k = none
```

Any implementation which is able to prove these theorems therefore guarantees behavior that a user would expect. Namely, that the empty tree is empty and that insertion does indeed insert a key-value pair into the tree without logically modifying the tree in any other way.

The implementation provided at the end of this document implements the provided interface, and proves both `AVLTreeInsertionProducesValue` and `EmptyAVLTreeLookup`. Unfortunately, due to time constraints, `AVLTreeInsertionDoesNotModifyOtherValues` was not able to be proven. However, we claim that the implementation does indeed satisfy this theorem, and that with sufficient time it could be proven in a similar way to `AVLTreeInsertionProducesValue`.

3 Conclusion and Acknowledgements

The only thing that remains is the implementation of the AVL tree itself. This is available in appendix A, and the original source is available on GitHub on request.

I would like to take a moment to thank Professor Jeremy Avigad, who was my supervisor for this project. His invaluable help allowed me to learn a lot about Lean and interactive theorem proving, and his expertise was able to help me get un-stuck numerous times during the proving process.

A AVL Tree Implementation

Implementation begins overleaf. Source is available on request.

```
import Mathlib.Data.Nat.Basic
import Mathlib.Tactic

/-
A formally verified AVL tree.

Some common abbreviations:

  h: height
  k: key
  v: value
  l: left
  r: right

And some common suffixes:

  ro: root
  ne: newly created (subscript w isn't a unicode character ☹)
  l: left
  r: right
  i: inserted

Left/right suffixes can be chained together.

-/

inductive Order : Type
| LESS    : Order
| EQUAL   : Order
| GREATER : Order
deriving Repr, DecidableEq, Inhabited

def XOR3 (a b c : Prop) : Prop := (a ∧ ¬ b ∧ ¬ c) ∨ (¬ a ∧ b ∧ ¬ c) ∨ (¬ a ∧ ¬ b ∧ c)

structure ComparisonFunction (α : Type) where
  cmp (k₁ : α) (k₂ : α) : Order
  cmpEq (k₁ : α) (k₂ : α) : cmp k₁ k₂ = Order.EQUAL ↔ k₁ = k₂
  cmpK (k : α) : cmp k k ≠ Order.LESS ∧ cmp k k = Order.EQUAL ∧ cmp k k ≠ Order.GREATER
  cmpXor (k₁ : α) (k₂ : α) : XOR3 (cmp k₁ k₂ = Order.LESS) (cmp k₁ k₂ = Order.EQUAL) (cmp k₁ k₂ = Order.GREATER)
  cmpTransitiveLess {k₁ k₂ k₃ : α} (h1 : cmp k₁ k₂ = Order.LESS) (h2 : cmp k₂ k₃ = Order.LESS)
    : cmp k₁ k₃ = Order.LESS
  cmpTransitiveGreater {k₁ k₂ k₃ : α} (h1 : cmp k₁ k₂ = Order.GREATER) (h2 : cmp k₂ k₃ = Order.GREATER)
    : cmp k₁ k₃ = Order.GREATER
  cmpLessToGreater {k₁ k₂ : α} (h : cmp k₁ k₂ = Order.LESS) : cmp k₂ k₁ = Order.GREATER
  cmpGreaterToLess {k₁ k₂ : α} (h : cmp k₁ k₂ = Order.GREATER) : cmp k₂ k₁ = Order.LESS

/- A hopefully intuitive definition of an AVL node. -/
inductive AVLNode (α : Type) (β : Type) (cmp : ComparisonFunction α) : Type
| Leaf : AVLNode α β cmp
| Node (height : ℤ) (key : α) (value : β) (left : AVLNode α β cmp) (right : AVLNode α β cmp) : AVLNode α β cmp
deriving Repr, DecidableEq, Inhabited

namespace AVLNode

  def max (a : ℤ) (b : ℤ) : ℤ := if a > b then a else b

  def isNode {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : Bool := match T with
    | Leaf => false
    | (Node _ _ _ _ _) => true

  def height {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : ℤ := match T with
    | Leaf => 0
    | (Node _ _ _ left right) => 1 + max left.height right.height

  def balanceFactor {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : ℤ := match T with
    | Leaf => 0
    | (Node _ _ _ left right) => right.height - left.height

  def allLess {α β : Type} {cmp : ComparisonFunction α} (k : α) (node : AVLNode α β cmp) : Prop := match node with
```

```
    | Leaf => true
    | (Node _ k' _ l r) => cmp.cmp k' k = Order.LESS ∧ allLess k l ∧ allLess k r

  def allGreater {α β : Type} {cmp : ComparisonFunction α} (k : α) (node : AVLNode α β cmp) : Prop := match node
with
    | Leaf => true
    | (Node _ k' _ l r) => cmp.cmp k' k = Order.GREATER ∧ allGreater k l ∧ allGreater k r

  def isOrdered {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : Prop := match T with
    | Leaf => true
    | (Node _ k _ left right) => (allLess k left ∧ allGreater k right) ∧ left.isOrdered ∧ right.isOrdered

  def heightCorrect {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : Prop := match T with
    | Leaf => true
    | (Node h _ _ left right) => (h = T.height) ∧ left.heightCorrect ∧ right.heightCorrect

  def isBalanced {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : Bool := match T with
    | Leaf => true
    | (Node _ _ _ left right) => (T.balanceFactor = -1 ∨ T.balanceFactor = 0 ∨ T.balanceFactor = 1)
        ∧ left.isBalanced ∧ right.isBalanced

  def isAVLTree {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : Prop :=
    T.isOrdered ∧ T.heightCorrect ∧ T.isBalanced

  def maps {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) (k : α) (v : β) : Prop := match T with
    | Leaf => false
    | (Node _ k' v' left right) => (cmp.cmp k k' = Order.EQUAL ∧ v = v')
        ∨ (cmp.cmp k k' = Order.LESS ∧ left.maps k v) ∨ (cmp.cmp k k' = Order.GREATER ∧ right.maps k v)

end AVLNode

/- An AVL tree is an underlying tree, accompanied by a proof that the tree is well-formed
   (i.e. observes the AVL invariants). -/
structure AVLTree (α : Type) (β : Type) (cmp : ComparisonFunction α) where
  root : AVLNode α β cmp
  wellFormed : root.isAVLTree
deriving Repr, DecidableEq

namespace AVLTree

  open AVLNode

  /- Intermediate structure for making an insertion into an AVL tree. This is needed to pass on certain
     proofs to callers. -/
  structure AVLTreeInsertion {α β : Type} {cmp : ComparisonFunction α}
    (kᵢ : α) -- key which was inserted into the tree
    (vᵢ : β) -- value which was inserted into the tree
    (origT : AVLTree α β cmp) -- tree that the key was inserted into
    (n : ℤ) -- height of origT
    where
    -- the new tree, which was obtained by inserting something with key kᵢ into origT
    T : AVLTree α β cmp
    -- a proof that the new tree obtained increased in height by at most 1
    heightBound : T.root.height = n ∨ T.root.height = n + 1
    -- a proof that if the original tree was all less than k, and if the inserted key is less than k,
    -- then the new tree is all less than k
    lessThanProof (k : α) (hKey : cmp.cmp kᵢ k = Order.LESS) (hTree : allLess k origT.root) : allLess k T.root
    -- a proof that if the original tree was all greater than k, and if the inserted key is greater than k,
    -- then the new tree is all greater than k
    greaterThanProof (k : α) (hKey : cmp.cmp kᵢ k = Order.GREATER) (hTree : allGreater k origT.root)
      : allGreater k T.root
    -- a proof that T is a node
    isNodeProof : T.root.isNode
    -- a proof that T maps kᵢ to vᵢ
    mapsTo : T.root.maps kᵢ vᵢ
  deriving Repr, DecidableEq

  lemma applyIff {a b : Prop} (hIff : a ↔ b) (ha : a) : b := by tauto
```

```
  def cachedHeight {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : ℤ := match T with
    | Leaf => 0
    | (Node h _ _ _ _) => h

  def extractIsOrdered {α β : Type} {cmp : ComparisonFunction α} {T : AVLNode α β cmp} (h : T.isAVLTree)
  : T.isOrdered := by
    unfold isAVLTree at h
    tauto

  def extractHeightCorrect {α β : Type} {cmp : ComparisonFunction α} {T : AVLNode α β cmp} (h : T.isAVLTree)
  : T.heightCorrect := by
    unfold isAVLTree at h
    tauto

  def extractIsBalanced {α β : Type} {cmp : ComparisonFunction α} {T : AVLNode α β cmp} (h : T.isAVLTree)
  : T.isBalanced := by
    unfold isAVLTree at h
    tauto

  lemma leafIsAVLTree {α β : Type} {cmp : ComparisonFunction α} : (@Leaf α β cmp).isAVLTree := by tauto

  lemma singletonIsAVLTree {α β : Type} {cmp : ComparisonFunction α} (key : α) (value : β)
  : ((@Node α β cmp) 1 key value Leaf Leaf).isAVLTree := by
    rw [isAVLTree, isOrdered, isBalanced, decide_eq_true_eq]
    tauto

  lemma childrenAreAVLTrees (hWellFormed : (Node h k v l r).isAVLTree) : l.isAVLTree ∧ r.isAVLTree := by
    rw [isAVLTree, isOrdered, heightCorrect, isBalanced, decide_eq_true_eq] at hWellFormed
    tauto

  lemma leftIsAVLTree (hWellFormed : (Node h k v l r).isAVLTree) : l.isAVLTree := by
    have _ := childrenAreAVLTrees hWellFormed
    tauto

  lemma rightIsAVLTree (hWellFormed : (Node h k v l r).isAVLTree) : r.isAVLTree := by
    have _ := childrenAreAVLTrees hWellFormed
    tauto

  lemma cachedHeightIsCorrect {α β : Type} {cmp : ComparisonFunction α}
  (node : AVLNode α β cmp) (hWellFormed : node.isAVLTree) : cachedHeight node = node.height := by match node with
    | Leaf => tauto
    | (Node h k v l r) =>
        have hOrigHeight := extractHeightCorrect hWellFormed
        simp only [heightCorrect] at hOrigHeight
        rw [cachedHeight, ←hOrigHeight.1]

  lemma heightIncreasesLeft {α β : Type} {cmp : ComparisonFunction α} {node : AVLNode α β cmp} {h k v left right}
      (hEq : node = (Node h k v left right)) : height node ≥ height left := by
    rw [hEq, height, AVLNode.max]
    split <;> linarith

  lemma heightIncreasesRight {α β : Type} {cmp : ComparisonFunction α} {node : AVLNode α β cmp} {h k v left right}
      (hEq : node = (Node h k v left right)) : height node ≥ height right := by
    rw [hEq, height, AVLNode.max]
    split <;> linarith

  lemma childHeightLeft {α β : Type} {cmp : ComparisonFunction α} {n : ℤ} {node : AVLNode α β cmp} {h k v left
right}
      (hHeight : height node = n) (hEq : node = (Node h k v left right)) (hBal : node.isBalanced)
  : height left = n - 2 ∨ height left = n - 1 := by
    simp only [hEq, height, AVLNode.max] at hHeight

    if hOrdering : height left > height right then
      have hIf : (if height left > height right then height left else height right) = height left := by
        split <;> tauto
      rw [hIf] at hHeight
      have : height left = n - 1 := by linarith
      tauto
    else
```

```
    have hIf : (if height left > height right then height left else height right) = height right := by
      split <;> tauto
    rw [hIf] at hHeight
    simp only [hEq, isBalanced, decide_eq_true_eq] at hBal
    cases hBal.1

    case inl hBal' =>
      simp only [balanceFactor] at hBal'
      linarith

    case inr hBal' =>
      cases hBal'

      case inl hBal' =>
        simp only [balanceFactor] at hBal'
        have : height left = n - 1 := by linarith
        tauto

      case inr hBal' =>
        simp only [balanceFactor] at hBal'
        have : height left = n - 2 := by linarith
        tauto

lemma childHeightRight {α β : Type} {cmp : ComparisonFunction α} {n : ℤ} {node : AVLNode α β cmp}
    {h k v left right} (hHeight : height node = n) (hEq : node = (Node h k v left right)) (hBal : node.isBalanced)

  : height right = n - 2 ∨ height right = n - 1 := by
  simp only [hEq, height, AVLNode.max] at hHeight

  if hOrdering : height left > height right then
    have hIf : (if height left > height right then height left else height right) = height left := by
      split <;> tauto
    rw [hIf] at hHeight

    simp only [hEq, isBalanced, decide_eq_true_eq] at hBal
    cases hBal.1

    case inl hBal' =>
      simp only [balanceFactor] at hBal'
      have : height right = n - 2 := by linarith
      tauto

    case inr hBal' =>
      cases hBal'

      case inl hBal' =>
        simp only [balanceFactor] at hBal'
        have : height right = n - 1 := by linarith
        tauto

      case inr hBal' =>
        simp only [balanceFactor] at hBal'
        linarith
  else
    have hIf : (if height left > height right then height left else height right) = height right := by
      split <;> tauto
    rw [hIf] at hHeight
    have : height right = n - 1 := by linarith
    tauto


lemma leafOrNode {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : T.isNode ∨ T = Leaf := by
  cases T <;> tauto

lemma notNodeToLeaf {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) (hNotNode : ¬T.isNode)
: T = Leaf := by cases leafOrNode T <;> tauto

lemma maxNonNegative {a b : ℤ} (ha : a ≥ 0) (hb : b ≥ 0) : AVLNode.max a b ≥ 0 := by
  unfold AVLNode.max
```

```
  split <;> linarith

lemma heightIsNonNegative {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) : T.height ≥ 0 := by
    induction T
    case Leaf => tauto
    case Node l r ihl ihr =>
      rw [height, AVLNode.max]
      split <;> linarith

lemma lessAllLess {α β : Type} {cmp : ComparisonFunction α} {k₁ k₂ : α} {T : AVLNode α β cmp}
  (hLess : cmp.cmp k₁ k₂ = Order.LESS) (hAllLess : allLess k₁ T) : allLess k₂ T := by

  induction T

  case Leaf => tauto

  case Node _ _ _ _ _ left_ih right_ih =>
    unfold allLess
    unfold allLess at hAllLess

    apply And.intro

    . exact cmp.cmpTransitiveLess hAllLess.1 hLess

    apply And.intro

    . exact left_ih hAllLess.2.1
    . exact right_ih hAllLess.2.2

lemma greaterAllGreater {α β : Type} {cmp : ComparisonFunction α} {k₁ k₂ : α} {T : AVLNode α β cmp}
  (hGreater : cmp.cmp k₁ k₂ = Order.GREATER) (hAllGreater : allGreater k₁ T) : allGreater k₂ T := by

  induction T

  case Leaf => tauto

  case Node _ _ _ _ _ left_ih right_ih =>
    unfold allGreater
    unfold allGreater at hAllGreater

    apply And.intro

    . exact cmp.cmpTransitiveGreater hAllGreater.1 hGreater

    apply And.intro

    . exact left_ih hAllGreater.2.1
    . exact right_ih hAllGreater.2.2

lemma insertLeftChildLessThanProof {α β : Type} {cmp : ComparisonFunction α} {lₙₑ rootₙₑ}
  {newWellFormed : isAVLTree rootₙₑ} {T : AVLTree α β cmp}
  (hᵣₒ : ℤ) (kᵣₒ : α) (vᵣₒ : β) (lᵣₒ rᵣₒ : AVLNode α β cmp)
  (kᵢ : α) (vᵢ : β)
  (hNewLeft : lₙₑ = (@Node α β cmp) 1 kᵢ vᵢ Leaf Leaf)
  (hNewRoot : rootₙₑ = Node (1 + AVLNode.max lₙₑ.height rᵣₒ.height) kᵣₒ vᵣₒ lₙₑ rᵣₒ)
  (hT : T = { root := rootₙₑ, wellFormed := newWellFormed })
  (k : α) (hKey : cmp.cmp kᵢ k = Order.LESS) (hTree : allLess k (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ))
  : allLess k T.root := by
    simp only [hT, allLess, hNewRoot, hNewLeft]
    unfold allLess at hTree
    tauto

lemma insertLeftChildGreaterThanProof {α β : Type} {cmp : ComparisonFunction α} {lₙₑ rootₙₑ}
  {newWellFormed : isAVLTree rootₙₑ} {T : AVLTree α β cmp}
  (hᵣₒ : ℤ) (kᵣₒ : α) (vᵣₒ : β) (lᵣₒ rᵣₒ : AVLNode α β cmp)
  (kᵢ : α) (vᵢ : β)
  (hNewLeft : lₙₑ = (@Node α β cmp) 1 kᵢ vᵢ Leaf Leaf)
  (hNewRoot : rootₙₑ = Node (1 + AVLNode.max lₙₑ.height rᵣₒ.height) kᵣₒ vᵣₒ lₙₑ rᵣₒ)
  (hT : T = { root := rootₙₑ, wellFormed := newWellFormed })
```

```
  (k : α) (hKey : cmp.cmp kᵢ k = Order.GREATER) (hTree : allGreater k (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ))
  : allGreater k T.root := by
    simp only [hT, allGreater, hNewRoot, hNewLeft]
    unfold allGreater at hTree
    tauto

def insertLeftChild {α β : Type} {cmp : ComparisonFunction α} {snd}
  (hᵣₒ : ℤ) (kᵣₒ : α) (vᵣₒ : β) (lᵣₒ rᵣₒ : AVLNode α β cmp)
  (hWellFormed : (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ).isAVLTree)
  (kᵢ : α) (vᵢ : β) (hMatch : cmp.cmp kᵢ kᵣₒ = Order.LESS ∧ lᵣₒ = Leaf ∧ rᵣₒ = snd)

  : AVLTreeInsertion kᵢ vᵢ { root := Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ, wellFormed := hWellFormed } hᵣₒ :=

  let lₙₑ := (@Node α β cmp) 1 kᵢ vᵢ Leaf Leaf
  let rootₙₑ := Node (1 + AVLNode.max (cachedHeight lₙₑ) (cachedHeight rᵣₒ)) kᵣₒ vᵣₒ lₙₑ rᵣₒ

  have hCachedHeightLₙₑ := cachedHeightIsCorrect lₙₑ (singletonIsAVLTree kᵢ vᵢ)
  have hCachedHeightRᵣₒ := cachedHeightIsCorrect rᵣₒ (rightIsAVLTree hWellFormed)

  have hHeightLᵣₒ : height lᵣₒ = 0 := by
    rw [hMatch.2.1]
    tauto
  have hHeightLₙₑ : height lₙₑ = 1 := by tauto

  have balanceFactorChange {n : ℤ} (hBal : balanceFactor (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ) = n)
  : balanceFactor rootₙₑ = n-1 := by
    simp only [balanceFactor, hHeightLᵣₒ, sub_zero] at hBal
    simp only [balanceFactor, hHeightLₙₑ, hBal]

  have heightRᵣₒ {n : ℤ} (hBal : balanceFactor (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ) = n) : rᵣₒ.height = n := by
    simp only [balanceFactor, hHeightLᵣₒ, sub_zero] at hBal
    exact hBal

  -- used for some 'contradiction' tactics
  have balanceFactorRightHeavy (hBal : balanceFactor (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ) = -1) : ⊥ := by
    have hHeightRᵣₒ : height rᵣₒ = -1 := heightRᵣₒ hBal
    have : height rᵣₒ ≥ 0 := heightIsNonNegative rᵣₒ
    linarith

  have heightRoot {n : ℤ} (hn : ¬ n < 0) (hBal : balanceFactor (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ) = n) : hᵣₒ = n+1 := by
    have hHeightRᵣₒ : height rᵣₒ = n := heightRᵣₒ hBal
    have hOrigHeight := extractHeightCorrect hWellFormed
    unfold heightCorrect at hOrigHeight
    have hOrigHeight' := hOrigHeight.1

    rw [height, hHeightLᵣₒ, hHeightRᵣₒ, AVLNode.max] at hOrigHeight'
    have hIf : (if n < 0 then 0 else n) = n := by split <;> first | contradiction | rfl
    rw [hOrigHeight', hIf]
    linarith

  have hNewRootIsAVLTree : rootₙₑ.isAVLTree := by
    unfold isAVLTree
    apply And.intro

    . unfold isOrdered
      apply And.intro

      . apply And.intro

        . tauto
        . have hOrigOrdered := extractIsOrdered hWellFormed
          unfold isOrdered at hOrigOrdered
          exact hOrigOrdered.1.2

      apply And.intro

      . unfold isOrdered
        tauto
```

```
    . have hOrigOrdered := extractIsOrdered hWellFormed
      unfold isOrdered at hOrigOrdered
      exact hOrigOrdered.2.2

  apply And.intro

  . unfold heightCorrect
    apply And.intro

    . tauto

    apply And.intro

    . tauto
    . have hOrigHeight := extractHeightCorrect hWellFormed
      unfold heightCorrect at hOrigHeight
      exact hOrigHeight.2.2

  . rw [isBalanced, decide_eq_true_eq]
    apply And.intro

    . have hOrigBal := extractIsBalanced hWellFormed
      rw [isBalanced, decide_eq_true_eq] at hOrigBal
      cases hOrigBal.1

      case inl hBal =>
        contradiction

      case inr hBal =>
        cases hBal

        case inl hBal =>
          have _ := balanceFactorChange hBal
          tauto

        case inr hBal =>
          have _ := balanceFactorChange hBal
          tauto

    apply And.intro

    . tauto

    . have hOrigBal := extractIsBalanced hWellFormed
      rw [isBalanced, decide_eq_true_eq] at hOrigBal
      exact hOrigBal.2.2

let T := { root := rootₙₑ, wellFormed := hNewRootIsAVLTree }

have hHeightBound := by
  have hOrigBal := extractIsBalanced hWellFormed
  rw [isBalanced, decide_eq_true_eq] at hOrigBal
  cases hOrigBal.1

  case inl hBal => contradiction

  case inr hBal =>
    cases hBal

    case inl hBal =>
      have hHeightRoot := heightRoot (by simp only [lt_self_iff_false, not_false_eq_true]) hBal
      have hHeightRootₙₑ : T.root.height = 2 := by
        have hHeightRᵣₒ := heightRᵣₒ hBal
        have hEq : T.root = rootₙₑ := by simp only
        rw [hEq, height, hHeightLₙₑ, hHeightRᵣₒ, AVLNode.max]
        tauto

      rw [hHeightRootₙₑ, hHeightRoot]
      tauto
```

```
      case inr hBal =>
        have hHeightRoot := heightRoot (by simp only [Int.reduceLT, not_false_eq_true]) hBal
        have hHeightRoot_ne : T.root.height = 2 := by
          have hHeightR_ro := heightR_ro hBal
          have hEq : T.root = root_ne := by simp only
          rw [hEq, height, hHeightL_ne, hHeightR_ro, AVLNode.max]
          tauto

        rw [hHeightRoot_ne, hHeightRoot]
        tauto

  have hNewLeft : l_ne = (@Node α β cmp) 1 k_i v_i Leaf Leaf := by tauto
  have hNewRoot : root_ne = Node (1 + AVLNode.max l_ne.height r_ro.height) k_ro v_ro l_ne r_ro := by
    unfold_let root_ne
    rw [hCachedHeightL_ne, hCachedHeightR_ro]
  have hT : T = { root := root_ne, wellFormed := hNewRootIsAVLTree } := by tauto

  have lessThanProof := insertLeftChildLessThanProof h_ro k_ro v_ro l_ro r_ro k_i v_i hNewLeft hNewRoot hT
  have greaterThanProof := insertLeftChildGreaterThanProof h_ro k_ro v_ro l_ro r_ro k_i v_i hNewLeft hNewRoot hT

  have hMapsTo : maps T.root k_i v_i := by
    simp only [maps]
    have : cmp.cmp k_i k_ro = Order.LESS ∧ (cmp.cmp k_i k_i = Order.EQUAL ∧ True
      ∨ cmp.cmp k_i k_i = Order.LESS ∧ False ∨ cmp.cmp k_i k_i = Order.GREATER ∧ False) := by
      have : cmp.cmp k_i k_i = Order.EQUAL := (cmp.cmpK k_i).2.1
      tauto
    tauto

{ T := T , heightBound := hHeightBound, lessThanProof := lessThanProof,
  greaterThanProof := greaterThanProof, isNodeProof := by tauto, mapsTo := hMapsTo }

lemma insertRightChildLessThanProof {α β : Type} {cmp : ComparisonFunction α} {r_ne root_ne}
  {newWellFormed : isAVLTree root_ne} {T : AVLTree α β cmp}
  (h_ro : ℤ) (k_ro : α) (v_ro : β) (l_ro r_ro : AVLNode α β cmp)
  (k_i : α) (v_i : β)
  (hNewRight : r_ne = (@Node α β cmp) 1 k_i v_i Leaf Leaf)
  (hNewRoot : root_ne = Node (1 + AVLNode.max l_ro.height r_ne.height) k_ro v_ro l_ro r_ne)
  (hT : T = { root := root_ne, wellFormed := newWellFormed })
  (k : α) (hKey : cmp.cmp k_i k = Order.LESS) (hTree : allLess k (Node h_ro k_ro v_ro l_ro r_ro))
  : allLess k T.root := by
    simp only [hT, allLess, hNewRoot, hNewRight]
    unfold allLess at hTree
    tauto

lemma insertRightChildGreaterThanProof {α β : Type} {cmp : ComparisonFunction α} {r_ne root_ne}
  {newWellFormed : isAVLTree root_ne} {T : AVLTree α β cmp}
  (h_ro : ℤ) (k_ro : α) (v_ro : β) (l_ro r_ro : AVLNode α β cmp)
  (k_i : α) (v_i : β)
  (hNewRight : r_ne = (@Node α β cmp) 1 k_i v_i Leaf Leaf)
  (hNewRoot : root_ne = Node (1 + AVLNode.max l_ro.height r_ne.height) k_ro v_ro l_ro r_ne)
  (hT : T = { root := root_ne, wellFormed := newWellFormed })
  (k : α) (hKey : cmp.cmp k_i k = Order.GREATER) (hTree : allGreater k (Node h_ro k_ro v_ro l_ro r_ro))
  : allGreater k T.root := by
    simp only [hT, allGreater, hNewRoot, hNewRight]
    unfold allGreater at hTree
    tauto

def insertRightChild {α β : Type} {cmp : ComparisonFunction α} {fst}
  (h_ro : ℤ) (k_ro : α) (v_ro : β) (l_ro r_ro : AVLNode α β cmp)
  (hWellFormed : (Node h_ro k_ro v_ro l_ro r_ro).isAVLTree)
  (k_i : α) (v_i : β) (hMatch : cmp.cmp k_i k_ro = Order.GREATER ∧ l_ro = fst ∧ r_ro = Leaf)

  : AVLTreeInsertion k_i v_i { root := Node h_ro k_ro v_ro l_ro r_ro, wellFormed := hWellFormed } h_ro :=

  let r_ne := (@Node α β cmp) 1 k_i v_i Leaf Leaf
  let root_ne := Node (1 + AVLNode.max (cachedHeight l_ro) (cachedHeight r_ne)) k_ro v_ro l_ro r_ne

  have hCachedHeightL_ro := cachedHeightIsCorrect l_ro (leftIsAVLTree hWellFormed)
```

```
have hCachedHeightR$_{n_e}$ := cachedHeightIsCorrect r$_{n_e}$ (singletonIsAVLTree k$_i$ v$_i$)

have hHeightR$_{r_o}$ : height r$_{r_o}$ = 0 := by
  rw [hMatch.2.2]
  tauto
have hHeightR$_{n_e}$ : height r$_{n_e}$ = 1 := by tauto

have balanceFactorChange {n : ℤ} (hBal : balanceFactor (Node h$_{r_o}$ k$_{r_o}$ v$_{r_o}$ l$_{r_o}$ r$_{r_o}$) = n)
: balanceFactor root$_{n_e}$ = n+1 := by
  simp only [balanceFactor, hHeightR$_{r_o}$] at hBal
  simp only [balanceFactor, hHeightR$_{n_e}$]
  linarith

have heightL$_{r_o}$ {n : ℤ} (hBal : balanceFactor (Node h$_{r_o}$ k$_{r_o}$ v$_{r_o}$ l$_{r_o}$ r$_{r_o}$) = n) : l$_{r_o}$.height = -n := by
  simp only [balanceFactor, hHeightR$_{r_o}$, sub_zero] at hBal
  linarith

have balanceFactorLeftHeavy (hBal : balanceFactor (Node h$_{r_o}$ k$_{r_o}$ v$_{r_o}$ l$_{r_o}$ r$_{r_o}$) = 1) : ⊥ := by
  have hHeightL$_{r_o}$ : height l$_{r_o}$ = -1 := heightL$_{r_o}$ hBal
  have : height l$_{r_o}$ ≥ 0 := heightIsNonNegative l$_{r_o}$
  linarith

have heightRoot {n : ℤ} (hn : ¬ -n < 0) (hBal : balanceFactor (Node h$_{r_o}$ k$_{r_o}$ v$_{r_o}$ l$_{r_o}$ r$_{r_o}$) = n) : h$_{r_o}$ = 1-n := by
  have hHeightL$_{r_o}$ : height l$_{r_o}$ = -n := heightL$_{r_o}$ hBal
  have hOrigHeight := extractHeightCorrect hWellFormed
  unfold heightCorrect at hOrigHeight
  have hOrigHeight' := hOrigHeight.1

  rw [height, hHeightL$_{r_o}$, hHeightR$_{r_o}$, AVLNode.max] at hOrigHeight'
  have hIf : (if -n > 0 then -n else 0) = -n := by split <;> linarith
  rw [hOrigHeight', hIf]
  linarith

have hNewRootIsAVLTree : root$_{n_e}$.isAVLTree := by
  unfold isAVLTree
  apply And.intro

  . unfold isOrdered
    apply And.intro

    . apply And.intro

      . have hOrigOrdered := extractIsOrdered hWellFormed
        unfold isOrdered at hOrigOrdered
        exact hOrigOrdered.1.1
      . tauto

    apply And.intro

    . have hOrigOrdered := extractIsOrdered hWellFormed
      unfold isOrdered at hOrigOrdered
      exact hOrigOrdered.2.1
    . unfold isOrdered
      tauto

  apply And.intro

  . unfold heightCorrect
    apply And.intro

    . tauto

    apply And.intro

    . have hOrigHeight := extractHeightCorrect hWellFormed
      unfold heightCorrect at hOrigHeight
      exact hOrigHeight.2.1
    . tauto
```

```
  . rw [isBalanced, decide_eq_true_eq]
    apply And.intro

    . have hOrigBal := extractIsBalanced hWellFormed
      rw [isBalanced, decide_eq_true_eq] at hOrigBal
      cases hOrigBal.1

      case inl hBal =>
        have _ := balanceFactorChange hBal
        tauto

      case inr hBal =>
        cases hBal

        case inl hBal =>
          have _ := balanceFactorChange hBal
          tauto

        case inr hBal =>
          contradiction

    apply And.intro

    . have hOrigBal := extractIsBalanced hWellFormed
      rw [isBalanced, decide_eq_true_eq] at hOrigBal
      exact hOrigBal.2.1

    . tauto

let T := { root := rootₙₑ, wellFormed := hNewRootIsAVLTree }

have hHeightBound := by
  have hOrigBal := extractIsBalanced hWellFormed
  rw [isBalanced, decide_eq_true_eq] at hOrigBal
  cases hOrigBal.1

  case inl hBal =>
    have hHeightRoot := heightRoot (by linarith) hBal
    have hHeightRootₙₑ : T.root.height = 2 := by
      have hHeightLᵣₒ := heightLᵣₒ hBal
      have hEq : T.root = rootₙₑ := by simp only
      rw [hEq, height, hHeightRₙₑ, hHeightLᵣₒ, AVLNode.max]
      tauto

    rw [hHeightRootₙₑ, hHeightRoot]
    tauto

  case inr hBal =>
    cases hBal

    case inl hBal =>
      have hHeightRoot := heightRoot (by linarith) hBal
      have hHeightRootₙₑ : T.root.height = 2 := by
        have hHeightLᵣₒ := heightLᵣₒ hBal
        have hEq : T.root = rootₙₑ := by simp only
        rw [hEq, height, hHeightRₙₑ, hHeightLᵣₒ, AVLNode.max]
        tauto

      rw [hHeightRootₙₑ, hHeightRoot]
      tauto

    case inr hBal => contradiction

have hNewRight : rₙₑ = (@Node α β cmp) 1 kᵢ vᵢ Leaf Leaf := by tauto
have hNewRoot : rootₙₑ = Node (1 + AVLNode.max lᵣₒ.height rₙₑ.height) kᵣₒ vᵣₒ lᵣₒ rₙₑ := by
  unfold_let rootₙₑ
  rw [hCachedHeightRₙₑ, hCachedHeightLᵣₒ]
have hT : T = { root := rootₙₑ, wellFormed := hNewRootIsAVLTree } := by tauto
```

```
  have lessThanProof := insertRightChildLessThanProof h_ro k_ro v_ro l_ro r_ro k_i v_i hNewRight hNewRoot hT
  have greaterThanProof := insertRightChildGreaterThanProof h_ro k_ro v_ro l_ro r_ro k_i v_i hNewRight hNewRoot hT

  have hMapsTo : maps T.root k_i v_i := by
    simp only [maps]
    have : cmp.cmp k_i k_ro = Order.GREATER ∧ (cmp.cmp k_i k_i = Order.EQUAL ∧ True
      ∨ cmp.cmp k_i k_i = Order.LESS ∧ False ∨ cmp.cmp k_i k_i = Order.GREATER ∧ False) := by
      have : cmp.cmp k_i k_i = Order.EQUAL := (cmp.cmpK k_i).2.1
      tauto
    tauto

{ T := T , heightBound := hHeightBound, lessThanProof := lessThanProof, greaterThanProof := greaterThanProof,
  isNodeProof := by tauto, mapsTo := hMapsTo }

structure RebalanceHypotheses {α β : Type} {cmp : ComparisonFunction α}
  (h_ro : ℤ) (k_ro : α) (v_ro : β) (l_ro r_ro : AVLNode α β cmp)
  (k_i : α) (leftOrRight : AVLNode α β cmp)
where
  allLess (k : α) (hKey : cmp.cmp k_i k = Order.LESS) (hOrig : allLess k (Node h_ro k_ro v_ro l_ro r_ro))
    : allLess k leftOrRight
  allGreater (k : α) (hKey : cmp.cmp k_i k = Order.GREATER) (hOrig : allGreater k (Node h_ro k_ro v_ro l_ro r_ro))
    : allGreater k leftOrRight
deriving Repr, DecidableEq

structure RightRotationHypotheses {α β : Type} {cmp : ComparisonFunction α} (left : AVLNode α β cmp) where
  leftIsRightHeavy {h k v l r} (hEqL : left = Node h k v l r) : ¬height r > height l
deriving Repr, DecidableEq

def rightRotation {α β : Type} {cmp : ComparisonFunction α} {l_ro}
  (h_ro : ℤ) (k_ro : α) (v_ro : β)
  (left : AVLNode α β cmp) (right : AVLNode α β cmp)
    {hWellFormed}
  (k_i : α) (v_i : β)
  (hWellFormedL : left.isAVLTree) (hWellFormedR : right.isAVLTree)
  (hBal : left.height = right.height + 2)
  (hH_ro : h_ro = right.height + 2)
  (hLNode : left.isNode) -- linter claims this is unused, but it is needed for the first match
  (hAllLessL : allLess k_ro left)
  (hAllGreaterR : allGreater k_ro right)
  (hRebalance : RebalanceHypotheses h_ro k_ro v_ro l_ro right k_i left)
  (hRightRotation: RightRotationHypotheses left)
  (hCmpK_i : cmp.cmp k_i k_ro = Order.LESS)
  (hMapsL : left.maps k_i v_i)

: (@AVLTreeInsertion α β cmp) k_i v_i { root := Node h_ro k_ro v_ro l_ro right, wellFormed := hWellFormed } h_ro :=

match hEqL : left with
| (Node h_l k_l v_l l_l r_l) =>

have hLeftIsRightHeavy := hRightRotation.leftIsRightHeavy (by rfl)

let r_ne := Node (1 + AVLNode.max (cachedHeight r_l) (cachedHeight right)) k_ro v_ro r_l right
let root_ne := Node (1 + AVLNode.max (cachedHeight l_l) (cachedHeight r_ne)) k_l v_l l_l r_ne

have hCachedHeightL_l := cachedHeightIsCorrect l_l (leftIsAVLTree hWellFormedL)
have hCachedHeightR_l := cachedHeightIsCorrect r_l (rightIsAVLTree hWellFormedL)
have hCachedHeightR := cachedHeightIsCorrect right hWellFormedR
have hCachedHeightR_ne : cachedHeight r_ne = r_ne.height := by
  have h1 : cachedHeight r_ne = 1 + AVLNode.max (cachedHeight r_l) (cachedHeight right) := by tauto
  have h2 : height r_ne = 1 + AVLNode.max (cachedHeight r_l) (cachedHeight right) := by
    rw [height, hCachedHeightR_l, hCachedHeightR]
  rw [h1, h2]

have hOrigBalL: (balanceFactor (Node h_l k_l v_l l_l r_l) = -1 ∨
    balanceFactor (Node h_l k_l v_l l_l r_l) = 0 ∨ balanceFactor (Node h_l k_l v_l l_l r_l) = 1) ∧
    isBalanced l_l = true ∧ isBalanced r_l = true := by
  have hBalL := extractIsBalanced hWellFormedL
  simp only [isBalanced, decide_eq_true_eq] at hBalL
  exact hBalL
```

```
have hBalL : balanceFactor left = -1 ∨ balanceFactor left = 0 := by
  rw [hEqL]
  cases hOrigBalL.1

  case inl => tauto

  case inr hOrigBalLNode =>
    cases hOrigBalLNode

    case inl => tauto

    case inr hOrigBalLNode =>
      simp only [balanceFactor] at hOrigBalLNode
      linarith

have heightsBalanceFactorMinus1 {n : ℤ} (hn : rₗ.height = n) (hBalL : balanceFactor left = -1)
  : (rₗ.height = n) ∧ (lₗ.height = n + 1) ∧ (right.height = n) := by
  have hHeightRₗ : rₗ.height = n := by tauto
  have hHeightLₗ : lₗ.height = n + 1 := by
    simp only [hEqL, balanceFactor, hHeightRₗ] at hBalL
    linarith
  have hOrderingL : lₗ.height > rₗ.height := by linarith
  have hHeightL : left.height = n + 2 := by
    have hHeight : height (Node hₗ kₗ vₗ lₗ rₗ) = 1 + AVLNode.max (height lₗ) (height rₗ) := by tauto
    simp only [hEqL, hHeight, AVLNode.max]
    split <;> linarith
  have hHeightR : right.height = n := by
    rw [←hEqL] at hBal
    linarith
  tauto

have heightsBalanceFactor0 {n : ℤ} (hn : rₗ.height = n) (hBalL : balanceFactor left = 0)
  : (rₗ.height = n) ∧ (lₗ.height = n) ∧ (right.height = n-1) := by
  have hHeightRₗ : rₗ.height = n := by tauto
  have hHeightLₗ : lₗ.height = n := by
    simp only [hEqL, balanceFactor, hHeightRₗ] at hBalL
    linarith
  have hOrderingL : ¬ lₗ.height > rₗ.height := by linarith
  have hHeightL : left.height = n + 1 := by
    have hHeight : height (Node hₗ kₗ vₗ lₗ rₗ) = 1 + AVLNode.max (height lₗ) (height rₗ) := by tauto
    simp only [hEqL, hHeight, AVLNode.max]
    split <;> linarith
  have hHeightR : right.height = n - 1 := by
    rw [←hEqL] at hBal
    linarith
  tauto

have hNewRootIsAVLTree : rootₙₑ.isAVLTree := by
  unfold isAVLTree
  apply And.intro

  . unfold isOrdered

    have hOrigOrderedL := extractIsOrdered hWellFormedL
    unfold isOrdered at hOrigOrderedL

    apply And.intro
    apply And.intro

    . exact hOrigOrderedL.1.1

    . have hGreater : cmp.cmp kᵣₒ kₗ = Order.GREATER := by
        unfold allLess at hAllLessL
        exact cmp.cmpLessToGreater hAllLessL.1

      unfold allGreater
      apply And.intro
```

```
    . exact hGreater

    apply And.intro

    . exact hOrigOrderedL.1.2

    . exact greaterAllGreater hGreater hAllGreaterR

  apply And.intro

  . exact hOrigOrderedL.2.1

  . unfold isOrdered

    apply And.intro
    apply And.intro

    . unfold allLess at hAllLessL
      exact hAllLessL.2.2
    . exact hAllGreaterR

    apply And.intro

    . exact hOrigOrderedL.2.2
    . exact extractIsOrdered hWellFormedR

apply And.intro

. have hOrigHeightL := extractHeightCorrect hWellFormedL
  unfold heightCorrect at hOrigHeightL

  unfold heightCorrect
  apply And.intro

  . rw [hCachedHeightLₗ, hCachedHeightRₙₑ]
    tauto

  apply And.intro

  . tauto

  . unfold heightCorrect
    apply And.intro

    . tauto

    apply And.intro

    . tauto

    . exact extractHeightCorrect hWellFormedR

. rw [isBalanced, decide_eq_true_eq]

  apply And.intro

  . cases hBalL

    case inl hBalL =>
      have hHeights := heightsBalanceFactorMinus1 (by tauto) hBalL
      have hBalₙₑ : balanceFactor rootₙₑ = 0 := by
        have hHeight : height (Node (1 + AVLNode.max (height rₗ) (height right)) kᵣₒ vᵣₒ rₗ right)
          = 1 + AVLNode.max (height rₗ) (height right) := by tauto
        simp only [balanceFactor]
        rw [hCachedHeightRₗ, hCachedHeightR, hHeight, hHeights.1, hHeights.2.1, hHeights.2.2, AVLNode.max]
        simp only [gt_iff_lt, lt_self_iff_false, ↓reduceIte]
        linarith
      tauto
```

```
      case inr hBalL =>
        have hHeights := heightsBalanceFactor0 (by tauto) hBalL
        have hBalₙₑ : balanceFactor rootₙₑ = 1 := by
          have hHeight : height (Node (1 + AVLNode.max (height rₗ) (height right)) kᵣₒ vᵣₒ rₗ right)
            = 1 + AVLNode.max (height rₗ) (height right) := by tauto
          simp only [balanceFactor]
          rw [hCachedHeightRₗ, hCachedHeightR, hHeight, hHeights.1, hHeights.2.1, hHeights.2.2, AVLNode.max]
          simp only [gt_iff_lt, sub_lt_self_iff, zero_lt_one, ↓reduceIte, add_sub_cancel]
        tauto

    apply And.intro

    . tauto

    . rw [isBalanced, decide_eq_true_eq]
      apply And.intro

      . cases hBalL

        case inl hBalL =>
          have hHeights := heightsBalanceFactorMinus1 (by tauto) hBalL
          have hBalRₙₑ : balanceFactor rₙₑ = 0 := by
            simp only [balanceFactor]
            linarith
          tauto
        case inr hBalL =>
          have hHeights := heightsBalanceFactor0 (by tauto) hBalL
          have hBalRₙₑ : balanceFactor rₙₑ = -1 := by
            simp only [balanceFactor]
            linarith
          tauto

      apply And.intro

      . tauto
      . exact extractIsBalanced hWellFormedR

let T := { root := rootₙₑ, wellFormed := hNewRootIsAVLTree }

have hHeightBound := by
  simp only
  unfold height

  cases hBalL

  case inl hBalL =>
    have hHeights := heightsBalanceFactorMinus1 (by tauto) hBalL
    have hEq : height (Node (1 + AVLNode.max (height rₗ) (height right)) kᵣₒ vᵣₒ rₗ right) = height rₗ + 1 := by
      simp only [height, hHeights.2.2, AVLNode.max, gt_iff_lt, lt_self_iff_false, ↓reduceIte]
      linarith
    rw [hCachedHeightRₗ, hCachedHeightR, hEq, hHeights.2.1, hHᵣₒ, hHeights.2.2, AVLNode.max]
    simp only [gt_iff_lt, lt_self_iff_false, ↓reduceIte]
    have : 1 + (height rₗ + 1) = height rₗ + 2 := by linarith
    tauto

  case inr hBalL =>
    have hHeights := heightsBalanceFactor0 (by tauto) hBalL
    have hEq : height (Node (1 + AVLNode.max (height rₗ) (height right)) kᵣₒ vᵣₒ rₗ right) = height rₗ + 1 := by
      simp only [height, hHeights.2.2, AVLNode.max, gt_iff_lt, sub_lt_self_iff, zero_lt_one, ↓reduceIte]
      linarith
    rw [hCachedHeightRₗ, hCachedHeightR, hEq, hHeights.2.1, hHᵣₒ, hHeights.2.2, AVLNode.max]
    simp only [gt_iff_lt, add_lt_iff_neg_left, Int.reduceLT, ↓reduceIte]
    have : 1 + (height rₗ + 1) = height rₗ - 1 + 2 + 1 := by linarith
    tauto

have lessThanProof (k : α) (hKey : cmp.cmp kᵢ k = Order.LESS) (hTree : allLess k (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ right))
  : allLess k T.root := by
  have hL := hRebalance.allLess k hKey hTree
  unfold allLess at hL hTree
```

```
  simp only [allLess]
  tauto

have greaterThanProof (k : α) (hKey : cmp.cmp kᵢ k = Order.GREATER)
  (hTree : allGreater k (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ right)) : allGreater k T.root := by
  have hR := hRebalance.allGreater k hKey hTree
  unfold allGreater at hR hTree
  simp only [allGreater]
  tauto

have hMapsTo : maps T.root kᵢ vᵢ := by
  simp only [maps]
  simp only [maps] at hMapsL
  cases hMapsL
  case inl => tauto
  case inr hMapsL =>
    cases hMapsL
    case inl h => exact Or.inr (Or.inl h)
    case inr h => exact Or.inr (Or.inr (And.intro h.1 (Or.inr (Or.inl (And.intro hCmpKᵢ h.2)))))

{ T := T , heightBound := hHeightBound, lessThanProof := lessThanProof, greaterThanProof := greaterThanProof,
  isNodeProof := by tauto, mapsTo := hMapsTo }

def rebalanceLeft {α β : Type} {cmp : ComparisonFunction α} {lᵣₒ}
  (hᵣₒ : ℤ) (kᵣₒ : α) (vᵣₒ : β)
  (left : AVLNode α β cmp) (right : AVLNode α β cmp)
    {hWellFormed}
  (kᵢ : α) (vᵢ : β)
  (hWellFormedL : left.isAVLTree) (hWellFormedR : right.isAVLTree)
  (hBal : left.height = right.height + 2)
  (hHᵣₒ : hᵣₒ = right.height + 2)
  (hLNode : left.isNode)
  (hAllLessL : allLess kᵣₒ left)
  (hAllGreaterR : allGreater kᵣₒ right)
  (hRebalance : RebalanceHypotheses hᵣₒ kᵣₒ vᵣₒ lᵣₒ right kᵢ left)
  (hCmpKᵢ : cmp.cmp kᵢ kᵣₒ = Order.LESS)
  (hMapsL : left.maps kᵢ vᵢ)

: (@AVLTreeInsertion α β cmp) kᵢ vᵢ { root := Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ right, wellFormed := hWellFormed } hᵣₒ :=

  match hEqL : left with
  | (Node hₗ kₗ vₗ lₗ rₗ) =>

  if hLeftIsRightHeavy : rₗ.height > lₗ.height then
    -- left right rotation

    -- linter claims this is unused, but its needed for the match directly following
    have hRLNode : rₗ.isNode := by
      by_contra hLeftRightNotNode
      have hRLLeaf : rₗ = Leaf := notNodeToLeaf rₗ hLeftRightNotNode
      have hHeightLLNonNegative := heightIsNonNegative lₗ
      rw [hRLLeaf, height] at hLeftIsRightHeavy
      linarith

    match hEqRL : rₗ with
    | (Node hₗᵣ kₗᵣ vₗᵣ lₗᵣ rₗᵣ) =>

    let newLₗ := lₗ
    let newLᵣ := lₗᵣ
    let newRₗ := rₗᵣ
    let newRᵣ := right

    let n := right.height

    have hBalL : balanceFactor left = 1 := by
      have hOrigBalL := extractIsBalanced hWellFormedL
      rw [isBalanced, decide_eq_true_eq, ←hEqL] at hOrigBalL
      cases hOrigBalL.1
```

```
  case inl hOrigBalLNode =>
    simp only [hEqL, balanceFactor, ←hEqRL] at hOrigBalLNode
    rw [←hEqRL] at hLeftIsRightHeavy
    linarith
  case inr hOrigBalLNode =>
    cases hOrigBalLNode

    case inl hOrigBalLNode =>
      simp only [hEqL, balanceFactor, ←hEqRL] at hOrigBalLNode
      rw [←hEqRL] at hLeftIsRightHeavy
      linarith
    case inr h => exact h

have hHeightLₗ : height lₗ = height rₗ - 1 := by
  simp only [hEqL, balanceFactor, ←hEqRL] at hBalL
  linarith

have hHeightRₗ : height rₗ = height left - 1 := by
  have hHeightEq : height left = height rₗ + 1 := by
    rw [←hEqRL] at hEqL
    simp only [hEqL, height]
    unfold AVLNode.max
    have hIf : (if height lₗ > height rₗ then height lₗ else height rₗ) = height rₗ := by split <;> linarith
    rw [hIf]
    linarith
  linarith

have hHeightNewLₗ : newLₗ.height = n := by
  rw [←hEqL] at hBal
  simp only [newLₗ, hHeightLₗ, hHeightRₗ, hBal]
  linarith

have hHeightNewLᵣ : newLᵣ.height = n-1 ∨ newLᵣ.height = n := by
  have hHeight : height rₗ = n+1 := by linarith
  have hBal : rₗ.isBalanced := by
    simp only [isAVLTree, isBalanced, decide_eq_true_eq] at hWellFormedL
    simp only [isBalanced, decide_eq_true_eq, hEqRL]
    tauto
  cases childHeightLeft hHeight hEqRL hBal
  case inl h =>
    have : height lₗᵣ = n - 1 := by linarith
    tauto
  case inr h =>
    simp only [add_sub_cancel] at h
    tauto

have hHeightNewRₗ : newRₗ.height = n-1 ∨ newRₗ.height = n := by
  have hHeight : height rₗ = n+1 := by linarith
  have hBal : rₗ.isBalanced := by
    simp only [isAVLTree, isBalanced, decide_eq_true_eq] at hWellFormedL
    simp only [isBalanced, decide_eq_true_eq, hEqRL]
    tauto
  cases childHeightRight hHeight hEqRL hBal
  case inl h =>
    have : height rₗᵣ = n - 1 := by linarith
    tauto
  case inr h =>
    simp only [add_sub_cancel] at h
    tauto

have hHeightRᵣ : newRᵣ.height = n := by tauto

let lₙₑ := Node (1 + AVLNode.max (cachedHeight newLₗ) (cachedHeight newLᵣ)) kₗ vₗ newLₗ newLᵣ
let rₙₑ := Node (1 + AVLNode.max (cachedHeight newRₗ) (cachedHeight newRᵣ)) kᵣₒ vᵣₒ newRₗ newRᵣ
let rootₙₑ := Node (1 + AVLNode.max (cachedHeight lₙₑ) (cachedHeight rₙₑ)) kᵣ vᵣ lₙₑ rₙₑ

have hCachedHeightLₙₑ : cachedHeight lₙₑ = lₙₑ.height := by
  have hNewLLCachedHeight := cachedHeightIsCorrect newLₗ (leftIsAVLTree hWellFormedL)
  have hNewLRCachedHeight := cachedHeightIsCorrect newLᵣ (leftIsAVLTree (rightIsAVLTree hWellFormedL))
```

```
  have h1 : cachedHeight lₙₑ = 1 + AVLNode.max (cachedHeight newLₗ) (cachedHeight newLᵣ) := by tauto
  have h2 : height lₙₑ = 1 + AVLNode.max (cachedHeight newLₗ) (cachedHeight newLᵣ) := by
    rw [height, hNewLLCachedHeight, hNewLRCachedHeight]
  rw [h1, h2]
have hCachedHeightRₙₑ : cachedHeight rₙₑ = rₙₑ.height := by
  have hNewRLCachedHeight := cachedHeightIsCorrect newRₗ (rightIsAVLTree (rightIsAVLTree hWellFormedL))
  have hNewRRCachedHeight := cachedHeightIsCorrect newRᵣ hWellFormedR
  have h1 : cachedHeight rₙₑ = 1 + AVLNode.max (cachedHeight newRₗ) (cachedHeight newRᵣ) := by tauto
  have h2 : height rₙₑ = 1 + AVLNode.max (cachedHeight newRₗ) (cachedHeight newRᵣ) := by
    rw [height, hNewRLCachedHeight, hNewRRCachedHeight]
  rw [h1, h2]

have hNewRootIsAVLTree : rootₙₑ.isAVLTree := by
  unfold isAVLTree
  apply And.intro

  . have hOrigOrdered := extractIsOrdered hWellFormedL
    unfold isOrdered at hOrigOrdered
    have hOrigOrderedLᵣ := hOrigOrdered.2.2
    unfold isOrdered at hOrigOrderedLᵣ
    unfold isOrdered
    apply And.intro
    apply And.intro

    . simp only [allLess, newLₗ, newLᵣ]
      have hCmp : cmp.cmp kₗ kₗᵣ = Order.LESS := by
        have hOrderingLᵣ := hOrigOrdered.1.2
        unfold allGreater at hOrderingLᵣ
        exact cmp.cmpGreaterToLess hOrderingLᵣ.1
      apply And.intro

      . exact hCmp

      apply And.intro

      . exact lessAllLess hCmp hOrigOrdered.1.1
      . exact hOrigOrderedLᵣ.1.1

    . have hCmp : cmp.cmp kᵣₒ kₗᵣ = Order.GREATER := by
        simp only [allLess] at hAllLessL
        exact cmp.cmpLessToGreater hAllLessL.2.2.1
      simp only [allGreater, newRₗ, newRᵣ]
      apply And.intro

      . exact hCmp

      apply And.intro

      . exact hOrigOrderedLᵣ.1.2
      . exact greaterAllGreater hCmp (by tauto)

    apply And.intro

    . simp only [isOrdered, newLₗ, newLᵣ]
      apply And.intro
      apply And.intro

      . exact hOrigOrdered.1.1
      . unfold allGreater at hOrigOrdered
        exact hOrigOrdered.1.2.2.1

      apply And.intro

      . exact hOrigOrdered.2.1
      . exact hOrigOrderedLᵣ.2.1

    . simp only [isOrdered, newRₗ, newRᵣ]
      apply And.intro
      apply And.intro
```

```
    . simp only [allLess] at hAllLessL
      exact hAllLessL.2.2.2.2
    . exact hAllGreaterR

    apply And.intro

    . exact hOrigOrderedLᵣ.2.2
    . unfold isAVLTree at hWellFormedR
      exact hWellFormedR.1

apply And.intro

. have hOrigHeight := extractHeightCorrect hWellFormedL
  unfold heightCorrect at hOrigHeight
  unfold heightCorrect
  apply And.intro

  . rw [hCachedHeightLₙₑ, hCachedHeightRₙₑ]
    tauto

  apply And.intro

  . simp only [heightCorrect, newLₗ, newLᵣ]
    apply And.intro

    . tauto

    apply And.intro

    . exact hOrigHeight.2.1
    . unfold heightCorrect at hOrigHeight
      exact hOrigHeight.2.2.2.1

  . simp only [heightCorrect, newRₗ, newRᵣ]
    apply And.intro

    . tauto

    apply And.intro

    . unfold heightCorrect at hOrigHeight
      exact hOrigHeight.2.2.2.2
    . unfold isAVLTree at hWellFormedR
      exact hWellFormedR.2.1

. simp only [isBalanced, decide_eq_true_eq, balanceFactor, height, AVLNode.max]
  apply And.intro

  . cases hHeightNewLᵣ
    case inl hHeightNewLᵣ =>
      cases hHeightNewRₗ
      case inl hHeightRₗ =>
        have hIf : (if n - 1 > n then n - 1 else n) = n := by simp only [gt_iff_lt, ite_eq_right_iff,
          sub_eq_self, one_ne_zero, imp_false, not_lt, tsub_le_iff_right, le_add_iff_nonneg_right,
          zero_le_one]
        have hIf' : (if n > n - 1 then n else n - 1) = n := by simp only [gt_iff_lt, sub_lt_self_iff,
          zero_lt_one, ↓reduceIte]
        have : 1 + n - (1 + n) = 0 := by linarith
        simp only [hHeightNewLₗ, hHeightNewLᵣ, hHeightRₗ, hHeightRₗ, hIf, hIf']
        tauto
      case inr hHeightRₗ =>
        simp only [hHeightNewLₗ, hHeightNewLᵣ, hHeightRₗ, hHeightRₗ, gt_iff_lt, lt_self_iff_false,
          ↓reduceIte, sub_lt_self_iff, zero_lt_one, sub_self, Int.reduceNeg, zero_eq_neg, one_ne_zero,
          zero_ne_one, or_false, or_true]
    case inr hHeightNewLᵣ =>
      cases hHeightNewRₗ
      case inl hHeightRₗ =>
        have hIf : (if n - 1 > n then n - 1 else n) = n := by simp only [gt_iff_lt, ite_eq_right_iff,
```

```
            sub_eq_self, one_ne_zero, imp_false, not_lt, tsub_le_iff_right, le_add_iff_nonneg_right,
            zero_le_one]
          have hIf' : (if n > n then n else n) = n := by simp only [gt_iff_lt, lt_self_iff_false, ↓reduceIte]
          have : 1 + n - (1 + n) = 0 := by linarith
          simp only [hHeightNewLₗ, hHeightNewLᵣ, hHeightRₗ, hHeightRₗ, hIf, hIf']
          tauto
        case inr hHeightRₗ =>
          simp only [hHeightNewLₗ, hHeightNewLᵣ, hHeightRₗ, hHeightRₗ, gt_iff_lt, lt_self_iff_false,
            ↓reduceIte, sub_self, Int.reduceNeg, zero_eq_neg, one_ne_zero, zero_ne_one, or_false, or_true]
    apply And.intro

    . apply And.intro

      . cases hHeightNewLᵣ
        case inl hHeightNewLᵣ =>
          simp only [hHeightNewLₗ, hHeightNewLᵣ, sub_sub_cancel_left, Int.reduceNeg, neg_eq_zero,
            one_ne_zero, neg_eq_self_iff, or_self, or_false]
        case inr hHeightNewLᵣ =>
          simp only [hHeightNewLₗ, hHeightNewLᵣ, sub_self, Int.reduceNeg, zero_eq_neg, one_ne_zero,
            zero_ne_one, or_false, or_true]

      . simp only [isAVLTree, isBalanced, decide_eq_true_eq] at hWellFormedL
        tauto

    . apply And.intro

      . cases hHeightNewRₗ
        case inl hHeightRₗ =>
          simp only [hHeightRₗ, hHeightRₗ, sub_sub_cancel, Int.reduceNeg, eq_neg_self_iff, one_ne_zero, or_true]
        case inr hHeightRₗ =>
          simp only [hHeightRₗ, hHeightRₗ, sub_self, Int.reduceNeg, zero_eq_neg, one_ne_zero, zero_ne_one,
            or_false, or_true]

      . simp only [isAVLTree, isBalanced, decide_eq_true_eq] at hWellFormedL hWellFormedR
        tauto

let T := { root := rootₙₑ, wellFormed := hNewRootIsAVLTree }

have hHeightBound := by
  have : height T.root = hᵣₒ := by
    have hNewLeft : height lₙₑ = n + 1 := by
      cases hHeightNewLᵣ
      case inl hHeightNewLᵣ =>
        rw [height, hHeightNewLₗ, hHeightNewLᵣ, AVLNode.max]
        simp only [gt_iff_lt, sub_lt_self_iff, zero_lt_one, ↓reduceIte]
        linarith
      case inr hHeightNewLᵣ =>
        rw [height, hHeightNewLₗ, hHeightNewLᵣ, AVLNode.max]
        simp only [gt_iff_lt, lt_self_iff_false, ↓reduceIte]
        linarith
    have hRₙₑ : height rₙₑ = n + 1 := by
      cases hHeightNewRₗ
      case inl hHeightRₗ =>
        have hIf : (if n - 1 > n then n - 1 else n) = n := by split <;> linarith
        rw [height, hHeightRₗ, hHeightRᵣ, AVLNode.max, hIf]
        linarith
      case inr hHeightRₗ =>
        rw [height, hHeightRₗ, hHeightRᵣ, AVLNode.max]
        simp only [gt_iff_lt, lt_self_iff_false, ↓reduceIte]
        linarith
    have hOldRoot : hᵣₒ = n + 2 := by linarith
    rw [height, hNewLeft, hRₙₑ, hOldRoot, AVLNode.max]
    simp only [gt_iff_lt, lt_self_iff_false, ↓reduceIte]
    linarith
  tauto

have lessThanProof (k : α) (hKey : cmp.cmp kᵢ k = Order.LESS) (hTree : allLess k (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ right))

  : allLess k T.root := by
```

```
    have hKᵣₒ : cmp.cmp kᵣₒ k = Order.LESS := by
      unfold allLess at hTree
      exact hTree.1
    simp only [allLess] at hAllLessL
    simp only [allLess]
    apply And.intro

    . exact cmp.cmpTransitiveLess hAllLessL.2.2.1 hKᵣₒ

    apply And.intro

    . apply And.intro

      . exact cmp.cmpTransitiveLess hAllLessL.1 hKᵣₒ

      apply And.intro

      . exact lessAllLess hKᵣₒ hAllLessL.2.1
      . exact lessAllLess hKᵣₒ hAllLessL.2.2.2.1

    . apply And.intro

      . exact hKᵣₒ

      apply And.intro

      . exact lessAllLess hKᵣₒ hAllLessL.2.2.2.2
      . unfold allLess at hTree
        exact hTree.2.2

  have greaterThanProof (k : α) (hKey : cmp.cmp kᵢ k = Order.GREATER)
    (hTree : allGreater k (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ right)) : allGreater k T.root := by
    have hKᵣₒ : cmp.cmp kᵣₒ k = Order.GREATER := by
      unfold allGreater at hTree
      exact hTree.1
    have hAllGreaterL := hRebalance.allGreater k hKey hTree
    simp only [allGreater] at hAllGreaterL
    simp only [allGreater]
    apply And.intro

    . exact hAllGreaterL.2.2.1

    apply And.intro

    . apply And.intro

      . exact hAllGreaterL.1

      apply And.intro

      . exact hAllGreaterL.2.1
      . exact hAllGreaterL.2.2.2.1

    . apply And.intro

      . exact hKᵣₒ

      apply And.intro

      . exact hAllGreaterL.2.2.2.2
      . unfold allGreater at hTree
        exact hTree.2.2

  have hMapsTo : maps T.root kᵢ vᵢ := by
    have hCmp : cmp.cmp kₗ kₗᵣ = Order.LESS := by
      have hCmp' : cmp.cmp kₗᵣ kₗ = Order.GREATER := by
        simp only [isAVLTree, isOrdered, allGreater] at hWellFormedL
        tauto
      exact cmp.cmpGreaterToLess hCmp'
```

```
    simp only [maps]
    simp only [maps] at hMapsL
    cases hMapsL
    case inl h =>
      exact Or.inr (Or.inl (And.intro (by rw [←(applyIff (cmp.cmpEq kᵢ kₗ) h.1)] at hCmp; exact hCmp) (Or.inl h)))

    case inr hMapsL =>
      cases hMapsL
      case inl h => exact Or.inr (Or.inl (And.intro (cmp.cmpTransitiveLess h.1 hCmp) (Or.inr (Or.inl h))))
      case inr h =>
        cases h.2
        case inl h' => exact Or.inl h'
        case inr h' =>
          cases h'
          case inl h' => exact Or.inr (Or.inl (And.intro h'.1 (Or.inr (Or.inr (And.intro h.1 h'.2)))))
          case inr h' => exact Or.inr (Or.inr (And.intro h'.1 (Or.inr (Or.inl (And.intro hCmpKᵢ h'.2)))))

  { T := T , heightBound := hHeightBound, lessThanProof := lessThanProof, greaterThanProof := greaterThanProof,
    isNodeProof := by tauto, mapsTo := hMapsTo }
  else
    -- single right rotation
    have leftIsRightHeavy {h k v l r} (hEqL' : left = Node h k v l r) : ¬height r > height l := by
      simp only [hEqL, Node.injEq] at hEqL'
      rw [hEqL'.2.2.2.1, hEqL'.2.2.2.2] at hLeftIsRightHeavy
      exact hLeftIsRightHeavy
    rightRotation hᵣₒ kᵣₒ vᵣₒ left right kᵢ vᵢ (by rw [←hEqL] at hWellFormedL; tauto) hWellFormedR
      (by rw [←hEqL] at hBal; tauto) hHᵣₒ (by rw [←hEqL] at hLNode; tauto) (by rw [←hEqL] at hAllLessL; tauto)
      hAllGreaterR (by rw [←hEqL] at hRebalance; tauto) {leftIsRightHeavy := leftIsRightHeavy} hCmpKᵢ
      (by rw [←hEqL] at hMapsL; tauto)


def insertLeftSubtree {α β : Type} {cmp : ComparisonFunction α} {hₗ kₗ vₗ lₗ rₗ hWellFormedL sibling}
  (hᵣₒ : ℤ) (kᵣₒ : α) (vᵣₒ : β) (lᵣₒ rᵣₒ : AVLNode α β cmp)
  (hWellFormed : (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ).isAVLTree)
  (kᵢ : α) (vᵢ : β) (lₙₑ : AVLTreeInsertion kᵢ vᵢ { root := Node hₗ kₗ vₗ lₗ rₗ, wellFormed := hWellFormedL } hₗ)
  (hMatch : cmp.cmp kᵢ kᵣₒ = Order.LESS ∧ lᵣₒ = Node hₗ kₗ vₗ lₗ rₗ ∧ rᵣₒ = sibling)

: AVLTreeInsertion kᵢ vᵢ { root := Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ, wellFormed := hWellFormed } hᵣₒ :=

let rootₙₑ := Node (1 + AVLNode.max (cachedHeight lₙₑ.T.root) (cachedHeight rᵣₒ)) kᵣₒ vᵣₒ lₙₑ.T.root rᵣₒ

have hCachedHeightLₙₑ := cachedHeightIsCorrect lₙₑ.T.root lₙₑ.T.wellFormed
have hCachedHeightRᵣₒ := cachedHeightIsCorrect rᵣₒ (rightIsAVLTree hWellFormed)

if hBal : rootₙₑ.isBalanced then
  have hNewRootIsAVLTree : rootₙₑ.isAVLTree := by
    have hWellFormedLₙₑ := lₙₑ.T.wellFormed
    unfold isAVLTree at hWellFormedLₙₑ

    unfold isAVLTree
    apply And.intro

    . unfold isOrdered
      have hOrigOrdered := extractIsOrdered hWellFormed
      unfold isOrdered at hOrigOrdered
      apply And.intro
      apply And.intro

      . have hTree : allLess kᵣₒ (Node hₗ kₗ vₗ lₗ rₗ) := by
          rw [←hMatch.2.1]
          tauto
        exact lₙₑ.lessThanProof kᵣₒ hMatch.1 hTree
      . exact hOrigOrdered.1.2

      apply And.intro

      . exact hWellFormedLₙₑ.1
      . exact hOrigOrdered.2.2
```

```
  apply And.intro

  . have hOrigHeight := extractHeightCorrect hWellFormed
    unfold heightCorrect at hOrigHeight
    rw [heightCorrect, hCachedHeightLₙₑ, hCachedHeightRᵣₒ]
    tauto

  . exact hBal

let T := { root := rootₙₑ, wellFormed := hNewRootIsAVLTree }

have hHeightBound := by
  have hHₗ : height lᵣₒ = hₗ := by
    have hOrigHeight := extractHeightCorrect hWellFormed
    unfold heightCorrect at hOrigHeight
    have hHeightL := hOrigHeight.2.1
    rw [hMatch.2.1, heightCorrect, ←hMatch.2.1] at hHeightL
    tauto

  have hHᵣₒ : hᵣₒ = 1 + AVLNode.max (height lᵣₒ) (height rᵣₒ) := by
    have hOrigHeight := extractHeightCorrect hWellFormed
    unfold heightCorrect at hOrigHeight
    tauto

  cases lₙₑ.heightBound

  case inl hBound =>
    have hOrigHeight := extractHeightCorrect hWellFormed
    unfold heightCorrect at hOrigHeight
    have hHeightLᵣₒ := hOrigHeight.2.1
    have hLᵣₒ : lᵣₒ = Node hₗ kₗ vₗ lₗ rₗ := by tauto
    rw [hMatch.2.1, heightCorrect] at hHeightLᵣₒ
    rw [height, hBound, hHeightLᵣₒ.1, ←hLᵣₒ]
    tauto

  case inr hBound =>
    if hOrdering : height lₙₑ.T.root ≤ rᵣₒ.height then
      have hOldHeight : hᵣₒ = rᵣₒ.height + 1 := by
        have hIf : (if height lᵣₒ > height rᵣₒ then height lᵣₒ else height rᵣₒ) = height rᵣₒ := by
          split <;> linarith
        rw [AVLNode.max, hIf] at hHᵣₒ
        linarith
      have hNewHeight : T.root.height = rᵣₒ.height + 1 := by
        have h : height T.root = 1 + AVLNode.max (height lₙₑ.T.root) (height rᵣₒ) := by tauto
        have hIf : (if height lₙₑ.T.root > height rᵣₒ then height lₙₑ.T.root else height rᵣₒ) = height rᵣₒ := by
          split <;> linarith
        rw [h, AVLNode.max, hIf]
        linarith
      rw [hOldHeight, hNewHeight]
      tauto
    else
      have hOldHeight : hᵣₒ = lᵣₒ.height + 1 := by
        have hIf : (if height lᵣₒ > height rᵣₒ then height lᵣₒ else height rᵣₒ) = height lᵣₒ := by
          split <;> linarith
        rw [AVLNode.max, hIf] at hHᵣₒ
        linarith
      have hNewHeight : T.root.height = lᵣₒ.height + 2 := by
        rw [hHₗ, height, hBound, ←hHₗ, AVLNode.max]
        split <;> linarith
      rw [hOldHeight, hNewHeight]
      have _ : height lᵣₒ + 2 = height lᵣₒ + 1 + 1 := by linarith
      tauto

have lessThanProof (k : α) (hKey : cmp.cmp kᵢ k = Order.LESS) (hTree : allLess k (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ))
  : allLess k T.root := by
  simp only [allLess]
  unfold allLess at hTree
  apply And.intro
```

```
    . unfold allLess at hTree
      exact hTree.1

    apply And.intro

    . have hTree' : allLess k (Node hₗ kₗ vₗ lₗ rₗ) := by
        rw [hMatch.2.1] at hTree
        exact hTree.2.1
      exact lₙₑ.lessThanProof k hKey hTree'
    . exact hTree.2.2

  have greaterThanProof (k : α) (hKey : cmp.cmp kᵢ k = Order.GREATER)
    (hTree : allGreater k (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ)) : allGreater k T.root := by
    simp only [allGreater]
    unfold allGreater at hTree
    apply And.intro

    . unfold allGreater at hTree
      exact hTree.1

    apply And.intro

    . have hTree' : allGreater k (Node hₗ kₗ vₗ lₗ rₗ) := by
        rw [hMatch.2.1] at hTree
        exact hTree.2.1
      exact lₙₑ.greaterThanProof k hKey hTree'
    . exact hTree.2.2

  have hMapsTo : maps T.root kᵢ vᵢ := by
    simp only [maps]
    exact Or.inr (Or.inl (And.intro hMatch.1 lₙₑ.mapsTo))

  { T := T , heightBound := hHeightBound, lessThanProof := lessThanProof, greaterThanProof := greaterThanProof,
    isNodeProof := by tauto, mapsTo := hMapsTo }
else
  have hWellFormedL : lₙₑ.T.root.isAVLTree := lₙₑ.T.wellFormed
  have hWellFormedR : rᵣₒ.isAVLTree := rightIsAVLTree hWellFormed
  have hBal : lₙₑ.T.root.height = rᵣₒ.height + 2 := by
    have hOrigBal := extractIsBalanced hWellFormed
    rw [isBalanced, decide_eq_true_eq] at hOrigBal
    unfold isAVLTree at hWellFormedL hWellFormedR

    have hHeightLₙₑ : lₙₑ.T.root.height = lᵣₒ.height + 1 := by
      have hHₗ : height lᵣₒ = hₗ := by
        have hOrigHeight := extractHeightCorrect hWellFormed
        rw [heightCorrect, hMatch.2.1, heightCorrect, ←hMatch.2.1] at hOrigHeight
        tauto

      cases lₙₑ.heightBound

      case inl hHeightLₙₑ =>
        have hBal' : isBalanced rootₙₑ = true := by
          simp only [isBalanced, balanceFactor, decide_eq_true_eq, hHeightLₙₑ]
          apply And.intro

          . simp only [balanceFactor, hHₗ] at hOrigBal
            tauto

          . tauto
        contradiction
      case inr hHeightLₙₑ =>
        rw [hHeightLₙₑ, hHₗ]

    have hBalOld : lᵣₒ.height = rᵣₒ.height + 1 := by
      cases hOrigBal.1

      case inl hBalOld' =>
        simp only [balanceFactor] at hBalOld'
        linarith
```

```
    case inr hBalOld' =>
      cases hBalOld'

      case inl hBalOld' =>
        simp only [balanceFactor] at hBalOld'
        have hEq : height l_ro = height r_ro := by linarith
        rw [hEq] at hHeightL_ne
        have hBal' : isBalanced root_ne = true := by
          simp only [isBalanced, decide_eq_true_eq, hHeightL_ne, balanceFactor]
          have _ : height r_ro - (height r_ro + 1) = -1 := by linarith
          tauto
        contradiction

      case inr hBalOld' =>
        simp only [balanceFactor] at hBalOld'
        have hEq : height l_ro = height r_ro - 1 := by linarith
        rw [hEq] at hHeightL_ne
        have hBal' : isBalanced root_ne = true := by
          simp only [isBalanced, decide_eq_true_eq, hHeightL_ne, balanceFactor]
          have _ : height r_ro - (height r_ro - 1 + 1) = 0 := by linarith
          tauto
        contradiction

  rw [hHeightL_ne, hBalOld]
  linarith
have hAllLessL : allLess k_ro l_ne.T.root := by
  have hTree : allLess k_ro (Node h_i k_i v_i l_i r_i) := by
    rw [←hMatch.2.1]
    have hOrigOrdered := extractIsOrdered hWellFormed
    unfold isOrdered at hOrigOrdered
    exact hOrigOrdered.1.1
  exact l_ne.lessThanProof k_ro hMatch.1 hTree
have hAllGreaterR : allGreater k_ro r_ro := by
  have hOrigOrdered := extractIsOrdered hWellFormed
  unfold isOrdered at hOrigOrdered
  exact hOrigOrdered.1.2

have hRebalanceAllLess (k : α) (hKey : cmp.cmp k_i k = Order.LESS) (hOrig : allLess k (Node h_ro k_ro v_ro l_ro r_ro))

  : allLess k l_ne.T.root := by
  have hL : allLess k (Node h_i k_i v_i l_i r_i) := by
    rw [hMatch.2.1, allLess] at hOrig
    exact hOrig.2.1
  exact l_ne.lessThanProof k hKey hL
have hRebalanceAllGreater (k : α) (hKey : cmp.cmp k_i k = Order.GREATER)
  (hOrig : allGreater k (Node h_ro k_ro v_ro l_ro r_ro)) : allGreater k l_ne.T.root := by
  have hr : allGreater k (Node h_i k_i v_i l_i r_i) := by
    rw [hMatch.2.1, allGreater] at hOrig
    exact hOrig.2.1
  exact l_ne.greaterThanProof k hKey hr
have hRebalance := {allLess := hRebalanceAllLess, allGreater := hRebalanceAllGreater}

have hH_ro := by
  have hHeightLIncreased : l_ne.T.root.height = l_ro.height + 1 := by
    have hOrigHeight := extractHeightCorrect hWellFormed
    simp only [heightCorrect, hMatch.2.1] at hOrigHeight
    rw [←hMatch.2.1] at hOrigHeight
    cases l_ne.heightBound

    case inl hHeight =>
      have hBal' : root_ne.isBalanced := by
        have hH_i : h_i = l_ro.height := hOrigHeight.2.1.1
        have hOrigBal := extractIsBalanced hWellFormed
        simp only [isBalanced, decide_eq_true_eq, balanceFactor] at hOrigBal
        simp only [isBalanced, decide_eq_true_eq, balanceFactor, hHeight, hH_i]
        have _ := extractIsBalanced hWellFormedL
        tauto
      contradiction
```

```
      case inr hHeight => linarith
    have hLRootGreater : l_ro.height > r_ro.height := by linarith
    have hOrigHeight := extractHeightCorrect hWellFormed
    simp only [heightCorrect, height] at hOrigHeight
    rw [hOrigHeight.1, AVLNode.max]
    split <;> linarith

  rebalanceLeft h_ro k_ro v_ro l_ne.T.root r_ro k_i v_i hWellFormedL hWellFormedR hBal hH_ro l_ne.isNodeProof hAllLessL
    hAllGreaterR hRebalance hMatch.1 l_ne.mapsTo

structure LeftRotationHypotheses {α β : Type} {cmp : ComparisonFunction α} (right : AVLNode α β cmp) where
  rightIsLeftHeavy {h k v l r} (hEqR : right = Node h k v l r) : ¬height l > height r
deriving Repr, DecidableEq

def leftRotation {α β : Type} {cmp : ComparisonFunction α} {r_ro}
  (h_ro : ℤ) (k_ro : α) (v_ro : β)
  (left : AVLNode α β cmp) (right : AVLNode α β cmp)
    {hWellFormed}
  (k_i : α) (v_i : β)
  (hWellFormedL : left.isAVLTree) (hWellFormedR : right.isAVLTree)
  (hBal : right.height = left.height + 2)
  (hH_ro : h_ro = left.height + 2)
  (hRNode : right.isNode) -- linter claims this is unused, but it is needed for the first match
  (hAllLessL : allLess k_ro left)
  (hAllGreaterR : allGreater k_ro right)
  (hRebalance : RebalanceHypotheses h_ro k_ro v_ro left r_ro k_i right)
  (hLeftRotation: LeftRotationHypotheses right)
  (hCmpK_i : cmp.cmp k_i k_ro = Order.GREATER)
  (hMapsR : right.maps k_i v_i)

: (@AVLTreeInsertion α β cmp) k_i v_i { root := Node h_ro k_ro v_ro left r_ro, wellFormed := hWellFormed } h_ro :=

match hEqR : right with
| (Node h_r k_r v_r l_r r_r) =>

have hRightIsLeftHeavy := hLeftRotation.rightIsLeftHeavy (by rfl)

let l_ne := Node (1 + AVLNode.max (cachedHeight left) (cachedHeight l_r)) k_ro v_ro left l_r
let root_ne := Node (1 + AVLNode.max (cachedHeight l_ne) (cachedHeight r_r)) k_r v_r l_ne r_r

have hCachedHeightR_r := cachedHeightIsCorrect r_r (rightIsAVLTree hWellFormedR)
have hCachedHeightL_r := cachedHeightIsCorrect l_r (leftIsAVLTree hWellFormedR)
have hCachedHeightL := cachedHeightIsCorrect left hWellFormedL
have hCachedHeightL_ne : cachedHeight l_ne = l_ne.height := by
  have h1 : cachedHeight l_ne = 1 + AVLNode.max (cachedHeight left) (cachedHeight l_r) := by tauto
  have h2 : height l_ne = 1 + AVLNode.max (cachedHeight left) (cachedHeight l_r) := by
    rw [height, hCachedHeightL_r, hCachedHeightL]
  rw [h1, h2]

have hOrigBalR: (balanceFactor (Node h_r k_r v_r l_r r_r) = -1 ∨
    balanceFactor (Node h_r k_r v_r l_r r_r) = 0 ∨ balanceFactor (Node h_r k_r v_r l_r r_r) = 1) ∧
    isBalanced l_r = true ∧ isBalanced r_r = true := by
  have hBalR := extractIsBalanced hWellFormedR
  simp only [isBalanced, decide_eq_true_eq] at hBalR
  exact hBalR

have hBalR : balanceFactor right = 0 ∨ balanceFactor right = 1 := by
  rw [hEqR]
  cases hOrigBalR.1

  case inl hOrigBalRNode =>
    simp only [balanceFactor] at hOrigBalRNode
    linarith

  case inr hOrigBalRNode =>
    cases hOrigBalRNode

    case inl => tauto
```

```
    case inr hOrigBalRNode => tauto


have heightsBalanceFactor1 {n : ℤ} (hn : lᵣ.height = n) (hBalR : balanceFactor right = 1)
  : (lᵣ.height = n) ∧ (rᵣ.height = n + 1) ∧ (left.height = n) := by
  have hHeightLᵣ : lᵣ.height = n := by tauto
  have hHeightRᵣ : rᵣ.height = n + 1 := by
    simp only [hEqR, balanceFactor, hHeightLᵣ] at hBalR
    linarith
  have hOrderingR : rᵣ.height > lᵣ.height := by linarith
  have hHeightR : right.height = n + 2 := by
    have hHeight : height (Node hᵣ kᵣ vᵣ lᵣ rᵣ) = 1 + AVLNode.max (height lᵣ) (height rᵣ) := by tauto
    simp only [hEqR, hHeight, AVLNode.max]
    split <;> linarith
  have hHeightL : left.height = n := by
    rw [←hEqR] at hBal
    linarith
  tauto

have heightsBalanceFactor0 {n : ℤ} (hn : lᵣ.height = n) (hBalR : balanceFactor right = 0)
  : (lᵣ.height = n) ∧ (rᵣ.height = n) ∧ (left.height = n-1) := by
  have hHeightLᵣ : lᵣ.height = n := by tauto
  have hHeightRᵣ : rᵣ.height = n := by
    simp only [hEqR, balanceFactor, hHeightLᵣ] at hBalR
    linarith
  have hOrderingR : ¬ rᵣ.height > lᵣ.height := by linarith
  have hHeightR : right.height = n + 1 := by
    have hHeight : height (Node hᵣ kᵣ vᵣ lᵣ rᵣ) = 1 + AVLNode.max (height lᵣ) (height rᵣ) := by tauto
    simp only [hEqR, hHeight, AVLNode.max]
    split <;> linarith
  have hHeightL : left.height = n - 1 := by
    rw [←hEqR] at hBal
    linarith
  tauto

have hNewRootIsAVLTree : rootₙₑ.isAVLTree := by
  unfold isAVLTree
  apply And.intro

  . unfold isOrdered

    have hOrigOrderedR := extractIsOrdered hWellFormedR
    unfold isOrdered at hOrigOrderedR

    apply And.intro
    apply And.intro

    . have hLess : cmp.cmp kᵣₒ kᵣ = Order.LESS := by
        unfold allGreater at hAllGreaterR
        exact cmp.cmpGreaterToLess hAllGreaterR.1

      unfold allLess
      apply And.intro

      . exact hLess

      apply And.intro

      . exact lessAllLess hLess hAllLessL

      . exact hOrigOrderedR.1.1

    . exact hOrigOrderedR.1.2

    apply And.intro

    . unfold isOrdered

      apply And.intro
```

```
    apply And.intro

    . exact hAllLessL
    . unfold allGreater at hAllGreaterR
      exact hAllGreaterR.2.1

    apply And.intro

    . exact extractIsOrdered hWellFormedL
    . exact hOrigOrderedR.2.1

  . exact hOrigOrderedR.2.2

apply And.intro

. have hOrigHeightR := extractHeightCorrect hWellFormedR
  unfold heightCorrect at hOrigHeightR

  unfold heightCorrect
  apply And.intro

  . rw [hCachedHeightRᵣ, hCachedHeightLₙₑ]
    tauto

  apply And.intro

  . unfold heightCorrect
    apply And.intro

    . tauto

    apply And.intro

    . exact extractHeightCorrect hWellFormedL

    . tauto

  . tauto

. rw [isBalanced, decide_eq_true_eq]

  apply And.intro

  . cases hBalR

    case inl hBalR =>
      have hHeights := heightsBalanceFactor0 (by tauto) hBalR
      have hBalₙₑ : balanceFactor rootₙₑ = -1 := by
        have hHeight : height (Node (1 + AVLNode.max (height left) (height lᵣ)) kᵣₒ vᵣₒ left lᵣ)
          = 1 + AVLNode.max (height left) (height lᵣ) := by tauto
        simp only [balanceFactor]
        rw [hCachedHeightLᵣ, hCachedHeightL, hHeight, hHeights.1, hHeights.2.1, hHeights.2.2, AVLNode.max]
        have hIf : (if height lᵣ < height lᵣ - 1 then height lᵣ - 1 else height lᵣ) = height lᵣ := by
          split <;> linarith
        rw [hIf]
        linarith
      tauto

    case inr hBalR =>
      have hHeights := heightsBalanceFactor1 (by tauto) hBalR
      have hBalₙₑ : balanceFactor rootₙₑ = 0 := by
        have hHeight : height (Node (1 + AVLNode.max (height left) (height lᵣ)) kᵣₒ vᵣₒ left lᵣ)
          = 1 + AVLNode.max (height left) (height lᵣ) := by tauto
        simp only [balanceFactor]
        rw [hCachedHeightLᵣ, hCachedHeightL, hHeight, hHeights.1, hHeights.2.1, hHeights.2.2, AVLNode.max]
        simp only [gt_iff_lt, lt_self_iff_false, ↓reduceIte]
        linarith
      tauto
```

```
    apply And.intro

    . rw [isBalanced, decide_eq_true_eq]
      apply And.intro

      . cases hBalR

        case inl hBalR =>
          have hHeights := heightsBalanceFactor0 (by tauto) hBalR
          have hBalLₙₑ : balanceFactor lₙₑ = 1 := by
            simp only [balanceFactor]
            linarith
          tauto
        case inr hBalR =>
          have hHeights := heightsBalanceFactor1 (by tauto) hBalR
          have hBalLₙₑ : balanceFactor lₙₑ = 0 := by
            simp only [balanceFactor]
            linarith
          tauto

      apply And.intro

      . exact extractIsBalanced hWellFormedL
      . tauto

    . tauto

let T := { root := rootₙₑ, wellFormed := hNewRootIsAVLTree }

have hHeightBound := by
  simp only
  unfold height

  cases hBalR

  case inl hBalR =>
    have hHeights := heightsBalanceFactor0 (by tauto) hBalR
    have hEq : height (Node (1 + AVLNode.max (height left) (height lᵣ)) kᵣₒ vᵣₒ left lᵣ) = height lᵣ + 1 := by
      have hIf : (if height lᵣ - 1 > height lᵣ then height lᵣ - 1 else height lᵣ) = height lᵣ := by
        split <;> linarith
      simp only [height, hHeights.2.2, AVLNode.max, gt_iff_lt]
      linarith
    rw [hCachedHeightLᵣ, hCachedHeightL, hEq, hHeights.2.1, hHᵣₒ, hHeights.2.2, AVLNode.max]
    simp only [gt_iff_lt, lt_add_iff_pos_right, zero_lt_one, ↓reduceIte]
    have : 1 + (height lᵣ + 1) = height lᵣ - 1 + 2 + 1 := by linarith
    tauto

  case inr hBalR =>
    have hHeights := heightsBalanceFactor1 (by tauto) hBalR
    have hEq : height (Node (1 + AVLNode.max (height left) (height lᵣ)) kᵣₒ vᵣₒ left lᵣ) = height lᵣ + 1 := by
      have hIf : (if height lᵣ < height lᵣ then height lᵣ else height lᵣ) = height lᵣ := by split <;> linarith
      simp only [height, hHeights.2.2, AVLNode.max, gt_iff_lt]
      linarith
    rw [hCachedHeightLᵣ, hCachedHeightL, hEq, hHeights.2.1, hHᵣₒ, hHeights.2.2, AVLNode.max]
    simp only [gt_iff_lt, lt_self_iff_false, ↓reduceIte]
    have : 1 + (height lᵣ + 1) = height lᵣ + 2 := by linarith
    tauto

have lessThanProof (k : α) (hKey : cmp.cmp kᵢ k = Order.LESS) (hTree : allLess k (Node hᵣₒ kᵣₒ vᵣₒ left rᵣₒ))
  : allLess k T.root := by
  have hL := hRebalance.allLess k hKey hTree
  unfold allLess at hL hTree
  simp only [allLess]
  tauto

have greaterThanProof (k : α) (hKey : cmp.cmp kᵢ k = Order.GREATER)
  (hTree : allGreater k (Node hᵣₒ kᵣₒ vᵣₒ left rᵣₒ)) : allGreater k T.root := by
  have hR := hRebalance.allGreater k hKey hTree
  unfold allGreater at hR hTree
```

```
  simp only [allGreater]
  tauto

have hMapsTo : maps T.root kᵢ vᵢ := by
  simp only [maps]
  simp only [maps] at hMapsR
  cases hMapsR
  case inl => tauto
  case inr hMapsR =>
    cases hMapsR
    case inl h => exact Or.inr (Or.inl (And.intro h.1 (Or.inr (Or.inr (And.intro hCmpKᵢ h.2)))))
    case inr h => exact Or.inr (Or.inr h)

{ T := T , heightBound := hHeightBound, lessThanProof := lessThanProof, greaterThanProof := greaterThanProof,
  isNodeProof := by tauto, mapsTo := hMapsTo }

def rebalanceRight {α β : Type} {cmp : ComparisonFunction α} {rₒ}
  (hₒ : ℤ) (kₒ : α) (vₒ : β)
  (left : AVLNode α β cmp) (right : AVLNode α β cmp)
    {hWellFormed}
  (kᵢ : α) (vᵢ : β)
  (hWellFormedL : left.isAVLTree) (hWellFormedR : right.isAVLTree)
  (hBal : right.height = left.height + 2)
  (hHₒ : hₒ = left.height + 2)
  (hRNode : right.isNode)
  (hAllLessL : allLess kₒ left)
  (hAllGreaterR : allGreater kₒ right)
  (hRebalance : RebalanceHypotheses hₒ kₒ vₒ left rₒ kᵢ right)
  (hCmpKᵢ : cmp.cmp kᵢ kₒ = Order.GREATER)
  (hMapsR : right.maps kᵢ vᵢ)

: (@AVLTreeInsertion α β cmp) kᵢ vᵢ { root := Node hₒ kₒ vₒ left rₒ, wellFormed := hWellFormed } hₒ :=

  match hEqR : right with
  | (Node hᵣ kᵣ vᵣ lᵣ rᵣ) =>

  if hRightIsLeftHeavy : lᵣ.height > rᵣ.height then
    -- right left rotation

    -- linter claims this is unused, but needed for next match
    have hLRNode : lᵣ.isNode := by
      by_contra hRightLeftNodeNode
      have hLRLeaf : lᵣ = Leaf := notNodeToLeaf lᵣ hRightLeftNodeNode
      have hHeightRRNonNegative := heightIsNonNegative rᵣ
      rw [hLRLeaf, height] at hRightIsLeftHeavy
      linarith

    match hEqLR : lᵣ with
    | (Node hᵣₗ kᵣₗ vᵣₗ lᵣₗ rᵣₗ) =>

    let newLₗ := left
    let newLᵣ := lᵣₗ
    let newRₗ := rᵣₗ
    let newRᵣ := rᵣ

    let n := left.height

    have hBalR : balanceFactor right = -1 := by
      have hOrigBalR := extractIsBalanced hWellFormedR
      rw [isBalanced, decide_eq_true_eq, ←hEqR] at hOrigBalR
      cases hOrigBalR.1

      case inl h => exact h
      case inr hOrigBalLNode =>
        cases hOrigBalLNode

        case inl hOrigBalRNode =>
          simp only [hEqR, balanceFactor, ←hEqLR] at hOrigBalRNode
          rw [←hEqLR] at hRightIsLeftHeavy
```

```
      linarith

    case inr hOrigBalRNode =>
      simp only [hEqR, balanceFactor, ←hEqLR] at hOrigBalRNode
      rw [←hEqLR] at hRightIsLeftHeavy
      linarith

have hHeightRᵣ : height rᵣ = height lᵣ - 1 := by
  simp only [hEqR, balanceFactor, ←hEqLR] at hBalR
  linarith

have hHeightLᵣ : height lᵣ = height right - 1 := by
  have hHeightEq : height right = height lᵣ + 1 := by
    rw [←hEqLR] at hEqR
    simp only [hEqR, height]
    unfold AVLNode.max
    have hIf : (if height lᵣ > height rᵣ then height lᵣ else height rᵣ) = height lᵣ := by split <;> linarith
    rw [hIf]
    linarith
  linarith

have hHeightNewRᵣ : newRᵣ.height = n := by
  rw [←hEqR] at hBal
  simp only [newRᵣ, hHeightRᵣ, hHeightLᵣ, hBal]
  linarith

have hHeightNewRₗ : newRₗ.height = n-1 ∨ newRₗ.height = n := by
  have hHeight : height lᵣ = n+1 := by linarith
  have hBal : lᵣ.isBalanced := by
    simp only [isAVLTree, isBalanced, decide_eq_true_eq] at hWellFormedR
    simp only [isBalanced, decide_eq_true_eq, hEqLR]
    tauto
  cases childHeightRight hHeight hEqLR hBal
  case inl h =>
    have : height rᵣₗ = n - 1 := by linarith
    tauto
  case inr h =>
    simp only [add_sub_cancel] at h
    tauto

have hHeightNewLᵣ : newLᵣ.height = n-1 ∨ newLᵣ.height = n := by
  have hHeight : height lᵣ = n+1 := by linarith
  have hBal : lᵣ.isBalanced := by
    simp only [isAVLTree, isBalanced, decide_eq_true_eq] at hWellFormedR
    simp only [isBalanced, decide_eq_true_eq, hEqLR]
    tauto
  cases childHeightLeft hHeight hEqLR hBal
  case inl h =>
    have : height lᵣᵣ = n - 1 := by linarith
    tauto
  case inr h =>
    simp only [add_sub_cancel] at h
    tauto

have hHeightLₗ : newLₗ.height = n := by tauto

let lₙₑ := Node (1 + AVLNode.max (cachedHeight newLₗ) (cachedHeight newLᵣ)) kᵣₒ vᵣₒ newLₗ newLᵣ
let rₙₑ := Node (1 + AVLNode.max (cachedHeight newRₗ) (cachedHeight newRᵣ)) kᵣ vᵣ newRₗ newRᵣ
let rootₙₑ := Node (1 + AVLNode.max (cachedHeight lₙₑ) (cachedHeight rₙₑ)) kᵣₗ vᵣₗ lₙₑ rₙₑ

have hCachedHeightLₙₑ : cachedHeight lₙₑ = lₙₑ.height := by
  have hNewLLCachedHeight := cachedHeightIsCorrect newLₗ hWellFormedL
  have hNewLRCachedHeight := cachedHeightIsCorrect newLᵣ (leftIsAVLTree (leftIsAVLTree hWellFormedR))
  have h1 : cachedHeight lₙₑ = 1 + AVLNode.max (cachedHeight newLₗ) (cachedHeight newLᵣ) := by tauto
  have h2 : height lₙₑ = 1 + AVLNode.max (cachedHeight newLₗ) (cachedHeight newLᵣ) := by
    rw [height, hNewLLCachedHeight, hNewLRCachedHeight]
  rw [h1, h2]
have hCachedHeightRₙₑ : cachedHeight rₙₑ = rₙₑ.height := by
  have hNewRLCachedHeight := cachedHeightIsCorrect newRₗ (rightIsAVLTree (leftIsAVLTree hWellFormedR))
```

```
  have hNewRRCachedHeight := cachedHeightIsCorrect newRᵣ (rightIsAVLTree hWellFormedR)
  have h1 : cachedHeight rₙₑ = 1 + AVLNode.max (cachedHeight newRₗ) (cachedHeight newRᵣ) := by tauto
  have h2 : height rₙₑ = 1 + AVLNode.max (cachedHeight newRₗ) (cachedHeight newRᵣ) := by
    rw [height, hNewRLCachedHeight, hNewRRCachedHeight]
  rw [h1, h2]

have hNewRootIsAVLTree : rootₙₑ.isAVLTree := by
  unfold isAVLTree
  apply And.intro

  . have hOrigOrdered := extractIsOrdered hWellFormedR
    unfold isOrdered at hOrigOrdered
    have hOrigOrderedRₗ := hOrigOrdered.2.1
    unfold isOrdered at hOrigOrderedRₗ
    unfold isOrdered
    apply And.intro
    apply And.intro

    . have hCmp : cmp.cmp kᵣₒ kᵣₗ = Order.LESS := by
        simp only [allGreater] at hAllGreaterR
        exact cmp.cmpGreaterToLess hAllGreaterR.2.1.1
      simp only [allLess, newLᵣ, newLₗ]
      apply And.intro

      . exact hCmp

      apply And.intro

      . exact lessAllLess hCmp (by tauto)
      . exact hOrigOrderedRₗ.1.1

    . simp only [allGreater, newRᵣ, newRₗ]
      have hCmp : cmp.cmp kᵣ kᵣₗ = Order.GREATER := by
        have hOrderingRₗ := hOrigOrdered.1.1
        unfold allLess at hOrderingRₗ
        exact cmp.cmpLessToGreater hOrderingRₗ.1
      apply And.intro

      . exact hCmp

      apply And.intro

      . exact hOrigOrderedRₗ.1.2
      . exact greaterAllGreater hCmp hOrigOrdered.1.2

    apply And.intro

    . simp only [isOrdered, newLᵣ, newLₗ]
      apply And.intro
      apply And.intro

      . exact hAllLessL
      . simp only [allGreater] at hAllGreaterR
        exact hAllGreaterR.2.1.2.1

      apply And.intro

      . unfold isAVLTree at hWellFormedL
        exact hWellFormedL.1
      . exact hOrigOrderedRₗ.2.1

    . simp only [isOrdered, newRᵣ, newRₗ]
      apply And.intro
      apply And.intro

      . unfold allLess at hOrigOrdered
        exact hOrigOrdered.1.1.2.2
      . exact hOrigOrdered.1.2
```

```
    apply And.intro

    . exact hOrigOrderedRₗ.2.2
    . exact hOrigOrdered.2.2

apply And.intro

. have hOrigHeight := extractHeightCorrect hWellFormedR
  unfold heightCorrect at hOrigHeight
  unfold heightCorrect
  apply And.intro

  . rw [hCachedHeightLₙₑ, hCachedHeightRₙₑ]
    tauto

  apply And.intro

  . simp only [heightCorrect, newRᵣ, newRₗ]
    apply And.intro

    . tauto

    apply And.intro

    . unfold isAVLTree at hWellFormedL
      exact hWellFormedL.2.1
    . unfold heightCorrect at hOrigHeight
      exact hOrigHeight.2.1.2.1

  . simp only [heightCorrect, newLᵣ, newLₗ]
    apply And.intro

    . tauto

    apply And.intro

    . unfold heightCorrect at hOrigHeight
      exact hOrigHeight.2.1.2.2
    . exact hOrigHeight.2.2

. simp only [isBalanced, decide_eq_true_eq, balanceFactor, height, AVLNode.max]
  apply And.intro

  . cases hHeightNewLᵣ
    case inl hHeightNewLᵣ =>
      cases hHeightNewRₗ
      case inl hHeightRₗ =>
        have hIf : (if n - 1 > n then n - 1 else n) = n := by
          simp only [gt_iff_lt, ite_eq_right_iff, sub_eq_self, one_ne_zero, imp_false, not_lt,
            tsub_le_iff_right, le_add_iff_nonneg_right, zero_le_one]
        have hIf' : (if n > n - 1 then n else n - 1) = n := by
          simp only [gt_iff_lt, sub_lt_self_iff, zero_lt_one, ↓reduceIte]
        have : 1 + n - (1 + n) = 0 := by linarith
        simp only [hHeightNewRᵣ, hHeightNewLᵣ, hHeightRₗ, hHeightRₗ, hIf, hIf']
        tauto
      case inr hHeightRₗ =>
        simp only [hHeightNewRᵣ, hHeightNewLᵣ, hHeightRₗ, hHeightRₗ, gt_iff_lt, lt_self_iff_false,
          ↓reduceIte, sub_lt_self_iff, zero_lt_one, sub_self, Int.reduceNeg, zero_eq_neg, one_ne_zero,
          zero_ne_one, or_false, or_true]
    case inr hHeightNewLᵣ =>
      cases hHeightNewRₗ
      case inl hHeightRₗ =>
        have hIf : (if n - 1 > n then n - 1 else n) = n := by
          simp only [gt_iff_lt, ite_eq_right_iff, sub_eq_self, one_ne_zero, imp_false, not_lt,
            tsub_le_iff_right, le_add_iff_nonneg_right, zero_le_one]
        have hIf' : (if n > n then n else n) = n := by simp only [gt_iff_lt, lt_self_iff_false, ↓reduceIte]
        have : 1 + n - (1 + n) = 0 := by linarith
        simp only [hHeightNewRᵣ, hHeightNewLᵣ, hHeightRₗ, hHeightRₗ, hIf, hIf']
        tauto
```

```
        case inr hHeightRₗ =>
          simp only [hHeightNewRᵣ, hHeightNewLᵣ, hHeightRₗ, hHeightRₗ, gt_iff_lt, lt_self_iff_false,
            ↓reduceIte, sub_self, Int.reduceNeg, zero_eq_neg, one_ne_zero, zero_ne_one, or_false, or_true]
    apply And.intro

    . apply And.intro

      . cases hHeightNewLᵣ
        case inl hHeightNewLᵣ =>
          simp only [hHeightNewRᵣ, hHeightNewLᵣ, sub_sub_cancel_left, Int.reduceNeg, neg_eq_zero,
            one_ne_zero, neg_eq_self_iff, or_self, or_false]
        case inr hHeightNewLᵣ =>
          simp only [hHeightNewRᵣ, hHeightNewLᵣ, sub_self, Int.reduceNeg, zero_eq_neg, one_ne_zero,
            zero_ne_one, or_false, or_true]

      . simp only [isAVLTree, isBalanced, decide_eq_true_eq] at hWellFormedR hWellFormedL
        tauto

    . apply And.intro

      . cases hHeightNewRₗ
        case inl hHeightRₗ =>
          simp only [hHeightNewRᵣ, hHeightRₗ, sub_sub_cancel, Int.reduceNeg, eq_neg_self_iff, one_ne_zero,
            or_true]
        case inr hHeightRₗ =>
          simp only [hHeightNewRᵣ, hHeightRₗ, sub_self, Int.reduceNeg, zero_eq_neg, one_ne_zero, zero_ne_one,
            or_false, or_true]

      . simp only [isAVLTree, isBalanced, decide_eq_true_eq] at hWellFormedR
        tauto
let T := { root := rootₙₑ, wellFormed := hNewRootIsAVLTree }

have hHeightBound := by
  have : height T.root = hᵣₒ := by
    have hNewRight : height rₙₑ = n + 1 := by
      cases hHeightNewRₗ
      case inl hHeightNewRₗ =>
        rw [height, hHeightNewRₗ, hHeightNewRᵣ, AVLNode.max]
        have hIf : (if n - 1 > n then n - 1 else n) = n := by split <;> linarith
        rw [hIf]
        linarith
      case inr hHeightNewRₗ =>
        rw [height, hHeightNewRₗ, hHeightNewRᵣ, AVLNode.max]
        simp only [gt_iff_lt, lt_self_iff_false, ↓reduceIte]
        linarith
    have hLₙₑ : height lₙₑ = n + 1 := by
      cases hHeightNewLᵣ
      case inl hHeightNewLᵣ =>
        have hIf : (if n > n - 1 then n else n - 1) = n := by split <;> linarith
        rw [height, hHeightNewLᵣ, hHeightLₗ, AVLNode.max, hIf]
        linarith
      case inr hHeightNewLᵣ =>
        rw [height, hHeightNewLᵣ, hHeightLₗ, AVLNode.max]
        simp only [gt_iff_lt, lt_self_iff_false, ↓reduceIte]
        linarith
    have hOldRoot : hᵣₒ = n + 2 := by linarith
    rw [height, hNewRight, hLₙₑ, hOldRoot, AVLNode.max]
    simp only [gt_iff_lt, lt_self_iff_false, ↓reduceIte]
    linarith
  tauto

have lessThanProof (k : α) (hKey : cmp.cmp kᵢ k = Order.LESS) (hTree : allLess k (Node hᵣₒ kᵣₒ vᵣₒ left rᵣₒ))
  : allLess k T.root := by
  have hKᵣₒ : cmp.cmp kᵣₒ k = Order.LESS := by
    unfold allLess at hTree
    exact hTree.1
  have hAllLessR := hRebalance.allLess k hKey hTree
  simp only [allLess] at hAllLessR
  simp only [allLess]
```

```
    apply And.intro

    . exact hAllLessR.2.1.1

    apply And.intro

    . apply And.intro

      . exact hK_ro

      apply And.intro

      . unfold allLess at hTree
        exact hTree.2.1
      . exact hAllLessR.2.1.2.1

    . apply And.intro

      . exact hAllLessR.1

      apply And.intro

      . exact hAllLessR.2.1.2.2
      . exact hAllLessR.2.2



have greaterThanProof (k : α) (hKey : cmp.cmp k_i k = Order.GREATER)
  (hTree : allGreater k (Node h_ro k_ro v_ro left r_ro)) : allGreater k T.root := by
  have hK_ro : cmp.cmp k_ro k = Order.GREATER := by
    unfold allGreater at hTree
    exact hTree.1
  simp only [allGreater] at hAllGreaterR
  simp only [allGreater]
  apply And.intro

  . exact cmp.cmpTransitiveGreater hAllGreaterR.2.1.1 hK_ro

  apply And.intro

  . apply And.intro

    . exact hK_ro

    apply And.intro

    . unfold allGreater at hTree
      exact hTree.2.1
    . exact greaterAllGreater hK_ro hAllGreaterR.2.1.2.1

  . apply And.intro

    . exact cmp.cmpTransitiveGreater hAllGreaterR.1 hK_ro

    apply And.intro

    . exact greaterAllGreater hK_ro hAllGreaterR.2.1.2.2
    . exact greaterAllGreater hK_ro hAllGreaterR.2.2

have hMapsTo : maps T.root k_i v_i := by
  have hCmp : cmp.cmp k_r k_r' = Order.GREATER := by
    have hCmp' : cmp.cmp k_r' k_r = Order.LESS := by
      simp only [isAVLTree, isOrdered, allLess] at hWellFormedR
      tauto
    exact cmp.cmpLessToGreater hCmp'
  simp only [maps]
  simp only [maps] at hMapsR
  cases hMapsR
  case inl h =>
```

```
      exact Or.inr (Or.inr (And.intro (by rw [←(applyIff (cmp.cmpEq kᵢ kᵣ) h.1)] at hCmp; exact hCmp) (Or.inl h)))

    case inr hMapsR =>
      cases hMapsR
      case inl h =>
        cases h.2
        case inl h' => exact Or.inl h'
        case inr h' =>
          cases h'
          case inl h' => exact Or.inr (Or.inl (And.intro h'.1 (Or.inr (Or.inr (And.intro hCmpKᵢ h'.2)))))
          case inr h' => exact Or.inr (Or.inr (And.intro h'.1 (Or.inr (Or.inl (And.intro h.1 h'.2)))))
      case inr h => exact Or.inr (Or.inr (And.intro (cmp.cmpTransitiveGreater h.1 hCmp) (Or.inr (Or.inr h))))

  { T := T , heightBound := hHeightBound, lessThanProof := lessThanProof, greaterThanProof := greaterThanProof,
    isNodeProof := by tauto, mapsTo := hMapsTo }

  else
    -- single left rotation
    have rightIsLeftHeavy {h k v l r} (hEqR' : right = Node h k v l r) : ¬height l > height r := by
      simp only [hEqR, Node.injEq] at hEqR'
      rw [hEqR'.2.2.2.1, hEqR'.2.2.2.2] at hRightIsLeftHeavy
      exact hRightIsLeftHeavy
    leftRotation hᵣₒ kᵣₒ vᵣₒ left right kᵢ vᵢ hWellFormedL (by rw [←hEqR] at hWellFormedR; tauto)
      (by rw [←hEqR] at hBal; tauto) hHᵣₒ (by rw [←hEqR] at hRNode; tauto) hAllLessL
      (by rw [←hEqR] at hAllGreaterR; tauto) (by rw [←hEqR] at hRebalance; tauto)
      {rightIsLeftHeavy := rightIsLeftHeavy} hCmpKᵢ (by rw [←hEqR] at hMapsR; tauto)

def insertRightSubtree {α β : Type} {cmp : ComparisonFunction α} {hᵣ kᵣ vᵣ lᵣ rᵣ hWellFormedR sibling}
  (hᵣₒ : ℤ) (kᵣₒ : α) (vᵣₒ : β) (lᵣₒ rᵣₒ : AVLNode α β cmp)
  (hWellFormed : (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ).isAVLTree)
  (kᵢ : α) (vᵢ : β) (rₙₑ : AVLTreeInsertion kᵢ vᵢ { root := Node hᵣ kᵣ vᵣ lᵣ rᵣ, wellFormed := hWellFormedR } hᵣ)
  (hMatch : cmp.cmp kᵢ kᵣₒ = Order.GREATER ∧ lᵣₒ = sibling ∧ rᵣₒ = Node hᵣ kᵣ vᵣ lᵣ rᵣ)

: AVLTreeInsertion kᵢ vᵢ { root := Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ, wellFormed := hWellFormed } hᵣₒ :=

let rootₙₑ := Node (1 + AVLNode.max (cachedHeight lᵣₒ) (cachedHeight rₙₑ.T.root)) kᵣₒ vᵣₒ lᵣₒ rₙₑ.T.root

have hCachedHeightRₙₑ := cachedHeightIsCorrect rₙₑ.T.root rₙₑ.T.wellFormed
have hCachedHeightLᵣₒ := cachedHeightIsCorrect lᵣₒ (leftIsAVLTree hWellFormed)

if hBal : rootₙₑ.isBalanced then
  have hNewRootIsAVLTree : rootₙₑ.isAVLTree := by
    have hWellFormedRₙₑ := rₙₑ.T.wellFormed
    unfold isAVLTree at hWellFormedRₙₑ

    unfold isAVLTree
    apply And.intro

    . unfold isOrdered
      have hOrigOrdered := extractIsOrdered hWellFormed
      unfold isOrdered at hOrigOrdered
      apply And.intro
      apply And.intro

      . exact hOrigOrdered.1.1
      . have hTree : allGreater kᵣₒ (Node hᵣ kᵣ vᵣ lᵣ rᵣ) := by
          rw [←hMatch.2.2]
          tauto
        exact rₙₑ.greaterThanProof kᵣₒ hMatch.1 hTree

      apply And.intro

      . exact hOrigOrdered.2.1
      . exact hWellFormedRₙₑ.1

    apply And.intro

    . have hOrigHeight := extractHeightCorrect hWellFormed
      unfold heightCorrect at hOrigHeight
```

```
    rw [heightCorrect, hCachedHeightR$_{n_e}$, hCachedHeightL$_{r_o}$]
    tauto

  . exact hBal

let T := { root := root$_{n_e}$, wellFormed := hNewRootIsAVLTree }

have hHeightBound := by
  have hH$_r$ : height r$_{r_o}$ = h$_r$ := by
    have hOrigHeight := extractHeightCorrect hWellFormed
    unfold heightCorrect at hOrigHeight
    have hHeightR := hOrigHeight.2.2
    rw [hMatch.2.2, heightCorrect, ←hMatch.2.2] at hHeightR
    tauto

  have hH$_{r_o}$ : h$_{r_o}$ = 1 + AVLNode.max (height l$_{r_o}$) (height r$_{r_o}$) := by
    have hOrigHeight := extractHeightCorrect hWellFormed
    unfold heightCorrect at hOrigHeight
    tauto

  cases r$_{n_e}$.heightBound

  case inl hBound =>
    have hOrigHeight := extractHeightCorrect hWellFormed
    unfold heightCorrect at hOrigHeight
    have hHeightR$_{r_o}$ := hOrigHeight.2.2
    have hR$_{r_o}$ : r$_{r_o}$ = Node h$_r$ k$_r$ v$_r$ l$_r$ r$_r$ := by tauto
    rw [hMatch.2.2, heightCorrect] at hHeightR$_{r_o}$
    rw [height, hBound, hHeightR$_{r_o}$.1, ←hR$_{r_o}$]
    tauto

  case inr hBound =>
    if hOrdering : height r$_{n_e}$.T.root ≤ l$_{r_o}$.height then
      have hOldHeight : h$_{r_o}$ = l$_{r_o}$.height + 1 := by
        have hIf : (if height l$_{r_o}$ > height r$_{r_o}$ then height l$_{r_o}$ else height r$_{r_o}$) = height l$_{r_o}$ := by
          split <;> linarith
        rw [AVLNode.max, hIf] at hH$_{r_o}$
        linarith
      have hNewHeight : T.root.height = l$_{r_o}$.height + 1 := by
        have h : height T.root = 1 + AVLNode.max (height l$_{r_o}$) (height r$_{n_e}$.T.root) := by tauto
        have hIf : (if height l$_{r_o}$ > height r$_{n_e}$.T.root then height l$_{r_o}$ else height r$_{n_e}$.T.root) = height l$_{r_o}$ := by
          split <;> linarith
        rw [h, AVLNode.max, hIf]
        linarith
      rw [hOldHeight, hNewHeight]
      tauto
    else
      have hOldHeight : h$_{r_o}$ = r$_{r_o}$.height + 1 := by
        have hIf : (if height l$_{r_o}$ > height r$_{r_o}$ then height l$_{r_o}$ else height r$_{r_o}$) = height r$_{r_o}$ := by
          split <;> linarith
        rw [AVLNode.max, hIf] at hH$_{r_o}$
        linarith
      have hNewHeight : T.root.height = r$_{r_o}$.height + 2 := by
        rw [hH$_r$, height, hBound, ←hH$_r$, AVLNode.max]
        split <;> linarith
      rw [hOldHeight, hNewHeight]
      have _ : height r$_{r_o}$ + 2 = height r$_{r_o}$ + 1 + 1 := by linarith
      tauto

have lessThanProof (k : α) (hKey : cmp.cmp k$_i$ k = Order.LESS) (hTree : allLess k (Node h$_{r_o}$ k$_{r_o}$ v$_{r_o}$ l$_{r_o}$ r$_{r_o}$))
  : allLess k T.root := by
  simp only [allLess]
  unfold allLess at hTree
  apply And.intro

  . unfold allLess at hTree
    exact hTree.1

  apply And.intro
```

```
    . exact hTree.2.1
    . have hTree' : allLess k (Node hᵣ kᵣ vᵣ lᵣ rᵣ) := by
        rw [hMatch.2.2] at hTree
        exact hTree.2.2
      exact rₙₑ.lessThanProof k hKey hTree'

  have greaterThanProof (k : α) (hKey : cmp.cmp kᵢ k = Order.GREATER)
    (hTree : allGreater k (Node hᵣₒ kᵣₒ vᵣₒ lᵣₒ rᵣₒ)) : allGreater k T.root := by
    simp only [allGreater]
    unfold allGreater at hTree
    apply And.intro

    . unfold allGreater at hTree
      exact hTree.1

    apply And.intro

    . exact hTree.2.1
    . have hTree' : allGreater k (Node hᵣ kᵣ vᵣ lᵣ rᵣ) := by
        rw [hMatch.2.2] at hTree
        exact hTree.2.2
      exact rₙₑ.greaterThanProof k hKey hTree'

  have hMapsTo : maps T.root kᵢ vᵢ := by
    simp only [maps]
    exact Or.inr (Or.inr (And.intro hMatch.1 rₙₑ.mapsTo))

  { T := T , heightBound := hHeightBound, lessThanProof := lessThanProof, greaterThanProof := greaterThanProof,
    isNodeProof := by tauto, mapsTo := hMapsTo }
else
  have hWellFormedL : lᵣₒ.isAVLTree := leftIsAVLTree hWellFormed
  have hWellFormedR : rₙₑ.T.root.isAVLTree := rₙₑ.T.wellFormed
  have hBal : rₙₑ.T.root.height = lᵣₒ.height + 2 := by
    have hOrigBal := extractIsBalanced hWellFormed
    rw [isBalanced, decide_eq_true_eq] at hOrigBal
    unfold isAVLTree at hWellFormedL hWellFormedR

    have hHeightRₙₑ : rₙₑ.T.root.height = rᵣₒ.height + 1 := by
      have hHᵣ : height rᵣₒ = hᵣ := by
        have hOrigHeight := extractHeightCorrect hWellFormed
        rw [heightCorrect, hMatch.2.2, heightCorrect, ←hMatch.2.2] at hOrigHeight
        tauto

      cases rₙₑ.heightBound

      case inl hHeightRₙₑ =>
        have hBal' : isBalanced rootₙₑ = true := by
          simp only [isBalanced, balanceFactor, decide_eq_true_eq, hHeightRₙₑ]
          apply And.intro

          . simp only [balanceFactor, hHᵣ] at hOrigBal
            tauto

          . tauto
        contradiction
      case inr hHeightRₙₑ =>
        rw [hHeightRₙₑ, hHᵣ]

    have hBalOld : rᵣₒ.height = lᵣₒ.height + 1 := by
      cases hOrigBal.1

      case inl hBalOld' =>
        simp only [balanceFactor] at hBalOld'
        have hEq : height rᵣₒ = height lᵣₒ - 1 := by linarith
        rw [hEq] at hHeightRₙₑ
        have hBal' : isBalanced rootₙₑ = true := by
          simp only [isBalanced, decide_eq_true_eq, hHeightRₙₑ, balanceFactor]
          have _ : height lᵣₒ - 1 + 1 - height lᵣₒ = 0 := by linarith
```

```
        tauto
      contradiction


    case inr hBalOld' =>
      cases hBalOld'

      case inl hBalOld' =>
        simp only [balanceFactor] at hBalOld'
        have hEq : height r_ro = height l_ro := by linarith
        rw [hEq] at hHeightR_ne
        have hBal' : isBalanced root_ne = true := by
          simp only [isBalanced, decide_eq_true_eq, hHeightR_ne, balanceFactor]
          have _ : height l_ro + 1 - height l_ro = 1 := by linarith
          tauto
        contradiction

      case inr hBalOld' =>
        simp only [balanceFactor] at hBalOld'
        linarith

  rw [hHeightR_ne, hBalOld]
  linarith

have hAllLessL : allLess k_ro l_ro := by
  have hOrigOrdered := extractIsOrdered hWellFormed
  unfold isOrdered at hOrigOrdered
  exact hOrigOrdered.1.1

have hAllGreaterR : allGreater k_ro r_ne.T.root := by
  have hTree : allGreater k_ro (Node h_r k_r v_r l_r r_r) := by
    rw [←hMatch.2.2]
    have hOrigOrdered := extractIsOrdered hWellFormed
    unfold isOrdered at hOrigOrdered
    exact hOrigOrdered.1.2
  exact r_ne.greaterThanProof k_ro hMatch.1 hTree

have hRebalanceAllLess (k : α) (hKey : cmp.cmp k_i k = Order.LESS) (hOrig : allLess k (Node h_ro k_ro v_ro l_ro r_ro))

  : allLess k r_ne.T.root := by
  have hL : allLess k (Node h_r k_r v_r l_r r_r) := by
    rw [hMatch.2.2, allLess] at hOrig
    exact hOrig.2.2
  exact r_ne.lessThanProof k hKey hL
have hRebalanceAllGreater (k : α) (hKey : cmp.cmp k_i k = Order.GREATER)
  (hOrig : allGreater k (Node h_ro k_ro v_ro l_ro r_ro)) : allGreater k r_ne.T.root := by
  have hr : allGreater k (Node h_r k_r v_r l_r r_r) := by
    rw [hMatch.2.2, allGreater] at hOrig
    exact hOrig.2.2
  exact r_ne.greaterThanProof k hKey hr
have hRebalance := {allLess := hRebalanceAllLess, allGreater := hRebalanceAllGreater}

have hH_ro := by
  have hHeightRIncreased : r_ne.T.root.height = r_ro.height + 1 := by
    have hOrigHeight := extractHeightCorrect hWellFormed
    simp only [heightCorrect, hMatch.2.2] at hOrigHeight
    rw [←hMatch.2.2] at hOrigHeight
    cases r_ne.heightBound

    case inl hHeight =>
      have hBal' : root_ne.isBalanced := by
        have hH_r : h_r = r_ro.height := hOrigHeight.2.2.1
        have hOrigBal := extractIsBalanced hWellFormed
        simp only [isBalanced, decide_eq_true_eq, balanceFactor] at hOrigBal
        simp only [isBalanced, decide_eq_true_eq, balanceFactor, hHeight, hH_r]
        have _ := extractIsBalanced hWellFormedR
        tauto
      contradiction
    case inr hHeight => linarith
```

```
    have hOrigHeight := extractHeightCorrect hWellFormed
    simp only [heightCorrect, height] at hOrigHeight
    rw [hOrigHeight.1, AVLNode.max]
    split <;> linarith

  rebalanceRight h_ro k_ro v_ro l_ro r_ne.T.root k_i v_i hWellFormedL hWellFormedR hBal hH_ro r_ne.isNodeProof hAllLessL
    hAllGreaterR hRebalance hMatch.1 r_ne.mapsTo

def insertNode {α β : Type} {cmp : ComparisonFunction α}
  (h_ro : ℤ) (k_ro : α) (v_ro : β) (l_ro r_ro : AVLNode α β cmp)
  (hWellFormed : (Node h_ro k_ro v_ro l_ro r_ro).isAVLTree)
  (k_i : α) (v_i : β)

  : AVLTreeInsertion k_i v_i { root := Node h_ro k_ro v_ro l_ro r_ro, wellFormed := hWellFormed } h_ro :=

    match hMatch : (cmp.cmp k_i k_ro, l_ro, r_ro) with

    -- key matches current node: replace data at node
    | (Order.EQUAL, _, _) => (
      let T := { root := Node h_ro k_ro v_i l_ro r_ro, wellFormed := by tauto }
      have hHeightBound := by
        have hHeight : T.root.height = h_ro := by
          have hOrigHeight := extractHeightCorrect hWellFormed
          unfold heightCorrect at hOrigHeight
          have hOrigHeight' := hOrigHeight.1
          simp only at hOrigHeight'
          rw [hOrigHeight', height]
          rfl
        tauto
      { T := T , heightBound := hHeightBound, lessThanProof := by tauto, greaterThanProof := by tauto,
        isNodeProof := by tauto, mapsTo := by simp only [Prod.mk.injEq] at hMatch; tauto }
    )

    -- insertion as left child
    | (Order.LESS, Leaf, _) =>
        insertLeftChild h_ro k_ro v_ro l_ro r_ro hWellFormed k_i v_i (by simp only [Prod.mk.injEq] at hMatch; tauto)

    -- symmetric to previous case
    | (Order.GREATER, _, Leaf) =>
        insertRightChild h_ro k_ro v_ro l_ro r_ro hWellFormed k_i v_i (by simp only [Prod.mk.injEq] at hMatch; tauto)

    -- insertion into left subtree
    | (Order.LESS, Node hₗ kₗ vₗ lₗ rₗ, sibling) => (
      have hWellFormedL : (Node hₗ kₗ vₗ lₗ rₗ).isAVLTree := by
        simp only [Prod.mk.injEq] at hMatch
        have hLEq := hMatch.2.1
        have hWellFormedL' := leftIsAVLTree hWellFormed
        rw [hLEq] at hWellFormedL'
        exact hWellFormedL'

      -- proves termination
      have hTerm : sizeOf lₗ + sizeOf rₗ < sizeOf l_ro + sizeOf r_ro := by
        simp only [Prod.mk.injEq] at hMatch
        rw [hMatch.2.1, hMatch.2.2]
        simp only [Node.sizeOf_spec, sizeOf_default, add_zero]
        linarith

      let l_ne := insertNode hₗ kₗ vₗ lₗ rₗ hWellFormedL k_i v_i

      insertLeftSubtree h_ro k_ro v_ro l_ro r_ro hWellFormed k_i v_i l_ne (by simp only [Prod.mk.injEq] at hMatch; tauto)
    )

    -- symmetric to previous case
    | (Order.GREATER, sibling, Node h_r k_r v_r l_r r_r) => (
      have hWellFormedR : (Node h_r k_r v_r l_r r_r).isAVLTree := by
        simp only [Prod.mk.injEq] at hMatch
        have hREq := hMatch.2.2
        have hWellFormedR' := rightIsAVLTree hWellFormed
        rw [hREq] at hWellFormedR'
```

```
        exact hWellFormedR'

      -- proves termination
      have hTerm : sizeOf l_r + sizeOf r_r < sizeOf l_ro + sizeOf r_ro := by
        simp only [Prod.mk.injEq] at hMatch
        rw [hMatch.2.1, hMatch.2.2]
        simp only [Node.sizeOf_spec, sizeOf_default, add_zero]
        linarith

      let r_ne := insertNode h_r k_r v_r l_r r_r hWellFormedR k_i v_i

      insertRightSubtree h_ro k_ro v_ro l_ro r_ro hWellFormed k_i v_i r_ne (by simp only [Prod.mk.injEq] at hMatch; tauto)
    )

  termination_by sizeOf l_ro + sizeOf r_ro

def empty (α β : Type) (cmp : ComparisonFunction α) : AVLTree α β cmp :=
  { root := Leaf, wellFormed := leafIsAVLTree }

def singleton {α β : Type} (key : α) (value : β) (cmp : ComparisonFunction α) : AVLTree α β cmp :=
  { root := (@Node α β cmp) 1 key value Leaf Leaf, wellFormed := singletonIsAVLTree key value }

def insert {α β : Type} {cmp : ComparisonFunction α} (T : AVLTree α β cmp) (key : α) (value : β) : AVLTree α β cmp
  := match T with
    | ⟨Leaf, _⟩ => singleton key value cmp
    | ⟨Node h_ro k_ro v_ro l_ro r_ro, wellFormed⟩ => (insertNode h_ro k_ro v_ro l_ro r_ro wellFormed key value).T

def lookupNode {α β : Type} {cmp : ComparisonFunction α} (T : AVLNode α β cmp) (key : α) : Option β
  := match T with
  | Leaf => none
  | (Node _ k v l r) => match cmp.cmp key k with
    | Order.LESS => lookupNode l key
    | Order.EQUAL => v
    | Order.GREATER => lookupNode r key

def lookup {α β : Type} {cmp : ComparisonFunction α} (T : AVLTree α β cmp) (key : α) : Option β
  := lookupNode T.root key

lemma xorContradiction {α : Type} {cmp : ComparisonFunction α} {k₁ k₂ : α} {cmp₁ cmp₂ : Order}
  (hCmp₁ : cmp.cmp k₁ k₂ = cmp₁) (hCmp₂ : cmp.cmp k₁ k₂ = cmp₂) (hCmpNeq : cmp₁ ≠ cmp₂) : ⊥ := by
  cases cmp.cmpXor k₁ k₂

  case inl =>
    cases cmp₁
    . cases cmp₂ <;> tauto
    . cases cmp₂ <;> tauto
    . cases cmp₂ <;> tauto
  case inr h =>
    cases h
    . cases cmp₁
      . cases cmp₂ <;> tauto
      . cases cmp₂ <;> tauto
      . cases cmp₂ <;> tauto
    . cases cmp₁
      . cases cmp₂ <;> tauto
      . cases cmp₂ <;> tauto
      . cases cmp₂ <;> tauto

lemma mapsToLookupNode {α β : Type} {cmp : ComparisonFunction α} {k : α} {v : β} (T : AVLNode α β cmp)
  (hWellFormed : T.isAVLTree) (hMapsTo : T.maps k v) : (AVLTree.mk T hWellFormed).lookup k = some v := by
  simp only [lookup]
  induction T

  case Leaf =>
    rw [maps] at hMapsTo
    contradiction
  case Node h_ro k_ro v_ro l_ro r_ro left_ih right_ih =>
    rw [lookupNode]
    rw [maps] at hMapsTo
```

```
      split

      case h_1 hEq =>
        cases hMapsTo
        case inl hMapsTo =>
          have : ⊥ := xorContradiction hEq hMapsTo.1 (by tauto)
          contradiction
        case inr hMapsTo =>
          cases hMapsTo
          case inl hMapsTo => exact left_ih (leftIsAVLTree hWellFormed) hMapsTo.2
          case inr hMapsTo =>
            have : ⊥ := xorContradiction hEq hMapsTo.1 (by tauto)
            contradiction

      case h_2 hEq =>
        cases hMapsTo
        case inl hMapsTo => tauto
        case inr hMapsTo =>
          cases hMapsTo
          case inl hMapsTo =>
            have : ⊥ := xorContradiction hEq hMapsTo.1 (by tauto)
            contradiction
          case inr hMapsTo =>
            have : ⊥ := xorContradiction hEq hMapsTo.1 (by tauto)
            contradiction

      case h_3 hEq =>
        cases hMapsTo
        case inl hMapsTo =>
          have : ⊥ := xorContradiction hEq hMapsTo.1 (by tauto)
          contradiction
        case inr hMapsTo =>
          cases hMapsTo
          case inl hMapsTo =>
            have : ⊥ := xorContradiction hEq hMapsTo.1 (by tauto)
            contradiction
          case inr hMapsTo => exact right_ih (rightIsAVLTree hWellFormed) hMapsTo.2

  lemma mapsToLookup {α β : Type} {cmp : ComparisonFunction α} {k : α} {v : β} (T : AVLTree α β cmp)
    (hMapsTo : T.root.maps k v) : T.lookup k = some v := mapsToLookupNode T.root T.wellFormed hMapsTo

  lemma AVLTreeInsertionProducesValue {α β : Type} {cmp : ComparisonFunction α} (T : AVLTree α β cmp) (k : α) (v :
β)
    : (T.insert k v).lookup k = some v := by
    rw [insert]
    split

    case h_1 =>
      simp only [singleton, lookup, lookupNode]
      split
      case h_1 =>
        have _ := cmp.cmpK k
        tauto
      case h_2 => rfl
      case h_3 =>
        have _ := cmp.cmpK k
        tauto

    case h_2 hₙₒ kₙₒ vₙₒ lₙₒ rₙₒ hWellFormed =>
      let T' := (insertNode hₙₒ kₙₒ vₙₒ lₙₒ rₙₒ hWellFormed k v)
      exact mapsToLookup T'.T T'.mapsTo

  lemma AVLTreeInsertionDoesNotModifyOtherValues {α β : Type} {cmp : ComparisonFunction α}
    (T : AVLTree α β cmp) (k : α) (v : β) (k' : α) (result : Option β) (hk : k ≠ k')
    : T.lookup k' = result → (T.insert k v).lookup k' = result := by sorry

  lemma EmptyAVLTreeLookup {α β : Type} {cmp : ComparisonFunction α} (k : α) : (empty α β cmp).lookup k = none := by
    rw [lookup, empty, lookupNode]
```

```
end AVLTree
```