Vrije Universiteit Amsterdam

Bachelor Thesis

# Verifying AVL Trees in Lean

**Author:** Sofia Konovalova (2635220)

*1st supervisor:* Jasmin Blanchette
*daily supervisor:* Jannis Limperg
*2nd reader:* Alexander Bentkamp

*A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science*

July 19, 2021

## Abstract

Using interactive theorem provers allow for researchers and mathematicians to create reusable formalized libraries of mathematics. This thesis contributes to this effort by creating a formalization of AVL trees. The AVL tree definitions and lemmas are outlined, with explanations of design changes made to fit Lean. Comparisons are made with existing definitions of red-black trees in Lean and AVL trees in other interactive theorem provers. Work is set to continue on the formalization, with changes made to make the proofs more efficient and remove overhead to integrate into the Lean mathlib library.

## Contents

The full code presented this this thesis may be found at the following link: `https://github.com/reglayass/lean-thesis`

## 1 Introduction

In computer science, interactive theorem provers are used to develop formalizations of mathematical or logical concepts. The advantage of using these provers is that proofs can be long, or difficult to write by hand – a good example is Fermat's Last Theorem, the full proof of which is known to be thousands of pages long. Additionally, there is no guarantee of the correctness of the initial proof statement. With interactive theorem provers, every proof step is verified by the lnaguage, which gives confidence in the correctness of the proof itself and the proof statement.

Interactive theorem provers, such as Coq, Isabelle and Lean, are used to develop formalizations of mathematical and logical concepts. Lean, though it is younger than Coq or Isabelle, already boasts mathlib [1], a large library of formalized mathematics. Lean also improves on Coq by having a smaller kernel, cleaner syntax with Unicode support that can be written in Lean source files, and support for metaprogramming.

Though matlib, through its detail and extensiveness has become a defacto standard library in Lean, is missing something, contrary to its counterparts Coq and Isabelle – formalized tree structures. This thesis serves as a contribution to Lean and the mathlib library in the form of formalizing AVL trees, as well as a case study of the possibility of further formalization of search trees in Lean.

The paper begins by giving a brief introduction to the Lean theorem prover, continuing on to a brief introduction on binary search trees and AVL trees, providing the corresponding Lean types and definitions. Afterwards, the verification process is discussed, proving all the necessary proof statements and changes made along the way. Finally, the formalization is compared to those made in Coq and Isabelle, and the future of this work is outlined.

## 2 Lean Theorem Prover

The Lean Theorem Prover is a proof assistant based on dependent type theory with inductive families and universe polymorphism [2]. The language contains dependent function types and inductive types.

In simple type theory, every expression has an associated type – for example, integers, booleans or functions $\alpha \rightarrow \beta$ where $\alpha$ and $\beta$ are types. Lean's dependent type theory extends simple type theory by having types themselves be terms [3], which can be constants, variables, applications (i.e. functions) and $\lambda$-expressions.

```
constant x : ℤ
#check λx : ℤ, square (abs x)
```

The code above shows an example of usage of simple types. we have a constant `x` which is an integer and a $\lambda$-expression. A $\lambda$-expression `λx : t` with `t :` $\alpha$, is essentially a function `x → t` where each value of `x` is mapped to `t`. Therefore, taking the example above, the expression would map the value denoted by 0 to `square (abs 0)`, 1 to `square (abs 1)` and so on.

Simple type theory allows to differentiate between certain structures of different types. For example, if we have a `list` $\alpha$, we can differentiate between a list of booleans `list bool` or lists of integers `list ℤ`.

Dependent types depend on *terms* instead of types. If we have a function `pick n` where the function returns a natural number between 1 and `n`, intuitively the type becomes the set of all natural numbers up to and including `n`. The type of `pick` is *dependent* on the type of `n`. In this example, `n` is the term and the set of natural number is the is the type that depends on it.

In Lean, an inductive type is has zero or more constructors, and each constructor specifies a way of building an inhabitant of the type. Below are two examples of inductive types in Lean : `nat` and `list`.

```
inductive nat : Type
| zero : nat
| succ : nat → nat

inductive list (T : Type u)
| nil : list
| cons (hd : T) (tl : list) : list
```

We can see that `nat` has two constructors, `zero` and `succ`, that can be used to create new values of type `nat`, and doesn't have any constructor arguments and is overall a pretty simple inductive type. Terms can be created from this constructor fairly simply - for example, zero is `nat .zero`, one is `nat.succ (nat.zero)`, and so on. The inductive type for lists has two constructors as well, for an empty list and a list with a head and a tail. This inductive type is recursive, as the constructor argument `tl` refers to the inductive type itself [4].

## 3 AVL Trees and Operations

AVL trees are a type of binary search trees (BSTs) that are self-balancing - meaning, it must be ordered and balanced after every operation. Balance and order is assumed and preserved by lookup, insertion and deletion. In Lean, the tree is implemented with an inductive type, and the operations are implemented as functions.

### 3.1 Binary Search Trees

A binary search tree (BST) – also called an ordered tree – is a tree that is either empty or is a node with up to two children, a node key and a value. The children are referred to as the *left child* and *right child*.

In Lean, binary search trees are defined as the following inductive type.

```
inductive btree (α : Type u)
| empty {} : btree
| node (l : btree) (k : nat) (a : α) (r : btree) : btree
```

This definition has two constructors: one for an empty tree, and one for a node. The node key is defined as a natural number. There is no leaf constructor, as a leaf is a node with no children, which can be defined with the `node` constructor.

From there, I define a function to search for a key in a tree. This function, `bound`, verifies that a key exists in a tree; therefore, it doesn't matter in which subtree it is located, as long as it is present in one of them.

```
def bound (x : nat) : btree α → bool
| btree.empty := ff
| (btree.node l k a r) :=
  x = k ∨ bound l ∨ bound r
```

A binary search tree must have the *binary search property.*

**Definition 3.1** (Binary Search Property)**.** Given any node N in a binary search tree, all the keys in the left subtree of N are smaller than the key of N, and all keys in the right subtree are greater than the key of N.

In Lean we define the binary search tree property wih two separate definitions. The first one, `forall_keys`, describes the relationship between a key and a tree – for all the keys that exist in the tree, the type of relation on natural numbers `nat → nat → Prop` (in this case, $>$ or $<$) holds for the input key and the keys in the tree.

```
def forall_keys (p : nat → nat → Prop) (k : nat) (t : btree α) : Prop :=
  ∀ k', bound k' t → p k k'
```

The second definition, `ordered`, formalizes the binary search property. A tree is only ordered if the children are ordered, and `forall_keys` holds for the key being larger than the keys in the left subtree and the key being smaller than the keys in the right subtree.

```
def ordered : btree α → Prop
| btree.empty := tt
| (btree.node l k a r) :=
  ordered l ∧ ordered r ∧ (forall_keys (>) k l) ∧ (forall_keys (<) k r)
```

The lookup operation is done recursively. During traversal, it is possible to compare every key to another one assuming that the keys are totally ordered. When traversing the tree, if the input key is smaller than the current node key, we recurse into the left subtree; if the input key is larger, we recurse into the right subtree.

```
def lookup (x : nat) : btree α → option α
| btree.empty := none
| (btree.node l k a r) :=
  if x < k
    then lookup l
    else if x > k
      then lookup r
    else a
```

## 3.2 AVL Trees

An AVL tree [5] is a self-balancing binary search tree, where the absolute value of the height difference between two child subtrees is no more than one. This may also be described by the *balancing factor*.

**Definition 3.2** (Tree height)**.** The height of a node in a tree is the maximal number of edges from that node to a leaf.

**Definition 3.3** (Balancing factor)**.** The balancing factor of any node is defined to be the height difference of its two child subtrees.

In Lean, the definitions `height` and `balanced` are done per Definitions 3.2 and 3.3.

The easiest way to write the definition for `balanced` would be to write it in terms of absolute value. This would create a problem, as height is defined with a natural number and absolute values in Lean use real numbers, either coercion or casting would need to be used which would cause difficulties when writing proofs. The current definition compares the two children with each other to determine balance instead.

```
def height : btree α → nat
| btree.empty := 0
| (btree.node l k a r) :=
  1 + (max (height l) (height r))

def balanced : btree α → bool
| btree.empty := tt
| (btree.node l k a r) :=
  if height l ≥ height r
    then height l ≤ height r + 1
    else height r ≤ height l + 1
```

The process of looking up a node or searching for a key is the same as in BSTs.

During insertion and deletion, however, it is more complicated. In self-balancing trees, rebalancing actions are not performed arbitrarily, but after operations such as insertion and deletion that may change the structure of the tree. In the case of AVL trees, a tree may become left-heavy or right-heavy after these actions, after which the tree is re-balanced with rotations. Left-heaviness can be fixed with a simple right rotation; right-heaviness with a simple left rotation.

A tree `t` being left-heavy means that the left subtree's height is $n+2$ given height of the right subtree $n$; similarly, in a right-heavy tree, given the height $n$ of the left subtree, the height of the right subtree is $n+2$. The definitions presented further closely follow [6]. The definition for `right_heavy` is mirrored.

```
def left_heavy : btree α → bool
| btree.empty := ff
| (btree.node btree.empty k a r) := ff
| (btree.node (btree.node ll lk la lr) k a r) :=
  (height ll ≥ height lr) ∧ (height ll ≤ height lr + 1) ∧
  (height lr ≥ height r) ∧ (height r + 1 = height ll)
```

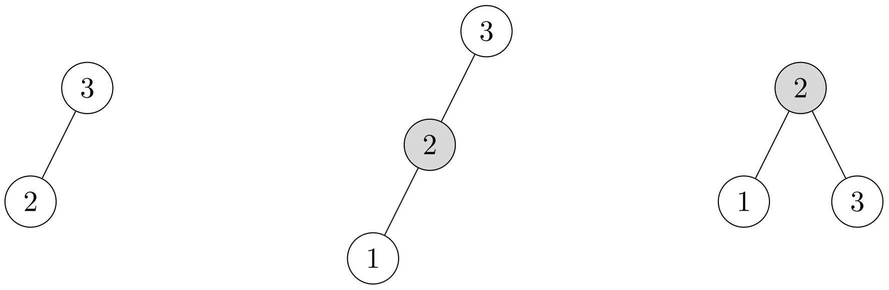

Figure 1

An example of a simple right rotation is shown in Figure 1. After the node with the key of 1 gets inserted, the tree becomes left-heavy, which can be fixed with a right rotation. The direct ancestor of the newly inserted node with the key of 2 becomes the new root of the tree, and the ancestor of the new root moves to the right. The result is a balanced tree, with order and the keys preserved.

```
def simple_right : btree α → btree α
| btree.empty := btree.empty
| (btree.node (btree.node ll lk la lr) k a r) :=
    btree.node ll lk la (btree.node lr k a r)
| (btree.node l k a r) := btree.node l k a r
```

Compound rotations are performed when after a simple rotation the tree is still unbalanced. If after a left rotation, the tree is becomes left-heavy then a right rotation is done to fix the heaviness. The definition for `rotate_left` is mirror.

```
def rotate_right : btree α → btree α
| btree.empty := btree.empty
| (btree.node l k a r) :=
  match l with
  | btree.empty := (btree.node l k a r)
  | (btree.node ll lk la lr) :=
    if height ll < height lr
      then simple_right (btree.node (simple_left l) k a r)
      else simple_right (btree.node l k a r)
  end
```

The insertion definition makes use of the definitions of heaviness and rotations to insert a node into a tree while retaining balance. If insertion creates a left-heavy tree, then a right rotation is done, and if it creates a right-heavy tree, then a left rotation is done.

```
def insert (x : nat) (v : α) : btree α → btree α
| btree.empty := btree.node btree.empty x v btree.empty
| (btree.node l k a r) :=
  if x < k
    then if left_heavy (btree.node (insert l) k a r)
      then rotate_right (btree.node (insert l) k a r)
      else btree.node (insert l) k a r
    else if x > k
      then if right_heavy (btree.node l k a (insert r))
        then rotate_left (btree.node l k a (insert r))
        else btree.node l k a (insert r)
    else btree.node l x v r
```

Deleting a node is more complicated, and depends on the its children. If its left child is empty, then the node can be removed and replaced by its right child. If the left subtree is not empty, we need to `shrink` it. The process of shrinking is simple – we travel the tree recursively along its right subtrees, looking for a node whose right child is empty. During this process, if the tree becomes imbalanced a right rotation is completed. Once the node is found, the `shrink` function returns its key, value, and the resulting shrunken tree. Since shrinking an empty tree is impossible, `shrink` returns an `option`.

```
def shrink : btree α → option (nat × α × btree α)
| btree.empty := none
| (btree.node l k v r) := some $
  match shrink r with
  | none := (k, v, l)
  | some (x, a, sh) :=
    if height l > height sh + 1
      then (x, a, rotate_right (btree.node l k v sh))
      else (x, a, btree.node l k v sh)
  end
```

After defining shriking, the next step is to define the function `del_node`. After the shrinking process is complete, a left rotation may be completed to re-balance the tree, and the key and node values of the tuple result of `shrink` replace the key and value of the node to delete.

```
def del_root : btree α → btree α
| btree.empty := btree.empty
| (btree.node l k v r) :=
  match shrink l with
  | none := r
  | some (x, a, sh) :=
    if height r > height sh + 1
      then rotate_left (node sh x a r)
      else node sh x a r
  end
```

Finally, the full `delete` function is complete.

```
def delete (x : nat) : btree α → btree α
| btree.empty := btree.empty
| (btree.node l k a r) :=
  let dl := delete l in
  let dr := delete r in
  if x = k
    then del_root (btree.node l k a r)
    else if x < k
      then if height r > height dl + 1
        then rotate_left (btree.node dl k a r)
        else btree.node dl k a r
    else if height l > height dr + 1
      then rotate_right (btree.node l k a dr)
    else (btree.node l k a dr)
```

First, find the node to delete. If it is found, then the `del_node` function is called on the entire subtree. During the search of the node to delete, the operation is called recursively, and rotations are applied if the tree becomes unbalanced.

## 4 Verification

In this section, we verify the operations presented above. Each subsction details the proof statements for specific operations and informally explains how to complete the proofs. The process begins with verifying correctness of the operations that do not make changes to the tree, like lookup and search, and then moves on to AVL tree rotations, finishing with insertion and deletion. For the latter two subsections, the proofs involve key and order preservation, and restoration of balance.

This section does not detail all of the proof constructions, but does give all the necessary lemmas and proofs are details when it is either important to understand the proof statement or when the proof construction led to core design decisions. Any auxiliary lemmas that were created during the process are outlined as well. The lemma statements presented closely follow [6], though some changes were made to statements to suit the Lean definitions.

### 4.1 BST Operations

First, we prove two lemmas related to boundedness and lookup in a tree, since `lookup` and `bound` are identical for both BSTs and AVL trees.

```
lemma bound_false (k : nat) (t : btree α) :
  bound k t = ff → lookup k t = none := ...

lemma bound_lookup (k : nat) (t : btree α) :
  ordered t → bound k t → ∃ (v : α), lookup k t = some v := ...
```

The lemma `bound_false` states that if a key is not bound in a tree, then lookup will not result in any node data being returned. The lemma `bound_lookup` states that if a key is bound in a tree, then some data will be returned. The existential quantifier is used in this lemma, because we cannot make assumptions on which key will return which value. In other words, we don't know the specific node value, but we do know that something will be returned. Both of the proofs were constructed by induction on the tree `t`.

### 4.2 Rotation

With rotations, we need to show that they restore balance, and preserve keys and order. This section will present these proofs based on a right rotation, and any core design changes made during the process of constructing these proofs. Left rotations are not mentioned in the following text, as the proofs for them mirror those for right rotations.

#### Order

All lemmas based on rotation assume that a tree `t` is ordered, and conclude that `t` remains ordered after a rotation. Since AVL trees are based on BSTs, and BSTs must be ordered, a rotation cannot violate this.

```
lemma rotate_right_ordered (t : btree α) :
  ordered t → ordered (rotate_right t) := ...
```

Since `rotate_right` and `rotate_left` are compound rotations, lemmas for simple rotations must be constructed too.

```
lemma simple_right_ordered (t : btree α) :
  ordered t → ordered (simple_right t) := ...

lemma simple_left_ordered (t : btree α) :
  ordered t → ordered (simple_left t) := ...
```

All of the above proofs were done by case splits on the tree `t`. Induction hypothesis are not needed since the rotation definitions are not recursive.

A lemma for the transitivity of `forall_keys` is written, and is applied in both the lemmas for compound rotations and simple rotations.

```
lemma forall_keys_trans (t : btree α) (p : nat → nat → Prop)
(z x : nat) (h₁ : p x z) (h₂ : ∀ a b c, p a b → p b c → p a c) :
  forall_keys p z t → forall_keys p x t := ...
```

During a rotation, the placement of a key may change but its relation to its ancestor and its children does not change. In this situation, `forall_keys_trans` is applied.

## Balance

We want show that balance restores rotation, therefore the assumption needs to be that a tree is either left or right imbalanced, and the corresponding rotation will make the tree balanced.

```
lemma rotate_right_balanced (t : btree α) :
  left_heavy t → balanced (rotate_right t) := ...
```

As with ordering, the corresponding lemmas for simple rotations need to be constructed.

```
lemma simple_right_balanced (t : btree α) :
  left_heavy t → balanced (simple_right t) := ...

lemma simple_left_balanced (t : btree α) :
  right_heavy t → balanced (simple_left t) := ...
```

Similarly to proofs about ordering, the above proofs were completed with case splitting on trees and subtrees. During construction of proofs with compound rotations, the lemmas about simple rotations were applied when needed.

The proofs about balance are a bit more complicated to solve. With the ordering lemmas, it's a case of finding the goals in the hypotheses, additionally using the transitivity lemma to achieve the goal. The balancing lemmas require more arithmetic, since the definitions of heaviness and `balanced` use tree height.

## Key Preservation

We first prove that rotations preserve keys.

```
lemma rotate_right_keys (t : btree α) (k : nat) :
  bound k t ↔ bound k (rotate_right t) := ...
```

As done with proofs for ordering, lemmas about simple rotations preserving keys are constructed as well.

```
lemma simple_right_keys (t : btree α) (k : nat) (x : bool) :
  bound k t = x ↔ bound k (simple_right t) = x := ...

lemma simple_left_keys (t : btree α) (k : nat) (x : bool) :
  bound k t = x ↔ bound k (simple_left t) = x := ...
```

In the lemma statements, we can see that instead of using `bound` as is, another parameter `x` is used to signify whether a key is bound or not. This was done so that these lemmas can be applied in situations where we want to prove that a key is not bound and we want to prove that a key is bound.

The proofs were written to be bi-implications. If we just take the left side into account – if a key is bound in a tree then it is still bound in a simple left rotation – there is no guarantee that after a simple left rotation, the keys all remain the same – the right side of the bi-implication.

It was during the construction of these proofs where the problem with the definition of `bound` was discovered. Previously, the definition for `bound` was similar to that of `lookup`, recursively searching the tree until the key was found.

This definition made it difficult to write the proofs needed. There would be an assumption that the key is bound in the same subtree after a rotation, which is not the case after a rotation. The definition was then changed to the one shown in Section 3.1, which made the proofs easier because then the definition did not take into account the placement of the key, just whether or not it exists. This fits better into the purpose of the proofs for rotation retaining keys – we want to make sure that some key is still present in the tree after a rotation, and not that the key is present in the same area of the tree.

## 4.3 Insertion

For insertion, we want to show that keys, order and balance are preserved.

### Key Preservation

There are three different lemmas in relation to key preservation.

```
lemma insert_bound (t : btree α) (k : nat) (v : α) :
  bound k (insert k v t) := ...

lemma insert_diff_bound (t : btree α) (k x : nat) (v : α) :
  bound x t → bound x (insert k v t) := ...

lemma insert_nbound (t : btree α) (k x : nat) (v : α) :
  (bound x t = ff ∧ x ≠ k) → bound x (insert k v t) = ff := ...
```

For `insert_bound`, the goal is to show that a key can be found as soon as it is inserted into a tree. For `insert_diff_bound` and `insert_nbound`, the goal is to show that the keys already existing in a tree do not get lost during insertion, and if a key does not exist in a tree, unless it is the new key, it does not exist after insertion. The proofs are completed by induction on the tree `t`.

Since `insert` uses rotations, previously constructed proofs about rotations preserving keys are applied in these proofs. The tactic `tauto` is used frequently, as the tactic is a decisional procedure for propositional logic and is able to split goals and assumptions that are disjunctions. Due to the bound definition using disjunction, this tactic does a lot of the heavy work of breaking down these forms, and then completing the separate goals based on reflexivity.

### Order

Lemma statements for insertion preserving order have the same structure as lemmas for rotations preserving order.

```
lemma insert_ordered (t : btree α) (k : nat) (v : α) :
  ordered t → ordered (insert k v t) := ...
```

This proof is constructed by induction on the tree `t` and previous lemmas for rotations preserving order are applied.

To complete the proof, an auxiliary proof `forall_insert` is written. In the process of completing `insert_ordered`, we need to show that previously existing keys in a tree preserve their relation with the tree after insertion. To bring an example from the proof – if a key `tk` is greater than all of the keys in the tree `tl` even after insertion, then it follows that `tk > k` and that `tk` is greater than all of the keys in `tl`. These are also the sub-goals that come out after applying the lemma.

```
lemma forall_insert (k x : nat) (t : btree α) (a : α)
(p : nat → nat → Prop) (h : p x k) :
  forall_keys p x t → forall_keys p x (insert k a t) := ...
```

Auxiliary proofs for rotation and `forall_keys` are also written since insertion uses rotations, and were completed with induction on the tree `t`.

```
lemma forall_rotate_right (x k : nat) (l r : btree α) (a : α)
(p : nat → nat → Prop) :
  forall_keys p x (btree.node l k a r) →
    forall_keys p x (rotate_right (btree.node l k a r)) := ...

lemma forall_simple_right (x k : nat) (l r : btree α) (a : α)
(p : nat → nat → Prop) :
  forall_keys p x (btree.node l k a r) →
    forall_keys p x (simple_right (btree.node l k a r)) := ...
```

### Balance

The lemma for insertion preserving balanced is formalized as such, and completed by induction on the tree `t`.

```
lemma insert_balanced (t : btree α) (k : nat) (v : α) :
  balanced t = tt → balanced (insert k v t) = tt :=
```

In the process of completing the proof, small changes were made to the `insert` definition. Previously, the definition follow [6] almost exactly. In that definition, the case splits would be made on the height differences. For example, to determine if a right rotation needs to be done after insertion, instead of using the definition `left_heavy`, the comparison that was done was `height (insert l) > height r` Since the proof for `insert_balanced` would be composed of the proofs for rotations preserving balance, applying these lemmas would result in either a `left_heavy` or `right_heavy` goal, but the hypotheses would all be expressions with `height`, which makes the proof more difficult to complete than what it could be. Therefore, in the definition, the cases that determine whether or not a rotation was done was changed from manual height comparisons to the `left_heavy` and `right_heavy` definitions. This meant that during the proof construction, there would be a case split on heaviness, and after applying a rotation lemma the goal would become trivial.

## 4.4 Deletion

This section details the steps taken to write a proof for deletion preserving order. Similar to proofs about rotations, if we want to prove deletion preserving order, any other definition that `delete` uses will have a lemma regarding order as well. Due to the way that `ordered` is defined with `forall_keys`, lemmas about boundedness are involved as well. Just from writing one lemma about deletion preserving order, we receive lemmas about key preservation as well, that can be used for other proofs. First we follow the steps taken to write the lemmas for order, then lemmas about key preservation are discussed specifically. Lastly, any design changes or additions made during this process are presented.

### Order

We begin with the lemma statement for deletion of a key and deletion of a root node preserving order.

```
lemma delete_ordered (t : btree α) (k : nat) :
  ordered t → ordered (delete k t) := ...

lemma del_root_ordered (t : btree α) :
  ordered t → ordered (del_root t) := ...
```

The proof for `delete_ordered` was constructed with induction on `t`, and `del_root` was completed with a case split on `t`.

The next step is to complete the proof for `shrink` preserving order. The proof for shrink needs to contain more information, as we cannot simply state that `t` is ordered and therefore `sh` is ordered, there needs to be a link between the two trees. Therefore, hypotheses need to contain the result of shrinking the tree, `shrink t = some (x, a, sh)`, and that the entire tree is ordered and therefore so is `sh`. The proof also concludes that `x` is larger than all of the keys in a shrunken tree.

```
lemma shrink_ordered {t sh : btree α} {x : nat} {a : α} :
  ordered t ∧ shrink t = some (x, a, sh) →
    ordered sh ∧ forall_keys gt x sh := ...
```

This lemma, could have been be split into two separate lemmas with one concluding that `sh` is ordered and that `x` is greater than all keys in `sh`. If the lemma was to be separated into two, the induction hypotheses would not be strong enough to complete the proofs.

The proof for `shrink_ordered` was done by induction on the tree generalizing `x`, `a` and `sh`.

In the inductive step, if the left subtree is larger than the height of `sh + 1`, then `sh = rotate_right (node l k v sh_1)`, where `sh_1` is from `shrink r = (x_1, a_1, sh_1)`. This case can be resolved with the lemma `rotate_right_keys` to show that keys are preserved after a rotation. Then we need to show that all the keys in sh come from the original tree, which is done by applying the lemma shrink_keys.

```
lemma shrink_keys {t sh : btree α} {x : nat} {a : α} :
  ordered t ∧ shrink t = some (x, a, sh) →
    bound x t ∧ (∀ k', bound k' t → bound k' sh) := ...
```

We know that x_1 is larger than all the keys in sh_1. Since (node l k v sh_1) is ordered, then it must be the case that k is smaller than all the keys in sh_1, and therefore x > k and x is greater than all the keys in l. This is formalized using the auxiliary lemma forall_shrink.

```
lemma forall_shrink {t sh : btree α} {k x : nat} {a : α}
{p : nat → nat → Prop} :
  forall_keys p k t ∧ shrink t = some (x, a, sh) →
    forall_keys p k sh ∧ p k x :=
```

As shrink_keys would have to be applied to shrink_ordered, the original forall_keys definitions had to be rewritten. The original definition was recursed into the two subtrees, as well as looking at the relation between the input key and current node key. It presented nothing about the key being bound in the tree, even though it is intuitive and a safe assumption to make. In order to use shrink_keys with forall_keys, we need a definition that includes an assumption of boundedness, but still compares the input key to keys in a tree. The result was the current definition of forall_keys.

After the new definition was written, a characterization lemma was needed in order to be able to work with the two different definitions, because we don't necessarily need the hypothesis that the keys compared are bound in the tree in all the proof constructions, or even in the entire proof construction.

The characterization lemma being a bi-implication allows for the lemma to be applied to any hypothesis containing a forall_keys, to extract information about the subtrees separately and the relationship between k and x. This made proof constructions with forall_keys significantly easier, as there was an alternative way to unfold forall_keys, instead of unfolding it in terms of bound like in the definition.

```
lemma forall_keys_unfolded {l r : btree α} {k x : nat} {v : α}
{p : nat → nat → Prop} :
  (forall_keys p k l ∧ p k x ∧ forall_keys p k r) ↔
    forall_keys p k (node l x v r) :=
```

## Views

During most proof constructions presented in this section, I simplified definitions to extract information, or create case splits. With shrink, this process would be long and would clutter the proof. There needed to be a way to split the result of shrink into the tree possible cases that can come out of shrinking a tree with one action, reducing any unnecessary writing. The same problem arose with del_node. To solve these problems we define two views for shrink[1] and del_node. The shrink_view is shown below.

```
inductive shrink_view {α} : btree α → option (nat × α × btree α) → Sort*
| empty : shrink_view empty none
```

[1]This approach was suggested by Jannis Limperg

```
| nonempty_empty : ∀ {l k v r},
  shrink r = none →
  shrink_view (node l k v r) (some (k, v, l))
| nonempty_nonempty₁ : ∀ {l k v r x a sh out},
  shrink r = some (x, a, sh) →
  height l > height sh + 1 →
  out = some (x, a, rotate_right (btree.node l k v sh)) →
  shrink_view (node l k v r) out
| nonempty_nonempty₂ : ∀ {l k v r x a sh},
  shrink r = some (x, a, sh) →
  height l ≤ height sh + 1 →
  shrink_view (node l k v r) (some (x, a, node l k v sh))
```

The views have a constructor for each possible result of `shrink` or `del_node`. In the case where rotations are made, an adjustment had to be made in the form of the assumption `out`. This was done because inductive types in Lean do not accept other function calls in the type constructor.

In order to use the views, auxiliary lemmas were written to apply a normal `shrink` or `del_node` and get the three case splits. The lemma for `shrink_view` is shown below. The proof for `del_node` is almost identical.

```
lemma shrink_shrink_view (t : btree α) :
  shrink_view t (shrink t) := ...
```

Applying the lemma would result in three cases in the inductive step of `shrink_ordered`, which matches the three cases from the definition – one for the right subtree being empty, leading to `shrink r = none`, another one where `shrink r = some (x, a, rotate_right(l k v sh))`, and another one where `shrink r = some (x, a, node l k v sh)`. This allowed to complete the proofs without creating additional case splits, cluttering the proof construction.

## 5 Related Work

Formalizations of AVL trees, as well as other search trees, are present in both the Coq [7] and Isabelle/HOL [8] interactive theorem provers. Lean does not have an AVL tree formalization yet, and the only other tree present int he mathlib library is a red-black tree.

### Coq

In contrast to my approach, the Coq formalization has one single balance function, `bal`, and no abstraction into separate left/right rotations. This can be seen as a stylistic choice, but it creates a very long definition and longer, more complicated proofs. While in my approach, the proofs are still relatively long, because lemmas are written for separate rotations, they can be applied to the long proof. It can have an effect on performance. The `bal` function is used in insertion, which is recursive. My approach to the `insert` function is to first determine whether it will result in a left- or right-heavy tree, and then rotate the tree after insertion, which saves operational time on not balancing a tree when it is not necessary.

The definition of binary search trees in Coq and Lean are similar, the only difference is that height is included in the definition of trees in Coq, while in Lean height is calculated when needed. While this can be a stylistic choice, it may have an effect on performance. With every new node added, one is added to the height of each ancestor, while with the Lean interpretation height is calculated on demand, which takes longer the bigger the tree is.

### Isabelle

In the Isabelle formalization, are two functions `balL` that re-balances left subtrees and `balR` that re-balances right subtrees. They function similarly to `rotate_right` and `rotate_left`, and are used in the balancing function, which makes proofs more modular. Like in Coq, the height of the tree is included in the tree definitions.

### Other Search Trees

Isabelle/HOL has a libary of data structures, which apart from containing an AVL tree formalization, has other tree formalizations like red-black trees, 2-3 trees, and standard binary trees. In Coq, finite sets are implemented using AVL trees and red-black trees, which so far are the only trees present in the standard library. A generic binary search tree structure is also present that is used by both AVL and red-black tree definitions.

Lean itself has an inductive datatype `bin_tree`, but no operations related to it. The mathlib library [1], a standard library in Lean, has an inductive type `tree` which is similar to `btree`. Both of the definitions do not contain a key value in the type constructors, so as is, they cannot be used to create search trees, or create maps or sets. Red-black trees have definitions for the inductive type `rbnode` and some definitions, but it is not completely formalized.

# 6 Conclusion

Theorem provers such as Lean are used to create libraries of mathematics and logic, to be used for research and teaching. In this thesis, I presented a formalization of AVL search trees using Lean. Lean proof constructions a relatively simple and mechanical task, once the main hurdle of learning and getting used to the syntax. Overall, the proof statements presented in Lean were not very different to those presented informally in [6], although some very small changes did have to be made. Additionally, definitions regarding the structure of AVL treees such as `balance` and `height` stay true to their mathematical definitions for trees. Hopefully, in the future, formalization of search trees as a form of contribution to mathematical libraries in Lean can continue and other abstract data structures may be formalized as well.

## Further Work

The continuation of this work would begin by completing proofs for insertion and deletion that are missing from the source code, for which the only cause is a lack of time. The next steps are to make use of Lean's automation and metaprogramming to create shorter and cleaner code, as well as improving definitions to remove overhead. The final step is the inclusion of this final formalization into the mathlib library.

## References

[1] T. mathlib Community, "The lean mathematical library," *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2020. DOI: `10.1145/3372885.3373824`. [Online]. Available: `http://dx.doi.org/10.1145/3372885.3373824`.

[2] P. Dybjer, "Inductive families," *Formal Aspects of Computing*, vol. 6, pp. 440–465, 1994.

[3] J. Avigad, L. de Moura, and S. Kong, *Theorem Proving in Lean*, version 3.23.0.

[4] J. Avigad, G. Ebner, and S. Ullrich, *The Lean Reference Manual*, version 3.3.0.

[5] G. Adelson-Velsky and E. Landis, "An algorithm for the organization of information," in *Soviet Mathematics - Doklady*, vol. 3, 1962, pp. 1259–1263.

[6] J. O'Donnell, C. Hall, and R. Page, "Discrete mathematics with a computer," in Springer, 2006, ch. 12, pp. 313–352, ISBN: 1-84628-241-1.

[7] "Library Coq.MSets.MSetAVL." (2021), [Online]. Available: `https://coq.inria.fr/library/Coq.MSets.MSetAVL.html`.

[8] D. Stüwe and T. Nipkow. "Theory AVL_Set_Code." (2021), [Online]. Available: `https://isabelle.in.tum.de/library/HOL/HOL-Data_Structures/AVL_Set_Code.html`.