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 20, 2021

Acknowledgements

I would like to begin my thanking my daily supervisor, Jannis Limperg. He was incredibly helpful, patient and kind to me throughout this process, and it helped me build my confidence in a topic that was a completely new universe to me, and I would never have gotten this far without his encouragement. I would like to also thank all my friends who have been by my side as I was writing this thesis. Finally, I want to thank my parents who, even though they don't always fully understand, are still very supportive of me and my academic path.

Abstract

Using interactive theorem provers allows 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. The work is expected to be eventually integrated into Lean's mathlib library.

Contents

1 Introduction **2**

2 Lean Theorem Prover **3**

3 AVL Trees and Operations **4**

3.1 Binary Search Trees 4

3.2 AVL Trees 5

4 Verification **9**

4.1 BST Operations 9

4.2 Rotation 9

4.3 Insertion 11

4.4 Deletion 13

5 Related Work **16**

6 Conclusion **17**

References **18**

1 Introduction

Interactive theorem provers are used to develop formalizations of mathematical and logical concepts. The advantage of using these provers is that proofs can be long, difficult, or almost impossible to write by hand. A famous example is the four-colour theorem, which was one of the first theorems whose result used a lot of computational power, until modern interactive theorem provers. Gontier formalized the four colour theorem in Coq [1], which due to Coq verifying every step, ended up being the most rigorous and accurate formal proof of the four colour theorem.

Interactive theorem provers such as Coq [2], Isabelle [3] and Lean [4] are used to develop these mathematical formalizations. Lean, though much younger than Coq or Isabelle, already possesses mathlib [5], a large library of formalized mathematics. Lean also has a smaller kernel, cleaner syntax with Unicode support and its own metalanguage.

Mathlib, through its extensiveness, has become a de facto standard library in Lean. However, contrary to its counterparts Coq and Isabelle, it is missing formalized tree structures, and only has definitions and tree operations. This thesis serves as a contribution to Lean and the mathlib library in the form of formalizing AVL trees[1].

The paper begins by giving a brief introduction to the Lean theorem prover (Section 2), continuing on to a brief introduction to binary search trees and AVL trees, providing the corresponding Lean types and definitions (Section 3). Afterwards, the verification process is discussed, proving all the necessary proof statements and changes made along the way (Section 4). Finally, the formalization is compared to those made in Coq and Isabelle (Section 5), and the future of this work is outlined (Section 6).

[1] The full code of the tree formalization can be found at the following link: `https://github.com/reglayass/lean-avl`

2 Lean Theorem Prover

The Lean theorem prover is a proof assistant based on dependent type theory with inductive families and universe polymorphism [6]. The language contains dependent function types and inductive families.

In simple type theory, every expression has an associated type – for example, integers, booleans, or functions $\alpha \rightarrow \beta$ where α and β are types. Lean's dependent type theory extends simple type theory by having types themselves be terms [7], which can be constants, variables, applications (i.e. functions), and λ-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 λ-expression. A λ-expression `λx : t` with `t` : α, is a function `x` $\mapsto$ `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 different types. For example, if we have a `list` α, we can differentiate between a list of booleans `list bool` or lists of integers `list` $\mathbb{Z}$.

Dependent types can 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 of `pick n` 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 numbers is the type that depends on it.

In Lean, an inductive type 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`. The type does not have any constructor arguments and is a simple inductive type. Terms can be created from this inductive type by directly using the constructors - for example, zero is constructed with `nat.zero`, one is constructed with `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 [8].

3 AVL Trees and Operations

AVL trees are a type of binary search trees (BSTs) that are self-balancing – meaning, they 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 two children, a node key and a value. The children are referred to as the *left child* and *right child*.

In Lean, binary 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 two empty children, which can be defined with the `node` constructor. For the purpose of this paper, the node key is defined as a natural number; for integration into mathlib, the keys will need to be generalized for other types.

Next, I define a function to check if a key exists 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
| empty := ff
| (node l k a r) :=
  x = k ∨ bound l ∨ bound r
```

A binary search tree must have the *binary search property*. Therefore, with the above definition, a `btree` is only a binary search tree if it has the 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 with 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, a given relation `nat → nat → Prop` (in our applications, $>$ 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 all keys in the left subtree are smaller than the input key, and all the keys in the right subtree are larger than the input key, which is defined with `forall_keys`.

```
def ordered : btree α → Prop
| empty := tt
| (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
| empty := none
| (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 [9] 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 a node is the height difference of its two child subtrees.

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, so 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
| empty := 0
| (node l k a r) :=
  1 + (max (height l) (height r))

def balanced : btree α → bool
| empty := tt
| (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, re-balancing actions are performed 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 $n+2$, where n is the height of the right subtree. 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 [10]. The definition for `right_heavy` is mirrored. The height comparison mentioned previously is alsone done on the children of the subtrees being compared.

```
def left_heavy : btree α → bool
| empty := ff
| (node empty k a r) := ff
| (node (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)
```

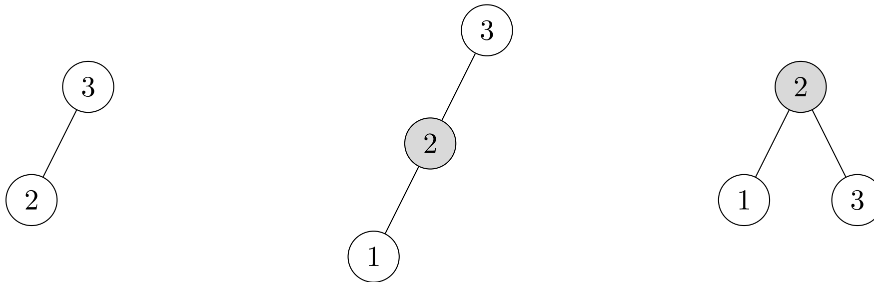

Figure 1

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
| empty := btree.empty
| (node (node ll lk la lr) k a r) :=
   node ll lk la (node lr k a r)
| (node l k a r) := 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 becomes left-heavy, then a right rotation is done to fix the heaviness. The definition for `rotate_left` is mirrored.

```
def rotate_right : btree α → btree α
| empty := btree.empty
| (node l k a r) :=
  match l with
```

```
  | empty := (node l k a r)
  | (node ll lk la lr) :=
    if height ll < height lr
      then simple_right (node (simple_left l) k a r)
      else simple_right (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
| empty := node empty x v empty
| (node l k a r) :=
  if x < k
    then let inl := insert l in let t := (node inl k a r) in
      if left_heavy t
        then rotate_right t
        else t
    else if x > k
      then let inr := insert r in let t := (node l k a inr) in
        if right_heavy t
          then rotate_left t
          else t
      else node l x v r
```

In the current `insert` definition, `(btree.node (insert l) k a r)` is evaluated twice. This is a slight performance issue, but in this paper is done for pedagogical purposes, and will be optimized.

Deleting a node is more complicated, and depends on its children. If its left child is empty, then the node can be removed and replaced by its right child. If the left child 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)
| empty := none
| (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 (node l k v sh))
      else (x, a, node l k v sh)
  end
```

After defining shrinking, 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
| empty := empty
| (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.

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.

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

4 Verification

In this section, we verify the operations presented above. Each subsection 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, as well as details of proofs where 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 [10], though some changes were made 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. Both of the proofs are 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. All the proofs are completed by case splits on the tree `t`. Induction hypotheses are not needed since the rotation definitions are not recursive.

```
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)
```

A lemma for the transitivity of `forall_keys` is needed, and is applied in both the lemmas for compound rotations and simple rotations. One of the hypotheses of the lemma is the the assumption of transitivy. Since `forall_keys` has key relations in the definition, we can apply this assumption to complete the proof.

```
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 rotation restores balance, 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)
```

As with proofs about ordering, the above proofs were done by 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 suffices to find the goals in the hypotheses, additionally using the transitivity lemma to achieve the goal. The balancing lemmas require some simple 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 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 two situations – where we want show that rotations don't remove keys, or where we want to show that rotations don't add keys that are not previously there.

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.

During the construction of these proofs, a 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 by reflexivity.

Order

Th 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. For example, 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 need to be solved 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 are completed by 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 (node l k a r) →
    forall_keys p x (rotate_right (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 (node l k a r) →
    forall_keys p x (simple_right (node l k a r))
```

Balance

The lemma for insertion preserving balanced is formalized as follows and is proven by induction on the tree `t`. It is not entirely complete, which is an element for future work.

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

Due to the structure of this lemma and the definition for `balance`, small changes were made to the `insert` definition. Previously, the definition followed [10] 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 unnecessarily difficult. Therefore, in the definition, the cases that determine whether or not a rotation is done was changed from manual height comparisons to the `left_heavy` and `right_heavy` definitions. With this change, during the proof construction, there is a case split on heaviness, and after applying a rotation lemma the goal becomes 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. Before he proof could be completed, auxiliary proofs with `forall_keys` and proofs about keys and boundedness with shrinking needed to be completed. First, we follow the steps taken to write the lemmas for order, then lemmas about key preservation are discussed specifically. Lastly, certain design changes or additions made during this process are presented.

The proofs for deletion preserving keys and balance are not completed, and is an element for future work on the formalization.

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`. But 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`.

```
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)
```

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`.

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 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 do not 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_char {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, definitions are simplified 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 three possible cases that can come out of shrinking a tree with one action, reducing boilerplate. The same problem arose with `del_node`. To solve these problems I define two views for `shrink` and `del_node`.[2]

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 arbitrary functions appearing in the indices of an inductive family are not well-supported in Lean. The `shrink_view` is shown below.

[2] This approach was suggested by Jannis Limperg

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

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 lemma for `del_node` is almost identical.

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

Applying the lemma results in three cases in the inductive step of `shrink_ordered`, which match 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 [2] and Isabelle/HOL [3] interactive theorem provers. Lean does not have an AVL tree formalization yet, and the only other tree present in the 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. 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 time since a tree is balanced only when 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, which 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. This creates overhead in calculating the height, which is a shortcoming that can be fixed before mathlib integration, by including the height of the tree in the inductive type like in Coq and Isabelle.

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 [5], 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

In this thesis, I presented a full formalization of AVL search trees using Lean, something that is not yet present in the mathlib library, with the tree structures present only including definitions and operations. Lean proof construction is a relatively simple and mechanical task, once the main hurdle of learning and getting used to the syntax is passed. Overall, the proof statements presented in Lean were not very different to those presented informally in [10], although some very small changes did had to be made. Additionally, definitions regarding the structure of AVL trees such as `balance` and `height` stay true to their mathematical definitions for trees.

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, like including the height in the definition of the tree instead of calculating the height of a tree on-demand. The final step is the inclusion of this final formalization into the mathlib library.

References

[1] G. Gonthier, “The Four Colour Theorem: Engineering of a Formal Proof,” Jan. 2007, p. 333, ISBN: 978-3-540-87826-1. DOI: 10.1007/978-3-540-87827-8_28.

[2] “Library Coq.MSets.MSetAVL,” [Online]. Available: https://coq.inria.fr/library/Coq.MSets.MSetAVL.html.

[3] T. Nipkow and C. Pusch, “Avl trees,” *Archive of Formal Proofs*, Mar. 2004, https://isa-afp.org/entries/AVL-Trees.html, Formal proof development, ISSN: 2150-914x.

[4] “Lean Theorem Prover,” [Online]. Available: https://leanprover.github.io/.

[5] The 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.

[6] P. Dybjer, “Inductive Families,” *Formal Aspects of Computing*, vol. 6, pp. 440–465, 1994.

[7] J. Avigad, L. de Moura, and S. Kong, *Theorem Proving in Lean*, version 3.23.0. [Online]. Available: https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf.

[8] J. Avigad, G. Ebner, and S. Ullrich, *The Lean Reference Manual*, version 3.3.0. [Online]. Available: https://leanprover.github.io/reference/lean_reference.pdf.

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

[10] 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.