CS 747 Final Project — Formalizing AVL Trees in Rocq

RICHARD ZHANG, University of Waterloo, Canada

In this project, I implemented AVL trees in Rocq (encoded directly as Gallina programs) and proceeded to verify many key properties of AVL trees, including proofs of correctness for insertion, deletion, BST search, and a proof of logarithmic height for AVL-balanced trees.

1 INTRODUCTION

1.1 The Original Vision

My original aim with this project was to "formalize various self-balancing binary search trees and prove key properties relating to their correctness and balancing properties". I specifically singled out three self-balancing binary search trees (BBSTs for short) of interest: Red-Black trees, AVL trees, and so-called "size-balanced" trees. I also listed out a couple of key properties that I wished to prove:

- Correctness of insertion, deletion, search, and order-statistics.
- BST invariance of tree-modifying operations (namely insertion and deletion); ie. binary search trees remain binary search trees after said operations.
- Balance invariance of tree-modifying operations — eg. AVL trees remain AVL balanced after insertion.
- Logarithmic depth of BBSTs.

Many of the finer points of implementation were intentionally left vague or unanswered in my initial proposal (such as the choice of Proof Assistant, whether to shallowly or deeply embed, etc.) in the hopes that the successes and setbacks I experienced over the course of implementation would provide for a far more informed decision compared to one made at the outset.

1.2 Final Results

In the end, I only followed through with formalizing 1 of the 3 singled-out BBSTs: that being AVL trees. I conducted the formalization through the Rocq Proof Assistant, directly encoding AVL trees and its operations as Gallina programs.

I implemented insertion (`insert`), deletion (`delete`), and BST search (`Contains`) (leaving out order-statistics operations), and proved all of the relevant properties I initially set out as goals — all without any use of `Admitted`. I also proved that AVL-balanced trees have logarithmic height. All of the key final results are contained in the file `AVLResults.v`.

Here is a taste of some of the key results, chosen somewhat arbitrarily:

- AVL trees remain AVL trees after insertion:

```
Theorem AVL_insert x t : AVL t → AVL (insert x t).
```

- AVL trees remain AVL trees after deletion:

```
Theorem AVL_delete x t : AVL t → AVL (delete x t).
```

- AVL trees have logarithmic height:

```
Theorem AVL_height_upperbound t : AVL t → height t ≤ 2 * Nat.log2 (size t) + 1.
```

- `insert` is idempotent for AVL trees:

```
Theorem AVL_insert_idempotent x t : AVL t → insert x (insert x t) = insert x t.
```

See the Appendix, or refer to the source files to view all of the key results (and definitions).

Author's address: Richard Zhang, rx5zhang@uwaterloo.ca, University of Waterloo, Canada.

2 DESIGN DECISIONS

2.1 Choice of Proof Assistant

I quickly settled on Rocq as our Proof Assistant of choice. This decision rested solely on one factor: the ease with which one could cobble together quick-and-dirty tactics and automation.

Despite the many, *many* quality-of-life improvements Lean would have offered over Rocq (such as dot-notation, a unified approach to typeclasses, a robust standard library, proper namespacing, etc.), it fell short in one crucial area: it was far more difficult to write custom tactics in Lean as compared to Rocq — and therefore far more difficult to realize "quick-and-dirty" automation.

As a concrete example, there is no exact equivalent to Rocq's `match goal with …` construct, much less something like `match goal with | context[…] ⇒ _ end`. This means that something like my `rewrite_match_compare` tactic (which I wrote up very quickly to abstract away a common pattern that appeared) would have been a major pain to write in Lean (see Fig. 1).

```
Ltac rewrite_match_compare :=
  repeat match goal with
  | [ |- context[match compare ?x ?x with | _ ⇒ _ end] ] ⇒
      rewrite compare_refl
  | [ H : ?x < ?y |- context[match compare ?x ?y with | _ ⇒ _ end] ] ⇒
      rewrite (proj2 (compare_lt_iff x y) H)
  | [ H : ?y < ?x |- context[match compare ?x ?y with | _ ⇒ _ end] ] ⇒
      rewrite (proj2 (compare_gt_iff x y) H)
  | [ H : ?x < ?y, H2 : context[match compare ?x ?y with | _ ⇒ _ end] |- _] ⇒
      rewrite (proj2 (compare_lt_iff x y) H) in H2
  | [ H : ?y < ?x, H2 : context[match compare ?x ?y with | _ ⇒ _ end] |- _] ⇒
      rewrite (proj2 (compare_gt_iff x y) H) in H2
  end.
```

Fig. 1. My `rewrite_match_compare` tactic, which looks for all occurrences of `match compare x y with …` (in both the goal and all hypotheses) and tries to simplify it with known assumptions regarding `x` and `y` already present in the context.

2.2 Restriction to AVL Trees

My original scope turned out to be too ambitious — making progress on verifying just *one* kind of BBST was already taking a large amount of time. Thus, I narrowed my goal down to verifying just one kind of tree, and left more up as stretch goals. I also chose to verify AVL trees specifically out of the three initial choices, since full-fledged formalizations of AVL trees were far more uncommon compared to those of Red-Black trees while still being more concretely do-able compared to size-balanced trees (which, despite its name, is quite different from *weight*-balanced trees).

Indeed, I was unable to find any full formalizations of AVL trees aside from a very old Rocq formalization (`FSetFullAVL` and `MSetFullAVL`) that used to be in the Rocq Standard Library, but seems to be missing nowadays. [1]

2.3 Choice of Encoding and Definitions

I decided to shallowly embed AVL trees in Gallina since:

[1]There are dead hyperlinks still referring to these implementations in the Rocq Standard Library; the only way I was able to access these was through the `rocq-archive` repository on Github.

(a) It was the easiest option for encoding — the translation between commonly seen code for AVL trees and the Rocq encoding would be immediate.
(b) It was much easier to reason with — you can directly use Rocq's inductive types and recursive functions to encode information about these trees and directly compute values.
(c) It allowed me to immediately get at the problem at hand (proving properties about AVL trees) and avoid an intermediate step of having to define syntax and semantics *just to state* the problems.

While the use of a library like Iris could help alleviate some of the pains described in (c), it would only add to the list of things I needed to familiarize myself with *on top of* the immediate problems I wished to solve.

2.3.1 The Influence of Problem Priority on Definitions. My desire to get at the core of the problem also manifests itself in my definitions; for instance, my binary tree definition does not contain a field for height information:

```
Inductive tree : Type :=
| Nil  : tree
| Node : ∀ (v : A) (l : tree) (r : tree), tree.
```

Fig. 2. Definition of binary tree used throughout the formalization.

This means that there is no need to juggle around height information, and no need to verify that the height value always corresponds with the actual tree height. However, this comes with the obvious trade-off that my implementation is not performant — every call to `height` is $\mathcal{O}(h)$ (where h is the height of the tree) rather than the $\mathcal{O}(1)$ expected in any serious use-case of AVL trees. This was a trade-off I deemed reasonable, as I believed that showing you could properly juggle this height information was a superficial task compared to the harder tasks of eg. proving that AVL balance is preserved by insertion and deletion.

2.3.2 Transparency of Definition. My wish for the translation between the Rocq code and common code to be readily transparent also manifests itself in my decision to split the tree balance operations into separate rotation and balance procedures, as is commonly seen in imperative versions of AVL trees; compare my Rocq and C++ implementations of `rotate_left` and `balance_left`:

Thus, even though the Rocq and C++ versions of `rotate_left` (and thus also `rotate_right`) look fairly different (with the C++ using destructive reassignments), the `balance_left` operations for both languages remain remarkably similar as a result of abstracting out the rotations! This makes it easier to write correct C++ code (or indeed, of any language of your choosing) based on the proven-to-be-correct Rocq code.

2.3.3 Fixpoint vs. Inductive Predicates. Finally, one last point I wish to draw attention to regarding definitions is the dual usage of both `Fixpoint` predicates and `Inductive` predicates — for many of my predicates on trees, I have two versions: a computable `Fixpoint` version, and a non-computing `Inductive` version, bound together with a lemma that ensures equivalence of the two definitions. For example, I have two predicates for testing element membership in a tree, `In` and `In'`, together with a lemma `In'_iff_In`:

```
Definition rotate_left v l r : tree :=
  match r with
  | Node rv rl rr ⇒ Node rv (Node v l rl) rr
  | Nil ⇒ Node v l r
  end.

Definition balance_left (v : A) (l r : tree) : tree :=
  if (1 + height r) <? (height l) then
    match l with
    | Nil ⇒ assert_false v l r
    | Node lv ll lr as l ⇒
        if (height lr <=? height ll)%nat then
          rotate_right v l r
        else
          rotate_right v (rotate_left lv ll lr) r
    end
  else
    Node v l r.
```

```
void rotate_left(Node*& t) {
  Node* r = t->r;
  if (!r) return;
  t->r = r->l;
  r->l = t;
  update(t), update(r);
  std::swap(t, r);
}

void bal_left(Node*& t) {
  if (!t) return;
  if (1 + height(t->r) >= height(t->l)) return;
  if (height(t->l->r) <= height(t->l->l))
    rotate_right(t);
  else
    rotate_left(t->l), rotate_right(t);
}
```

Fig. 3. Implementations of `rotate_left` and `balance_left` in Rocq and C++ respectively.

```
Fixpoint In (x : A) (t : tree) : Prop :=
  match t with
  | Nil ⇒ False
  | Node v l r ⇒
      v = x ∨ In x l ∨ In x r
  end.
```

```
Inductive In'  (x : A) : tree → Prop :=
| In'_node      : ∀ l r, In' x (Node x l r)
| In'_left      : ∀ y l r, In' x l → In' x (Node y l r)
| In'_right     : ∀ y l r, In' x r → In' x (Node y l r).
```

Fig. 4. Definitions for `In` and `In'` respectively.

```
Lemma In'_iff_In x t : In' x t ↔ In x t.
```

Fig. 5. Coherence lemma connecting `In` with `In'`.

Arguably the biggest reason for why I did this was due to simple ignorance — I wished for an easy way to do induction on this predicate, without realizing that `functional induction` existed and could fill this role. Indeed, after I re-discovered the existence of `functional induction`, I was able to greatly simplify many of proofs (and in the process also greatly speed them up).

However, it also turned out that it was often the case that automation behaved better with inductive predicates vs. with recursive predicates. We'll explore a concrete example of this later, during our discussion on Tactics.

3 PROOF PROCESS INSIGHTS

In retrospect, with *good and correct* definitions, the proofs of most lemmas and theorems were not too tough — the flow of smaller lemmas into larger lemmas into complete theorems was fairly natural feeling. *However*, it should be noted that the words "good and correct definitions" are doing a lot of heavy lifting in that sentence. In this section, we'll examine some of the pains I experienced due to a lack of "good and correct definitions", and briefly discuss my proof of logarithmic height for AVL trees.

3.1 The Pains of Clunky Definitions

One of the definitions that I regretted the most later into the development was my definition of the AVL balance invariant. I chose to define it like so:

```
Inductive Balanced : tree → Prop :=
(* a leaf node is Balanced *)
| Balanced_nil : Balanced Nil
(* two children of equal height is Balanced *)
| Balanced_equal :
  ∀ l r, Balanced l → Balanced r → height l = height r → ∀ v, Balanced (Node v l r)
(* if height l = height r + 1, then the resulting thing is Balanced *)
| Balanced_left_heavy :
  ∀ l r, Balanced l → Balanced r → height l = 1 + height r → ∀ v, Balanced (Node v l r)
(* symmetric. *)
| Balanced_right_heavy :
  ∀ l r, Balanced l → Balanced r → 1 + height l = height r → ∀ v, Balanced (Node v l r).
```

Contrast this with the definition used in the Standard Library's FSetFullAVL:

```
Inductive avl : tree -> Prop :=
  | RBLeaf : avl Leaf
  | RBNode : forall x l r h, avl l -> avl r ->
      -(2) <= height l - height r <= 2 ->
      h = max (height l) (height r) + 1 ->
      avl (Node l x r h).
```

Putting aside the fact that the FSetFullAVL uses a very non-standard definition of AVL trees [2], there is still a clear difference between the two definitions — my definition makes explicit all possible cases for the AVL invariant right in the definition, whereas FSetFullAVL groups it all together behind the (linear) constraint that $-2 \leq \text{height } l - \text{height } r \leq 2$.

In theory, my definition still has a place in a sort of pedagogical way: induction on Balanced directly inducts on all possible cases. However, in practice, a lot of work (especially case-work to rule out impossible cases) is being done by tactics like lia. And lia *excels* at dealing with information like $-2 \leq \text{height } l - \text{height } r \leq 2$ — both as a hypothesis and as a goal to be eliminated.

Another factor that plays into why my definition is poor is my definition of height — though it is not immediately evident from the definition of Balanced alone.

```
Fixpoint height (t : tree) : nat :=
  match t with
  | Nil ⇒ 0
  | Node _ l r ⇒ 1 + Nat.max (height l) (height r)
  end.
```

The type signature of my height function is `height : tree → nat`. While innocuous, the fact that it returns a nat and not an integer means that even simple things like `height l - height r` don't behave how they look like they should behave [3], thus leading to rather convoluted statements like,

[2]They require the balance factor to be within ±2, rather than the far more common ±1; in fact, this is the first time I've ever seen ±2 used. Nevertheless, the fact that it compiles gives confidence that it still yields a self-balancing binary search tree. Another win for proof formalization!

[3]Subtraction of nats is truncated by default.

```
Lemma balance_lemma v l r :
  Balanced l → Balanced r →
  (height r <= height l <= 1 + height r) ∨
  (height l <= height r <= 1 + height l) →
  Balanced (Node v l r).
```

Rather than the more obvious,

```
Lemma balance_lemma v l r :
  Balanced l → Balanced r → (-(1) <= height l - height r <= 1) → Balanced (Node v l r)
```

(As a side note, this `balance_lemma` was how I eventually bridged the gap between my definition and the `FSetFullAVL` definition of the AVL invariant.)

To summarize:

- If it's possible to subsume inductive cases into a single linear inequality, it's often worth it as it enables powerful tactics like `lia` to do the case-work for you.
- It is often worth it to use integer-valued functions, even when `nat`-valued functions suffice, as it allows tactics like `lia` and `ring` to perform a wider range of operations, assumptions, and simplifications (plus, it often leads to more concise formulations of the same information).

3.2 The Pains of Incorrect Definitions

For several days, I was stuck trying to show that inserting into an AVL-balanced tree preserved the AVL-balance invariant.

I initially started out trying to directly prove this stated goal:

```
Theorem insert_preserves_Balanced x t : Balanced t → Balanced (insert x t).
```

But it quickly became apparent that this was insufficient. I then tried augmenting my induction hypothesis like so:

```
Lemma insert_preserves_Balanced0 x t :
  Balanced t → Balanced (insert x t) ∧ height t ≤ height (insert x t)
```

But it was *still* not enough. In fact, I was unable to even show that the following fact was true:

```
Lemma hmmm v l r :
  Balanced l → Balanced r → 2 + height l = height r →
  Balanced (balance_right v l r).
```

If you are familiar with the proofs for AVL trees, this might be ringing a few alarm bells — this *should* be true, and it *should* be provable. However, I was sadly not too experienced with AVL trees — besides the fact that they were self-balancing binary search trees with a balancing criteria related to their height, I knew little else about them. In fact, I had never even coded one prior to this project! [4]

In the end, I managed to prove this theorem with a very technical choice of induction hypothesis:

```
Lemma insert_preserves_Balanced0 x t :
  Balanced t →
  Balanced (insert x t) ∧
    height t ≤ height (insert x t) ∧
    (t ≠ Nil →
      1 + height t = height (insert x t) →
      height (left_child (insert x t)) ≠ height (right_child (insert x t))).
```

[4] I've always personally used treaps and splay trees as my BBSTs of choice!

The essential idea behind the last condition is that insertion only increases the height of (at most) one subtree — it cannot increase the height of both subtrees simultaneously. Thus, at any point during the insertion, we will only ever need to fix *one* broken AVL invariant, not *two*.

Fantastic! Let's move on to showing the same thing, but for deletion then — or so I'd hoped. It turns out, no matter what, I could not prove that deletion preserved the AVL balance invariant. In fact, I was able to construct *an explicit counterexample* that could not be re-balanced.

Why would this possibly be the case!? Was the entire body of literature on AVL trees *wrong*? Of course not. Rather, my own definitions were. Take a look at this *correct* definition of `balance_left`, my rebalancing function which rebalances a possibly heavy left child:

```
Definition balance_left (v : A) (l r : tree) : tree :=
  if (1 + height r) <? (height l) then
    match l with
    | Nil ⇒ Node v l r
    | Node lv ll lr as l ⇒
        if (height lr <=? height ll)%nat then
          rotate_right v l r
        else
          rotate_right v (rotate_left lv ll lr) r
    end
  else
    Node v l r.
```

And now look at this very slightly different definition, the definition I had been working at this earlier point in time:

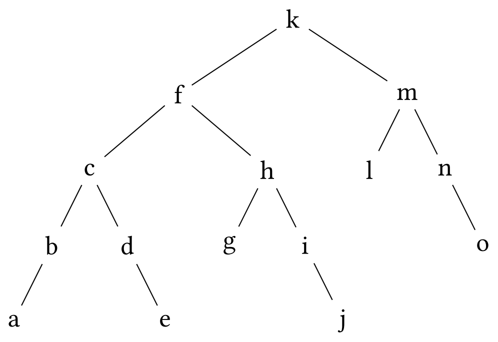
```
Definition balance_left (v : A) (l r : tree) : tree :=
  if (1 + height r) <? (height l) then
    match l with
    | Nil ⇒ Node v l r
    | Node lv ll lr as l ⇒
        if height lr <? height ll then
          rotate_right v l r
        else
          rotate_right v (rotate_left lv ll lr) r
    end
  else
    Node v l r.
```

The difference is *one character*: rather than the correct check of `height lr <=? height ll`, I instead had `height lr <? height ll`. That's it. That single character (well, together with its symmetrical counterpart in `balance_right`) was why my tree failed to rebalance deletions.

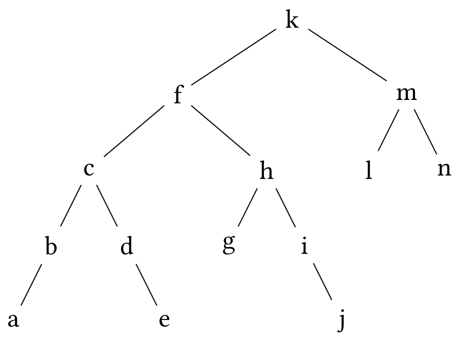
Indeed, consider the deletion showcased in Fig. 6, where the right-most element is deleted from an AVL-balanced tree.

After the deletion of h, the tree looks like it does in Fig. 6(b). Here, we note that the tree is left-heavy; the left-child, c, has a height of 3 whereas the right-child, g, has a height of 1 — thus the AVL invariant has been broken and needs to be restored; specifically, `balance_left` is called at the root.

Compare now the effects of the correct `balance_left` and the incorrect `balance_left`, as shown in Fig. 7. The correct `balance_left` does indeed produce an AVL-balanced tree (see Fig. 7(a)), but

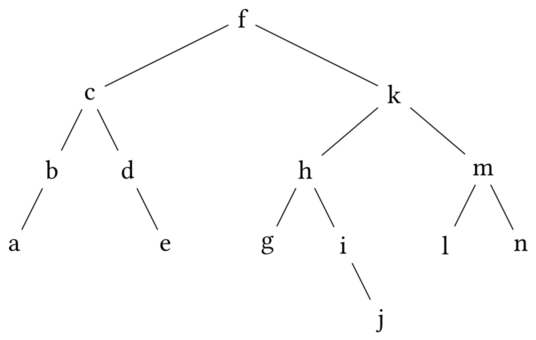


(a) The original tree

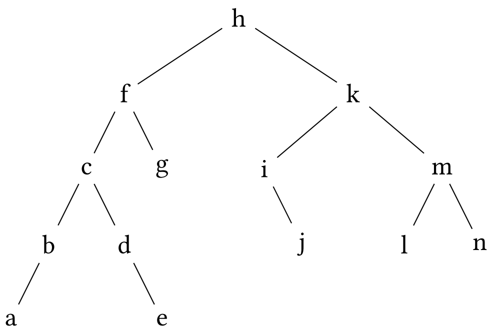


(b) After deleting o

Fig. 6. Deleting the right-most element, o, from an AVL-balanced tree.



(a) Behaviour of the correct `balance_left`



(b) Behaviour of the incorrect `balance_left`

Fig. 7. Behaviour comparison between the correct and incorrect `balance_left` implementations.

the incorrect `balance_left` fails to produce an AVL-balanced tree, as the tree rooted at f has a balance factor of -2 (see Fig. 7(b)).

With all that said, the interesting takeaway here for me personally is two-fold:

- It is *incredibly* easy to write an incorrect AVL tree that looks superficially correct.
- It is very cool that the "incorrect" AVL tree still preserves AVL balance on insertions, despite failing on deletions. Though, this makes sense — we were able to prove the insertion theorem thanks to the fact that insertion only ever breaks one invariant at a time, however deletion possibly allows for two invariants to be simultaneously broken.

The proof that the "incorrect" AVL tree still preserves AVL balance on insertions is documented in the file `AVL_incorrect.v`.

3.3 Proof of Logarithmic Height

I tackled the proof of logarithmic height for AVL trees by following the proof sketch outlined in https://people.csail.mit.edu/alinush/6.006-spring-2014/avl-height-proof.pdf

The key points in the proof are as follows:

- Define $N(h)$ to be the minimum number of nodes in an AVL tree of height h. (`min_size_of_height` in my proof.)
- Make the observation that an AVL tree with minimum size must have subtrees of differing heights (for otherwise, we could remove at least one node and still have the same height). (`balanced_uneven` in my proof.)
- Make the observation that an AVL tree of minimum size must have children which are AVL trees of minimum size. (`child_of_min_size_is_min_size_l`, `child_of_min_size_is_min_size_r`.)
- Conclude that $N(h) = 1 + N(h-1) + N(h-2)$ and that $N(0) = 0$, $N(1) = 1$. (`min_size_eqn`, `min_size_0`, `min_size_1`)
- Conclude that N is a strictly increasing function. (`min_size_strict_mono`)
- Conclude that $2^{h/2} \leq N(h)$. (`min_size_even`, `min_size_odd`)
- Take logarithms on both sides and find that $h/2 \leq \log_2 N(h)$ and so $h \leq 2\log_2 N(h) + 1$. (`min_size_log`)
- Conclude that, for a given AVL tree of size n and height h, since $N(h)$ is always the minimum size of an AVL tree of height h, therefore $N(h) \leq n$, therefore $h \leq 2\log_2 N(h) + 1 \leq 2\log_2 n + 1$, thus all AVL trees have logarithmic height. (`height_upperbound0`, `height_upperbound`)

The most interesting aspect of my proof is the way I handled $N(h)$ — since $N(0) = 0$, $N(1) = 1$, and $N(h) = 1 + N(h-1) + N(h-2)$, you could define $N(h)$ independently of the fact that it represents the minimum size of an AVL-balanced tree of height h! Indeed, this is the approach taken by the proof writers of `FSetFullAVL`.

However, I instead opted to define $N(h)$ very literally, by following this line of reasoning:

- Every perfect binary tree is AVL-balanced, and it's easy to construct a perfect binary tree for any given height h. (`perfect_tree`, `perfect_tree_height`, `perfect_tree_balanced`)
- Thus, for every height h, there exists a size s such that [there exists an AVL-balanced tree of size s]. (`exists_tree_of_height`, `exists_size_of_height`, `exists_min_size_of_height`)
- By the Well-Ordering Principle, there must exist a least-such s for every height h — this s must be exactly $N(h)$. (`min_size_of_height`)

Unfortunately, since I was looking at an outdated version of the Rocq standard library [5] which did not provide a proof of minimality for its version of the Well-Ordering Principle, I had to code my own. Fortunately, I was able to copy the proofs that the Lean Proof Assistant uses for its rendition of WOP called `Nat.find` — this is why my WOP function is called `nat_find`.

Also, very candidly, now that I'm re-examining my code for `exists_min_size_of_height`, I'm not too sure why I have a redundant a clause for minimality, considering that WOP already guarantees minimality. But this is likely a consequence of developing 90% of the proof for logarithmic height

[5]The webpage defaults to v8.9, rather than the much more recent v8.20 I developed my formalization with.

over the course of a single all-nighter (the night before the midway check-in). And hey, if it works, it works — and we can always mark it as a point of future work.

4 TACTIC TAKEAWAYS

Tactics, especially custom tactics and proof automation, played an essential role in the formalization process. Without the very, *very* extensive use of programmable proof search and computer-assisted case bash, I do not think that I would have been able to achieve what I have — they allowed me to focus on writing down statements of lemmas, rather than proofs of lemmas, so I could focus on the forest rather than the trees.

In my experience, there were two overarching themes relating to tactic usage and refinement which popped up again and again:

- Whenever possible, abstract out useful lemmas and automate their proofs. (Just like DRY in software engineering!)
- Big hammers solve big problems, but come at the cost of lots of computation time. It's almost always possible to find smaller hammers which can do the same thing in much less time.

At one point, I was relying so heavily on so many unoptimized "big hammer"-type tactics my compilation time reached 200 seconds. Every time I needed to make a change at the top of my file (such as a change in definitions or a refactor), I would have to wait an entire *three minutes* to continue proving things at the bottom. Spending just a few minutes to optimize my tactics thus ended up saving me quite a lot of time [6] — time I could then spend more productively doing other things.

Thus, there was always a constant tension between the time spent creating and optimizing tactics and the time spent actually proving theorems (and computing with tactics).

4.1 Hints

Wherever possible, `Hint`! Whether that be `Hint Resolve`, `Hint Rewrite`, `Hint Extern`, `Hint Constructor`, or even `Hint Unfold`.

Hints were *by far* the aspect of proof automation that had the lowest cost and highest reward — something that became more and more blindingly clear as time went on.

There really were only a few things that had to be kept in mind with regards to Hints:

- Be careful that `Hint Rewrite` doesn't rewrite away from more useful proof states.
- Make sure `Hint Rewrite` never constitutes a rewrite-loop.
- If `auto`/`eauto`/`crush`/[any other proof-search tactic] ever results in a proof state that *feels* trivial to resolve/to advance, chances are good that proper `Hint`ing can completely elide such goals.
- `Hint`s do not persist through `Section`s — I had to learn this the hard way.
- Always remember to Hint! (Although slightly glib, it turns out it's very easy to forget to `Hint`, even despite the blindingly obvious upsides.)

4.2 On crush

The `crush` tactic from *Certified Programming with Dependent Types* (CPDT) was one of, if not *the* biggest hammers in my toolbox. As such, it embodied both the extreme positives (could prove very complex goals with a single tactic call) and negatives (immense computation time) of "big hammer" tactics into sharp relief. Thus, it was the perfect tool for prototyping — as long as you remembered to actually return and optimize the proof.

[6] I was able optimize it down from 200 seconds to 100 seconds, and later down to just 70 seconds.

4.2.1 *An example.* This was the original proof developed using `crush`:

```
Lemma rotate_right_preserves_Ordered v l r :
  Ordered (Node v l r) → Ordered (rotate_right v l r).
Proof.
  rewrite !Ordered_iff_Ordered' /rotate_right.
  intros; destruct l; invert_ordered'; try by constructor.
  constructor; crush.
  apply (All_imp (fun x ⇒ v < x)); by crush.
Qed.
```

Thanks to `crush`, it's a very short proof. However, it's also a very slow proof. Just the line `constructor; crush` took over 3 seconds to run!

```
time (constructor; crush).
```

```
Tactic call ran for 3.159 secs (3.152u,0.006s) (success)
```

Now, here is the updated proof:

```
Lemma rotate_right_preserves_Ordered v l r :
  Ordered (Node v l r) → Ordered (rotate_right v l r).
Proof.
  rewrite !Ordered_iff_Ordered' /rotate_right.
  intros; destruct l; invert_ordered'; try by constructor.
  constructor; simpl'; eauto 2.
  apply (All_imp (fun x ⇒ v < x)); by eauto 2.
Qed.
```

It is slightly more complicated; calls to `crush` here have been replaced by either `simpl'; eauto 2` or just `eauto 2` — replacements that took a bit of trial and error to find.

However, the upshot of doing this trial and error to replace all calls to `crush` is that it is *significantly* faster. The line `constructor; simpl'; eauto 2` leaves the proof in the exact same state that `constructor; crush` previously did. But, compare the time this takes to run with the earlier time:

```
time (constructor; simpl'; eauto 2).
```

```
Tactic call ran for 0.273 secs (0.273u,0.s) (success)
```

So, in short: using `crush` greatly speeds up prototyping. But if you forget to go back and optimize the proofs to not rely on `crush`, you'll be spending a lot of time twiddling your thumbs as it computes.

4.3 Functional Induction

Speaking very candidly, at the start of the project I had mildly forgotten that functional induction was a feature available in Rocq.

It turns out functional induction is kind of useful. And by kind of useful, I mean *really, really useful.* For one thing, it made it very obvious what I needed to induct on in every proof. For another, it made a lot of the case-work needed to simplify the function redundant, thereby saving a lot of time.

4.3.1 An example. This was the original proof, where we manually unfolded the definition of balance_right using the tactic split_ifs and doing case-work on r via destruct r:

```
Lemma balance_right_preserves_Balanced v l r :
  Balanced (Node v l r) → Balanced (balance_right v l r).
Proof.
  move⇒Hbal.
  rewrite /balance_right.
  split_ifs.
  - destruct r.
    + by crush.
    + split_ifs; invert Hbal; by crush.
  - assumption.
Qed.
```

Of note, the line split_ifs; invert Hbal; by crush took over 3 seconds to run by itself.

```
time (split_ifs; invert Hbal; by crush).
```

```
Tactic call ran for 3.211 secs (3.197u,0.013s) (success)
```

Now, compare this to the new proof using functional induction:

```
Lemma balance_right_preserves_Balanced v l r :
  Balanced (Node v l r) → Balanced (balance_right v l r).
Proof.
  move ⇒ Hbal.
  functional induction (balance_right v l r); simplify; eauto; invert Hbal; by crush.
Qed.
```

The second line — the bulk of the proof — takes only a fraction of the time to run, despite still calling this same snippet of invert Hbal; by crush!

```
time (functional induction (balance_right v l r);
      simplify; eauto; invert Hbal; by crush).
```

```
Tactic call ran for 0.316 secs (0.312u,0.003s) (success)
```

4.4 Custom Tactics

Rocq makes it very easy to write quick-and-dirty tactics to abstract away common patterns — and so I did! In this section I just want to showcase a few of the tactics I wrote that proved to be the most useful.

A common theme that shows up here is that the match goal with construct Rocq provides is the core workhorse of all the most useful tactics.

4.4.1 bal_invert.

```
Ltac bal_invert :=
  match goal with
  | [ H : Balanced (Node _ _ _) |- _] ⇒ invert H
  end.
```

This is an incredibly simple tactic, to the point that explaining what it does is probably an affront to intelligence. But the reason why it's a very useful tactic is slightly more subtle: by running `bal_invert` instead of say, `invert H5`, I'm able to avoid explicitly naming the hypothesis I wish to invert. This serves double duty:

(1) It makes proofs more robust, since automatic variable naming is very fragile and subject to change.
(2) It enables me to incorporate this action into proof automation — other tactics can now perform the action of inverting a `Balanced` hypothesis without me needing to explicitly supply a name.

Naturally, there's nothing special about inverting `Balanced` specifically (apart from the fact that this action shows up very frequently in my proofs) — rather, it demonstrates a principle that is applicable to many similar situations.

4.4.2 False Hypotheses and `Hint Extern`. At some point, we prove this simple lemma:

```
Lemma insert_not_Nil x t : insert x t ≠ Nil.
```

However, it turns out that adding `Hint Resolve insert_not_Nil` seems to do very little — we still often end up with situations where we have a hypothesis of the form `insert x t = Nil` after running `eauto` or the likes.

Rather, adding this custom tactic to `Hint Extern` solves this issue:

```
Hint Extern 1 ⇒
  match goal with
  | [ H : insert _ _ = Nil |- _ ] ⇒ exfalso; exact: insert_not_Nil _ _ H
  end : core.
```

In general, it proved to be very fruitful to `Hint Extern` away commonly seen false hypotheses.

4.4.3 bash_heights.

```
Ltac bash_heights :=
  repeat (simplify; match goal with
  | [ H : height ?t = 0 |- _ ] ⇒
      let h := fresh in
      have h := (height_eq_zero_nil _ H);
      clear H;
      subst
  | [ H : 0 = height ?t |- _ ] ⇒
      symmetry in H;
      let h := fresh in
      have h := (height_eq_zero_nil _ H);
      clear H;
      subst
  | [ H : (_ < 1)%nat |- _ ] ⇒
      rewrite Nat.lt_1_r in H; subst
  | [ H : _ ≤ 0 |- _] ⇒
      rewrite Nat.le_0_r in H; subst
  | [ H : (S _ < S _)%nat |- _ ] ⇒ rewrite -Nat.succ_lt_mono in H
  | [ H : S _ = S _ |- _ ] ⇒ apply Nat.succ_inj in H
  | [ H : S _ ≤ S _ |- _ ] ⇒ apply le_S_n in H
  | [ H : height ?t = 1 |- _ ] ⇒ destruct t; linear_arithmetic'
  | [ H : 1 = height ?t |- _ ] ⇒ destruct t; linear_arithmetic'
  | [ H : height ?t ≤ 1 |- _ ] ⇒ destruct t; linear_arithmetic'
  | [ H : context[Nat.max _ _] |- _ ] ⇒ linear_arithmetic'
  | [ H : context[Nat.min _ _] |- _ ] ⇒ linear_arithmetic'
  end; clear_useless; lia'); try by lia'.
```

This monster of a tactic is one of my proudest creations. It enabled me to brute-force determine the exact heights of trees with heights ≤ 1 by doing case work on all possible heights and subsequently using lia to eliminate all impossible cases. It's safe to say that it simplified many proofs, a *lot*. (Trees of height ≤ 1 frequently appeared as a result of doing case-work on the children of a tree).

4.5 Fixpoint vs. Inductive Predicates

As a concrete example, during the proof this Lemma:

```
Lemma merge_In_complete_left x l r :
  In x l → In x (merge l r).
```

We arrive at this proof state:

```
1 goal (ID 10088)

  x : A
  r : tree
  v : A
  l : tree
  rv : A
  rl, rr : tree
  x_in_r' : In x (Node rv rl rr)
  x0 : A
  e : find_max (Node rv rl rr) = Some x0
  IHP0 : x0 = x ∨ In x (prune_max (Node rv rl rr))
  ============================
  x0 = x ∨ In x (Node v l (prune_max (Node rv rl rr)))
```

If we replace all the Ins with their Inductive counterpart, In', eauto is able to find the proof blazingly fast (in fact, just eauto 3 suffices):

```
time (rewrite <- In'_iff_In in *; destruct IHP0; by eauto).
```

```
Tactic call ran for 0.021 secs (0.021u,0.s) (success)

No more goals.
```

On the other hand, if we directly try attacking the goal at hand with just eauto (and some help from Hint Unfold In), we still get a result, but it is *far* slower:

```
time (destruct IHP0; by eauto).
```

```
Tactic call ran for 0.181 secs (0.181u,0.s) (success)

No more goals.
```

Indeed, we see that eauto 5 is the smallest depth at which eauto succeeds — this is contrasted with the earlier success at only eauto 3. We can manually give it some help by adding a simpl call before eauto, which cuts the search depth requirement from eauto 5 to eauto 4:

```
time (destruct IHP0; simpl; by eauto 4).
```

```
Tactic call ran for 0.063 secs (0.063u,0.s) (success)

No more goals.
```

But this is *still* slower than what we had earlier, by upwards of a factor of 2!.

4.6 `ltac:(eauto)` me a proof term!

Near the end of my time working on the project, I stumbled across a neat trick. But before we get to this trick, a bit of background.

In Lean, tactics have first-class integration with terms: in any place where I would expect a term, I could instead write `by [tactic]` to supply the term using `[tactic]`.

It turns out, I can do the same thing in Rocq! For instance, suppose I had a lemma,

```
H : ∀ x y, x < y → P
```

and I wanted to specialize it in the scenario where $x = 0$ and $y = 1$. Then, instead of doing something like,[7]

```
have h : 0 < 1 by lia.
have {h}H := H _ _ h.
```

I could instead chop it down to just,

```
have {}H := H 0 1 ltac:(lia).
```

In this scenario, `lia` was the choice of tactic to supply the term, but you're certainly not limited to just `lia` — in fact, with proper hinting you're probably going to end up using `eauto` to supply this term most of the time!

5 FINAL REFLECTION AND FUTURE WORK

Although I didn't end up completing the work set out in my original, ambitious scope, and I didn't even end up completing my stretch goals, I am very satisfied at the work that I *have* accomplished.

During my brief perusal of pre-existing formalizations of AVL trees, it turned out that many people have tried to take on the same task, but all of the ones I could find (with the exception of the archived Rocq standard library formalization) were lacking complete proofs of this or that key property — they usually failed to prove that deletion preserves AVL balance and logarithmic height.

Thus, it seems to me that I have accomplished a non-trivial task, and I'm pretty happy with that.

As for future work, here is but a shortlist of possible tasks:

- Refactor things to be less ugly, and to enable my proofs of general binary tree operations like tree-rotations to be re-used elsewhere.
- Incorporate height information directly into the tree to allow for true $\mathcal{O}(1)$ rebalancing.
- Change from using non-performant `nat`s to the performant binary integers.
- Formalize definitions and proofs for order-statistics operations and bulk set operations such as set union and set difference.
- Formalize AVL trees in the setting of imperative code via deep embedding.
- Formalize other BBSTs.

[7]Putting h inside the curly braces clears it after use.

A KEY DEFINITIONS

```
Inductive tree : Type :=
| Nil  : tree
| Node : ∀ (v : A) (l : tree) (r : tree), tree.
```

The definition of binary tree used throughout this formalization.

```
Fixpoint height (t : tree) : nat :=
  match t with
  | Nil ⇒ 0
  | Node _ l r ⇒ 1 + Nat.max (height l) (height r)
  end.
```

Tree height.

```
Fixpoint size (t : tree) : nat :=
  match t with
  | Nil ⇒ 0
  | Node _ l r ⇒ 1 + (size l) + (size r)
  end.
```

Tree size.

```
Fixpoint In (x : A) (t : tree) : Prop :=
  match t with
  | Nil ⇒ False
  | Node v l r ⇒
      v = x ∨ In x l ∨ In x r
  end.
```

Set-membership in a binary tree.

```
Fixpoint Contains x t : Prop :=
  match t with
  | Nil ⇒ False
  | Node v l r ⇒
      match (v ?= x) with
      | Eq ⇒ True
      | Gt ⇒ Contains x l
      | Lt ⇒ Contains x r
      end
  end.
```

BST search. (Equivalent to set-membership for BSTs.)

```
Fixpoint Any (P : A → Prop) (t : tree) : Prop :=
  match t with
  | Nil ⇒ False
  | Node v l r ⇒
      P v ∨ Any P l ∨ Any P r
  end.
```

```
Fixpoint All (P : A → Prop) (t : tree) : Prop :=
  match t with
  | Nil ⇒ True
  | Node v l r ⇒
      P v ∧ All P l ∧ All P r
  end.
```

Functions to check whether a predicate P holds for any/all values in a tree.

```
Definition rotate_left v l r : tree :=
  match r with
  | Node rv rl rr ⇒ Node rv (Node v l rl) rr
  | Nil ⇒ Node v l r
  end.
```

Left-rotation; this rotates the tree `Node v l r` towards its left child l. (`rotate_right` is symmetric.)

```
Definition balance_left (v : A) (l r : tree) : tree :=
  if (1 + height r) <? (height l) then
    match l with
    (* this is never true in a well-formed AVL tree *)
    | Nil ⇒ Node v l r
    (* rather, we will always be in this case *)
    | Node lv ll lr as l ⇒
        if (height lr <=? height ll)%nat then
          (* left-left, one rotation *)
          rotate_right v l r
        else
          (* left-right, two rotations *)
          rotate_right v (rotate_left lv ll lr) r
    end
  else
    Node v l r.
```

This function rebalances the tree `Node v l r` whose left child l is suspected to have a higher height than its right child r.

```
Fixpoint insert x t :=
  match t with
  | Nil ⇒ Node x Nil Nil
  | Node v l r ⇒
      match (v ?= x) with
      | Eq ⇒ Node v l r
      | Lt ⇒ balance_right v l (insert x r)
      | Gt ⇒ balance_left  v (insert x l) r
      end
  end.
```

Standard BST insertion.

```
(* shrink_max removes and returns the maximum (right-most) element of a tree *)
Fixpoint shrink_max (t : tree) : option A * tree :=
  match t with
  | Nil ⇒ (None, Nil)
  | Node v l r ⇒
      match fst (shrink_max r) with
      | None ⇒ (Some v, l)
      | Some x ⇒ (Some x, balance_left v l (snd (shrink_max r)))
      end
  end.

(* merge l r deletes the root, t = (Node _ l r).
* it accomplishes this by swapping the root node's value with the max element of its left subtree,
* then deleting the max element instead.
*)
Definition merge (l r : tree) : tree :=
  match shrink_max l with
  | (None, _) ⇒ r
  | (Some x, l') ⇒ balance_right x l' r
  end.

Fixpoint delete x t :=
  match t with
  | Nil ⇒ Nil
  | Node v l r ⇒
      match (v ?= x) with
      | Eq ⇒ merge l r
      | Lt ⇒ balance_left  v l (delete x r)
      | Gt ⇒ balance_right v (delete x l) r
      end
  end.
```

Standard BST deletion, done by swapping the root value with the right-most child of its left subtree.

```
Fixpoint Ordered (t : tree) : Prop :=
  match t with
  | Nil ⇒ True
  | Node v l r ⇒
      (All (fun x ⇒ x < v) l ∧ Ordered l) ∧ (All (fun x ⇒ v < x) r ∧ Ordered r)
  end.
```

`Ordered t` asserts that *t* is a binary search tree (with no duplicates). (So its inorder traversal orders values from least to greatest)

```
Inductive Balanced : tree → Prop :=
(* a leaf node is Balanced *)
| Balanced_nil : Balanced Nil
(* two children of equal height is Balanced *)
| Balanced_equal :
  ∀ l r, Balanced l → Balanced r → height l = height r → ∀ v, Balanced (Node v l r)
(* if height l = height r + 1, then the resulting thing is Balanced *)
| Balanced_left_heavy :
  ∀ l r, Balanced l → Balanced r → height l = 1 + height r → ∀ v, Balanced (Node v l r)
(* symmetric. *)
| Balanced_right_heavy :
  ∀ l r, Balanced l → Balanced r → 1 + height l = height r → ∀ v, Balanced (Node v l r).
```

AVL-balance invariant predicate; `Balanced t` asserts that the absolute difference of the heights of the children of t is ≤ 1.

```
Inductive AVL t : Prop :=
| AVL_intro : Ordered t → Balanced t → AVL t.
```

Definition of AVL trees — AVL trees are binary search trees which are AVL-balanced.

B KEY RESULTS

Note that all these results are in `AVLResults.v`.

BST Search behaves exactly like set-membership in AVL trees, so checking set membership is $\mathcal{O}(h)$, where h is the height of the tree:

```
Theorem AVL_In_iff_Contains t :
  AVL t → (∀ x, In x t ↔ Contains x t).
```

An AVL tree remains an AVL tree after insertion:

```
Theorem AVL_insert x t :
  AVL t → AVL (insert x t).
```

Insertion preserves all prior elements:

```
Theorem In_insert_of_In x y t :
  In x t → In x (insert y t).
```

The inserted element is guaranteed to be in the new tree:

```
Theorem In_insert x t :
  In x (insert x t).
```

The only elements in `insert y t` are y and the elements in t:

```
Theorem eq_or_In_of_In_insert x y t :
  In x (insert y t) → x = y ∨ In x t.
```

If an element is already present in an AVL tree, then the tree is unchanged after inserting that element:

```
Theorem AVL_insert_eq_of_In x t :
  AVL t → In x t → insert x t = t.
```

AVL trees are idempotent under insert:

```
Theorem AVL_insert_idempotent x t :
  AVL t → insert x (insert x t) = insert x t.
```

An AVL tree remains an AVL tree after deletion:

```
Theorem AVL_delete x t :
  AVL t → AVL (delete x t).
```

The elements of an AVL tree following a deletion forms a subset of the original tree:

```
Theorem AVL_In_of_In_delete x y t :
  AVL t → In x (delete y t) → In x t ∧ x ≠ y.
```

The deleted element is not in the new tree.

```
Theorem AVL_not_In_delete x t :
  AVL t → ¬ In x (delete x t).
```

If x was already in t, and x is not equal to the deleted element, then x is in the new tree:

```
Theorem AVL_In_delete_of_In_of_neq x y t :
  AVL t → In x t → x ≠ y → In x (delete y t).
```

If x was not present in t, then deleting x from t does nothing:

```
Theorem AVL_delete_eq_of_not_In x t :
  AVL t → ¬ In x t → delete x t = t.
```

AVL trees are idempotent under deletion:

```
Theorem AVL_delete_idempotent x t :
  AVL t → delete x (delete x t) = delete x t.
```

AVL trees have logarithmic height, that is, they have a height of $\mathcal{O}(\log_2 n)$, where n is the number of nodes in the tree:

```
Theorem AVL_height_upperbound t :
  AVL t → height t ≤ 2 * Nat.log2 (size t) + 1.
```