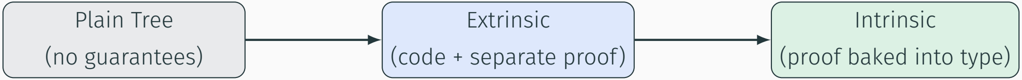
Verified Binary Search Trees in Lean 4

From Code to Proof — Three Approaches

The Big Picture



Approach	Write code	Write proofs	Can express bugs?
Plain	yes	no	yes
Extrinsic	yes	yes (separate)	yes, but caught
Intrinsic	yes	yes (inline)	**no**

Getting the Code

Fetch the code for today's lecture:

```
git clone https://www.github.com/maksym-radziwill/TT
cd TT/LeanStuff
```

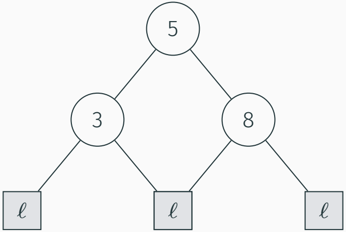
Open `LeanStuff.lean` in VS Code with the Lean 4 extension. We'll need it open throughout the lecture — when we get to the proofs, reading them on slides alone isn't enough. You need to click through each line and see the **proof context** (goals, hypotheses) update in the Lean Infoview panel.

Part 1: Plain Binary Tree

Defining the Tree Type

```
inductive Tree : (α : Type) → Type where
  | leaf : Tree α
  | node : (left : Tree α) → (val : α)
         → (right : Tree α) → Tree α
```

- α is the element type (generic / polymorphic)
- Two constructors: an empty `leaf`, or a `node` with left subtree, value, right subtree
- This is just *data* — nothing prevents us from storing elements in the wrong order




```
-- This tree in Lean syntax:
def exampleTree : Tree Nat :=
  .node (.node .leaf 3 .leaf) 5 (.node .leaf 8 .leaf)
```

Computing the Size

```
def Tree.size (t : Tree α) : Nat :=
  match t with
  | .leaf => 0
  | .node left _ right => 1 + left.size + right.size
```

Why `Nat` and not `Int`?

A tree can never have a negative number of nodes. Using `Nat` makes that obvious from the type alone. This is a mini example of "making invalid states unrepresentable."

In-Order Traversal

```
def Tree.toList (t : Tree α) : List α :=
  match t with
  | .leaf => []
  | .node left val right =>
      left.toList ++ [val] ++ right.toList
```

For our example tree `(.node (.node .leaf 3 .leaf) 5 (.node .leaf 8 .leaf))`:

$$\texttt{toList} \Rightarrow [3, 5, 8]$$

If the tree is a valid BST, `toList` returns a sorted list. But right now nothing *enforces* that.

Insertion

```
@[simp] def Tree.insert [Ord α] (t : Tree α) (el : α) : Tree α :=
  match t with
  | .leaf => .node .leaf el .leaf
  | .node left cur right =>
    match compare el cur with
    | .lt => .node (left.insert el) cur right
    | .eq => .node left cur right
    | .gt => .node left cur (right.insert el)
```

- `[Ord α]` — requires that α has a comparison function
- `@[simp]` — registers this definition for the `simp` tactic (we'll need this later)
- Standard BST insertion: go left if smaller, right if larger, replace if equal

We have a binary tree with `size`, `toList`, and `insert`.

What's missing?

Nothing stops us from building a "BST" where 99 is in the left subtree of 1. The type `Tree` α makes no ordering promises.

Next: Define what "correct" means, then *prove* our code satisfies it.

Part 2: Extrinsic Verification

The Extrinsic Approach

Write the code first. Define correctness second. Prove it third.

Three steps:

`ForAll P t` — “every value in tree `t` satisfies predicate `P`”

`BST t` — the full BST invariant, built on top of `ForAll`

Theorems — `insert` preserves both `ForAll` and `BST`

Step 1: ForAll

```
@[simp] def Tree.ForAll (P : α → Prop) : Tree α → Prop
  | .leaf        => True
  | .node l v r => P v ∧ l.ForAll P ∧ r.ForAll P
```

- A leaf trivially satisfies any predicate (there's nothing to check)
- A node satisfies *P* when:
    - the value *v* satisfies *P*, **and**
    - every value in the left subtree satisfies *P*, **and**
    - every value in the right subtree satisfies *P*

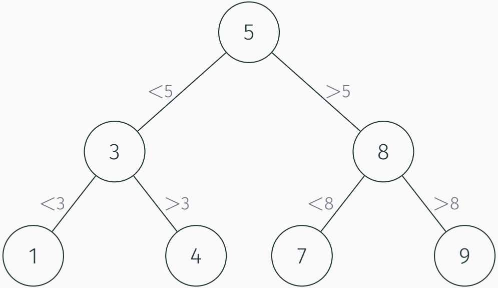
Step 2: The BST Invariant

```
@[simp] def Tree.BST [Ord α] : Tree α → Prop
  | .leaf       => True
  | .node l v r =>
    l.ForAll (fun x => compare x v = .lt) ∧
    r.ForAll (fun x => compare x v = .gt) ∧
    l.BST ∧
    r.BST
```

Four conjuncts for every node:

1. Everything in the left subtree is **less than** the root
2. Everything in the right subtree is **greater than** the root
3. The left subtree is itself a BST
4. The right subtree is itself a BST

Visualising BST



BST checks these ordering constraints at **every** level, recursively.

Step 3a: ForAll is Preserved by Insert

```
@[simp] theorem Tree.ForAll_insert [Ord α]
    {P : α → Prop} {t : Tree α} {x : α}
    (hx : P x) (ht : t.ForAll P)
    : (t.insert x).ForAll P := by
  match t with
  | .leaf => simp_all
  | .node left val right =>
    simp_all
    match compare x val with
    | .lt => simp_all; exact ForAll_insert hx ht.2.1
    | .gt => simp_all; exact ForAll_insert hx ht.2.2
    | .eq => simp_all
```

Reading the ForAll Proof

What does this theorem say in English?

If x satisfies P, and every element already in the tree satisfies P, then every element in the tree after inserting x still satisfies P.

Proof strategy:

- **Base case** (`leaf`): inserting into an empty tree gives `node leaf x leaf` — the only value is x, and we know `P x`.
- **Recursive case** (`node`): we branch on which subtree x goes into, then apply the theorem recursively on that subtree.
- `simp_all` unfolds our `@[simp]` definitions and simplifies automatically.

Step 3b: BST is Preserved by Insert

```
@[simp] theorem Tree.BST_insert [Ord α]
    {t : Tree α} {x : α}
    (h : t.BST) : (t.insert x).BST := by
  match t with
  | .leaf => simp_all
  | .node left val right =>
    simp_all
    obtain ⟨h1, h2, h3, h4⟩ := h
    match hc : compare x val with
    | .lt => simp_all; exact Tree.BST_insert h3
    | .gt => simp_all; exact Tree.BST_insert h4
    | .eq => simp_all
```

Reading the BST Proof

What does this theorem say in English?

If a tree is a valid BST, then inserting any element produces a valid BST.

Key move: `obtain ⟨h1, h2, h3, h4⟩ := h` destructures the four-part BST hypothesis:

- `h1`: left subtree values are less than root
- `h2`: right subtree values are greater than root
- `h3`: left subtree is a BST
- `h4`: right subtree is a BST

Then we case-split on `compare x val` and recurse into the appropriate subtree.

A `structure` bundles several fields into one type:

```
structure Point where
  x : Float
  y : Float
```

You create a value with angle brackets and access fields by name:

```
def origin : Point := ⟨0.0, 0.0⟩

#check origin.x   -- Float
```

Fields can be *data* or *proofs* — Lean doesn't distinguish. A proof is just a value whose type is a `Prop`.

The Extrinsic Wrapper

We want to bundle a tree together with a proof that it's a BST. We *could* write a structure by hand:

```
structure BSTree' (α : Type) [Ord α] where
  val      : Tree α
  property : val.BST
```

Lean has built-in shorthand for exactly this pattern — a **subtype**:

```
def BSTree (α : Type) [Ord α] :=
  { t : Tree α // t.BST }
```

These are the same thing. The subtype gives us `.val` (the tree) and `.property` (the proof), just like the structure above.

BSTree Operations

```
def BSTree.empty [Ord α] : BSTree α :=
  ⟨.leaf, trivial⟩

def BSTree.insert [Ord α] (x : α)
    (t : BSTree α) : BSTree α :=
  ⟨t.val.insert x, Tree.BST_insert t.property⟩

def BSTree.toList [Ord α] (t : BSTree α) : List α :=
  t.val.toList
```

- `t.val` — the underlying tree
- `t.property` — the proof that it's a BST
- `Tree.BST_insert t.property` — applies our theorem to get the new proof

The caller never sees a proof obligation. It's handled internally.

What we built	Purpose
`Tree.ForAll`	Predicate: "every value satisfies P"
`Tree.BST`	Predicate: "this is a valid BST"
`ForAll_insert`	Theorem: insert preserves ForAll
`BST_insert`	Theorem: insert preserves BST
`BSTree`	Wrapper: tree + proof bundled together

Downside: We wrote the code, then wrote *separate* proofs. If we change the data structure, we must update the proofs too. Can we do better?

Part 3: Intrinsic Verification

What if the *type itself* made it
impossible to build an invalid tree?

Instead of proving correctness *after the fact*, we design a type where:

- Every value that's *constructible* is automatically a valid BST
- There are no proofs to write separately — the type checker does the work

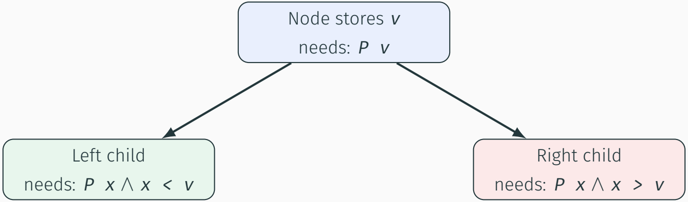
BTree — A Tree Indexed by a Predicate

```
inductive BTree (α : Type) [Ord α]
    : (α → Prop) → Type where
  | leaf : BTree α P
  | node : (v : α) → P v
    → BTree α (fun x => P x ∧ compare x v = .lt)
    → BTree α (fun x => P x ∧ compare x v = .gt)
    → BTree α P
```

The type `BTree` α `P` means: *“a BST where every stored value satisfies P.”*

How the Predicate Strengthens

When we build a ***node*** with value v:



Each level **adds** an ordering constraint. By the time you reach a leaf, the predicate has accumulated *all* the ordering requirements from every ancestor.

SortedTree

```
abbrev SortedTree (α : Type) [Ord α] :=
  BTree α (fun _ => True)
```

We start with the weakest possible predicate: “anything goes.”

The *only* constraints that accumulate are the `compare x v = .lt` / `.gt` conditions from each node — which is exactly the BST invariant.

No separate *BST* proposition. No separate proofs. The types do it all.

Intrinsic Insert

```
def BTree.insert [Ord α] (x : α) (hx : P x) :
    BTree α P → BTree α P
  | .leaf => .node x hx .leaf .leaf
  | .node v hv left right =>
    match hc : compare x v with
    | .lt => .node v hv (left.insert x ⟨hx, hc⟩) right
    | .eq => .node v hv left right
    | .gt => .node v hv left (right.insert x ⟨hx, hc⟩)
```

Why Does Insert Require a Proof?

`insert` takes not just a value *x*, but also *`hx : P x`* — proof that *x* belongs.

For **`SortedTree`**: *P* is *`fun _ => True`*, so the proof is just *`trivial`*. You never think about it.

But imagine a tree of positive numbers: *P* is *`fun x => x > 0`*.

```
-- This would NOT compile:
tree.insert (-3) (proof_that_neg3_gt_0)
-- ↑ You can't produce this proof, so you can't insert -3.
```

The type system rejects bad data *at compile time*.

The Key Move: ⟨`hx, hc`⟩

When inserting x into the left subtree (where `compare x v = .lt`):

```
| .lt => .node v hv (left.insert x ⟨hx, hc⟩) right
```

The left subtree has type `BTree` α `(fun x => P x ∧ compare x v = .lt)`.

So to insert, we need a proof of `P x ∧ compare x v = .lt`.

- *hx* — we already have `P x` (it was passed in)
- *hc* — we just pattern-matched on `compare x v` and got `.lt`

The pair ⟨*hx*, *hc*⟩ is exactly the proof we need. No `theorem` required.

Extrinsic vs. Intrinsic — Side by Side

Extrinsic

- Define *`Tree`* (no invariant)
- Define *`BST`* as a *`Prop`*
- Prove *`BST_insert`* theorem
- Bundle into *`BSTree`* subtype
- Proofs are *separate artifacts*

Intrinsic

- Define *`BTree`* (invariant in the type)
- No separate *`BST`* definition
- No separate theorem
- Just write *`insert`*
- Proofs are *part of the code*

Trade-off: Intrinsic types are harder to *design* but easier to *use*. Extrinsic proofs are easier to *start* but accumulate maintenance burden.

When to Use Which?

	Extrinsic	Intrinsic
Good for	Verifying existing code	Designing from scratch
Refactoring	Must update proofs	Types guide changes
Complexity	Proofs can get long	Types can get complex
Flexibility	Can add properties later	Properties fixed at design time

In practice, most Lean projects use a mix of both styles.

Summary

What We Built Today

- Plain `Tree` — recursive data structure with `insert`, `size`, `toList`
- Extrinsic verification
  - `ForAll` and `BST` as separate propositions
  - Proved `insert` preserves both
  - Bundled into `BSTree` subtype
- Intrinsic verification
  - `BTree` indexed by a predicate
  - Ordering invariants accumulate in the type
  - No separate proofs needed

- Types can encode invariants, not just data shapes
- Extrinsic: prove it correct after writing it — good for retrofitting proofs onto existing code
- Intrinsic: make invalid states unrepresentable — good for new designs where you want guarantees baked in from the start
- `@[simp]` + `simp_all` is a powerful combination for recursive proofs over algebraic data types
- Lean's type system is expressive enough for both approaches in the same language

Questions?