Formal verification of 2-3 Trees in Coq

Horia-Matei Cornea

July 17, 2024

1 Introduction

1.1 Motivation

This thesis explores the expanding field of formal verification in computer science. Formal verification involves creating rigorous mathematical proofs for executable code, ensuring that our specified models will always behave as intended under given constraints [2]. The focus of this work is on 2-3 trees, a type of search tree.

There are several reasons for choosing 2-3 trees. Firstly, trees can be defined recursively, as can the operations that act on them. This recursive nature will be beneficial for proving properties, especially given the limited experience with proof assistants and formal verification. Secondly, trees are fundamental to computer science and are powerful across its various fields. Gaining a deeper understanding of such structures will likely be advantageous in the future.

1.2 What is Coq?

Coq is an interactive theorem prover with a history that began in 1984 with the implementation of the Calculus of Constructions at INRIA-Rocquencourt by Thierry Coquand and Gérard Huet. In 1991, Christine Paulin extended it to include the Calculus of Inductive Constructions [4]. Over the years, more than 200 contributors have helped develop Coq, which is currently maintained by the Coq Team. Detailed information about the contributors and system changes can be found in the Coq Reference Manual [5].

Coq implements a program specification and mathematical higher-level language called Gallina, based on the Calculus of Inductive Constructions. This formal language combines higher-order logic with a richly-typed functional programming language. Coq allows users to define functions and predicates, state mathematical theorems and software specifications, develop formal proofs interactively, and machine-check these proofs using a small certification kernel. An important feature is that certified programs can be extracted to languages such as OCaml, Haskell, or Scheme.

1.3 Problem statement

This thesis aims to provide a formal verification of 2-3 trees. Secondarily, the goal is to gain insight into formal verification, particularly with Coq, the proof assistant chosen for this project.

What does it mean to formally verify a data structure? Logical reasoning is applied to prove or disprove the correctness of an algorithm that can be performed on the data structure, with respect to a functional specification or property. In this project, we focused on two approaches.

Firstly, we consider invariant maintenance, which ensures that all predefined properties of the data structure are preserved throughout its operations. In this case, for operations that modify the tree, a proof would show that under any allowed manipulation of a 2-3 tree, the resulting structure is still a valid 2-3 tree.

Secondly, we look at algebraic specifications, which are meant to define the behavior of the operations using algebraic laws and equations. To provide a comparison, associativity and commutativity are examples of algebraic specifications for addition. For 2-3 trees, algebraic specifications would entail analyzing how the operations of the data structure interact with each other. Algebraic specifications may be formulated using either generic input or specific input.

To achieve our goals, a specification of the tree and its functions will be produced using Coq's specification language, Gallina. Once a working specification is completed, the properties that our data structure must follow will be identified and expressed as predicates. Through the application of logical rules, these predicates will be used to first define and then prove theorems regarding invariant maintenance and algebraic specifications.

1.4 Outline

The following chapters in this document will explore the implementation and proving process. Starting with the specification of 2-3 trees, the necessary foundation and structure of the trees will be established. Following this, the operations on 2-3 trees will be examined in detail, including both their specification and the proving process. Acknowledgments will then be given to the tools and methodologies that facilitated this research and development. Next, potential improvements and future work will be discussed, outlining possible expansions and enhancements to the conducted research. Finally, the thesis will conclude with a summary of the work done.

2 2-3 Trees

2.1 What are 2-3 trees?

The purpose of this research project is to formally represent the 2-3 tree data structure and also verify its specific operations in the Coq environment. A 2-3 tree is a B-tree of order 3, and like the B-tree, AVL, or Red-Black tree, it is a self-balancing tree data structure. In 2-3 trees, there are two types of nodes: 2-Nodes, which have one data value and two children, and 3-Nodes, which have 2 data values and three children.

Because this is a search tree, the stored values have to be sorted. Similarly to binary search trees, for a 2-Node containing a single value `x`, all values in the left subtree are smaller than `x`, and all values in the right subtree are greater than `x`. 3-Nodes, which contain two values, follow a similar principle: the two values are sorted such that the left value is smaller than the right value. The left and right children of large nodes follow the same principle as small nodes, with each side corresponding to its respective value. The middle child of a large node contains only values between the left value and the right value. In Figure 1 there is an example for each kind of node, but also an example of a 2-3 tree composed of multiple such nodes.

The properties of 2-3 trees are:

1. Every node of a 2-3 tree is either a 2-Node or a 3-Node.
2. A 2-Node and a 3-Node must have 2 and 3 children respectively.
3. The items are sorted.
4. The tree is balanced, so every path from the root to the leaves has the same length

Also, it is important to mention that for this project, the tree will contain values maintaining a strict total order, therefore the trees will not contain duplicate values.

2.2 Type definition

Describing a 2-3 tree is relatively simple. It requires three types of nodes: an empty node, for leaves and empty trees, a small node, and a large node. Similar to simple binary trees, the leaf node contains no values.

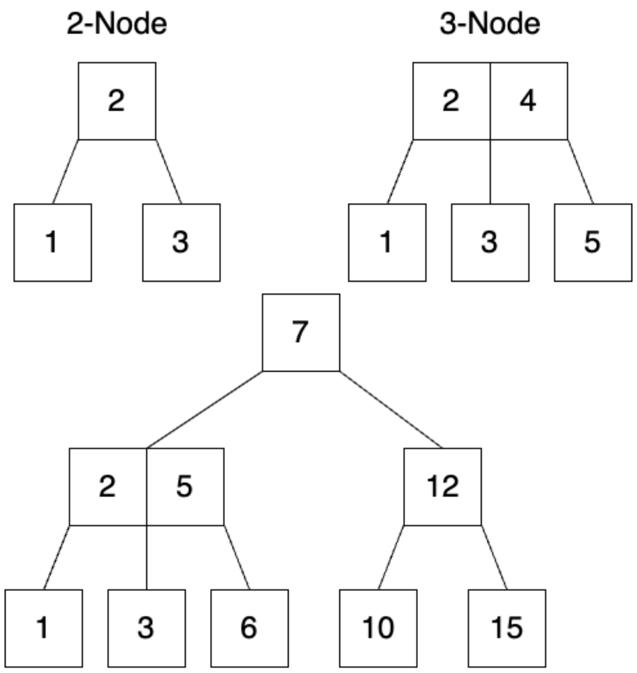

Figure 1: Examples of a 2-Node, a 3-Node, and a 2-3 Tree

The small node holds one value and has two children. The main distinction lies in the large node, which contains two values and has three children.

```
Inductive TTtree : Type :=
  | Empty
  | Node2 (l : TTtree) (v : nat) (r : TTtree)
  | Node3 (l : TTtree) (v1 : nat) (m : TTtree) (v2 : nat) (r : TTtree)
.
```

Listing 1: 2-3 Tree type definition

The difference in the number of constructors will influence the proving process, particularly when using structural induction. When induction is applied to an appropriate theorem goal, it generates two types of sub-goals: a base case and one or more inductive hypotheses. In this scenario, there will be two inductive steps due to the presence of two distinct constructors defined recursively.

To be more concise in later chapters, we need abbreviations for the components of the tree. For a 2-Node we will use `v` for the data value and `l` and `r` for the children. For a 3-Node we will call the 2 data values `v1` and `v2`, and `l`, `m` and `r` for the children.

2.3 Invariants

Invariant maintenance refers to ensuring that specific properties of the data structure remain true before and after any operation is performed on it. Therefore, to guarantee invariant maintenance for 2-3 Trees, we have to look at which properties must be preserved.

Since we have a functional specification, properties 1 and 2 are already satisfied. This happens because the constructors ensure that there are only 2 kinds of nodes, the 2-Node, and the 3-Node. Furthermore, a 2-Node will have one data value and 2 children, and a 3-Node will have 2 data values and 3 children. These rules are built into the type definition and cannot be violated. Thus, the only invariants we need to define are for properties 3 and 4.

Property 3 is concerned with the orderedness of 2-3 Trees. We will use the invariant in Listing 2 for it.

```
Fixpoint ForallTTtree (P : nat -> Prop) (t : TTtree) : Prop :=
  match t with
  | Empty => True
  | Node2 l v r => P v /\ ForallTTtree P l /\ ForallTTtree P r
  | Node3 l v1 m v2 r => P v1 /\ P v2 /\ ForallTTtree P l /\ ForallTTtree P m /\
      ForallTTtree P r
  end
.

Inductive ordered : TTtree -> Prop :=
  | orderedEmpty : ordered Empty
  | orderedNode2 : forall l v r,
      ForallTTtree (fun x => x < v) l ->
      ForallTTtree (fun x => x > v) r ->
      ordered l ->
      ordered r ->
      ordered (Node2 l v r)
  | orderedNode3 : forall l v1 m v2 r,
      v1 < v2 ->
      ForallTTtree (fun x => x < v1) l ->
      ForallTTtree (fun x => x > v1 /\ x < v2) m ->
      ForallTTtree (fun x => x > v2) r ->
      ordered l ->
      ordered m ->
      ordered r ->
      ordered (Node3 l v1 m v2 r)
.
```

Listing 2: ordered invariant

The function ForallTTtree checks if a property P holds for all values of a 2-3 tree. Then, this function is used in the ordered predicate, describing when is each type of node ordered. An empty 2-3 tree node is always ordered. A 2-Node is ordered if: every value in the `l` is smaller than `v`, every value in `r` is greater than `v`, and `l` and `r` are also ordered. A 3-Node is ordered if: `v1` and `v2` are ordered (`v1 < v2`), every value in `l` is lower than `v1`, every value in `m` is in between `v1` and `v2`, every value in `r` is greater than `v2`, and if `l`, `m` and `r` are ordered.

Property 4 is concerned with a 2-3 tree being balanced. For this property, we defined the balanced invariant, which can be seen in Listing 3.

```
Fixpoint height (t : TTtree) : nat :=
  match t with
  | Empty => 0
  | Node2 l _ r => 1 + max (height l) (height r)
  | Node3 l _ m _ r => 1 + max (height l) (max (height m) (height r))
  end
.

Inductive balanced : TTtree -> Prop :=
  | balancedEmpty : balanced Empty
  | balancedNode2 : forall l r v,
      balanced l ->
      balanced r ->
      height l = height r ->
      balanced (Node2 l v r)
  | balancedNode3 : forall l m r v1 v2,
      balanced l ->
      balanced m ->
      balanced r ->
      height l = height m ->
      height m = height r ->
      balanced (Node3 l v1 m v2 r)
```

.

Listing 3: balanced invariant

To verify if a tree is balanced, we first need the height function, which calculates the height of a 2-3 tree by adding 1 to the maximum height of a node's children. With the height function we establish the balanced predicate, similarly to the ordered invariant. We can say that an empty node is always balanced. A 2-Node is balanced if the height of `l` and `r` are the same, and if `l` and `r` are balanced themselves as 2-3 trees. A 3-Node is balanced if the height of `l`, `m` and `r` is the same, and if `l`, `m` and `r` are balanced.

Having these 2 invariants, we can now write theorems to prove that the operations of 2-3 trees preserve them.

3 Operations of 2-3 Trees

3.1 Lookup

3.1.1 Implementation

To determine whether an element is part of a 2-3 tree, the process is very similar to that of a standard binary search tree. In fact, it is identical when considering only 2-Nodes and leaf nodes. The difference arises when dealing with 3-Nodes, though even in this situation, a straightforward adaptation stands. Instead of comparing the element to just one of the node's values, we now compare it to two, resulting in three cases. The first case is when the element is smaller than the left value of the 3-Node. The second case corresponds to the element being larger than the right value of the 3-Node, and the last case occurs when the element falls between the 3-Node's values. Depending on the case, the lookup operation is called recursively on the appropriate subtree of the 3-Node. If the element is equal to either of the 3-Node's values, there is no need to look any further. An example of how the lookup operation works can be seen in Figure 2, and the lookup function itself can be seen in Listing 4.

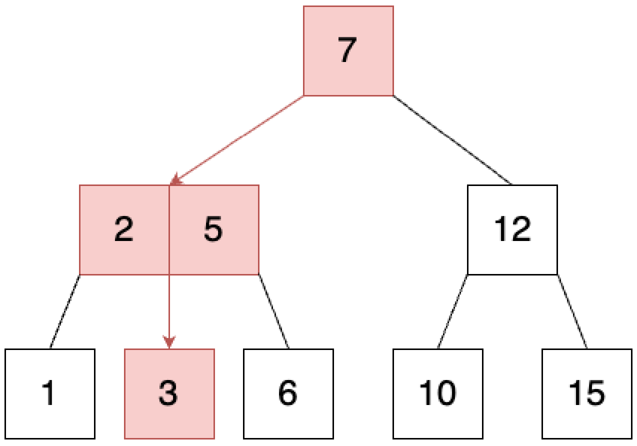

Figure 2: Lookup value 3 on a 2-3 tree

```
Fixpoint lookup (x : nat) (t : TTtree) : bool :=
  match t with
  | Empty => false
  | Node2 l v r =>
      if x =? v then true
      else if x <? v then lookup x l
      else lookup x r
  | Node3 l v1 m v2 r =>
      if (x =? v1) || (x =? v2) then true
      else if x <? v1 then lookup x l
      else if v2 <? x then lookup x r
      else lookup x m
  end
.
```

Listing 4: lookup function

3.1.2 Proofs

The lookup function is an ideal starting point for our proving process because it does not modify the tree. Consequently, the invariants are always preserved during this operation. Additionally, while the lookup function is used in all of our algebraic specifications, these specifications also include the insertion function, therefore we will address them in the next section.

Nevertheless, this is a good moment to display how proofs work. For example, we have a very simple proof showing that the lookup operation on an empty 2-3 tree always returns false. This can be seen in Listing 5.

```
Theorem lookupEmpty : forall x : nat, lookup x Empty = false.
Proof.
  intros x.
  simpl.
  reflexivity.
Qed.
```

Listing 5: lookup on empty tree

In addition to this, for most of the main proofs, including those related to invariant maintenance, we required some subproofs. Firstly, we shall examine the simplest proofs that did not require any additional subproofs.

```
Lemma ForallTTtreeSplit : forall (value1 value2 : nat) (tree : TTtree),
  ForallTTtree (fun x : nat => x > value1 /\ x < value2) tree <->
  ForallTTtree (fun x : nat => x > value1) tree /\ ForallTTtree (fun x : nat => x < value2)
      tree.
Proof.
  intros value1 value2 tree.
  split.
  - (* -> direction *)
    induction tree as [| l IHl v r IHr | l IHl v1 m IHm v2 r IHr].

    + (* Case: tree = Empty *)

    + (* Case: tree = Node2 l v r *)

    + (* Case: tree = Node3 l v1 m v2 r *)

  - (* <- direction *)
    induction tree as [| l IHl v r IHr | l IHl v1 m IHm v2 r IHr].

    + (* Case: tree = Empty *)

    + (* Case: tree = Node2 l v r *)

    + (* Case: tree = Node3 l v1 m v2 r *)
Qed.

Lemma ForallTTtreeTrans : forall (P Q : nat -> Prop) (t : TTtree),
  ForallTTtree P t ->
  (forall x, P x -> Q x) ->
  ForallTTtree Q t.
Proof.
  intros P Q t HForall HPQ.
  induction t as [| l IHl v r IHr | l IHl v1 m IHm v2 r IHr]; simpl in *.
  - (* Case: t = Empty *)

  - (* Case: t = Node2 l v r *)

  - (* Case: t = Node3 l v1 m v2 r *)
Qed.

Lemma ForallTTtreeLookup : forall P t x,
  ForallTTtree P t ->
  lookup x t = true ->
  P x.
Proof.
  intros P t x HForall HLookup.
  induction t as [| l IHl value r IHr | l IHl value1 m IHm value2 r IHr].

  -- (* Case: t = Empty *)

  -- (* Case: t = Node2 l value r *)

  -- (* Case: t = Node3 l value1 m value2 r *)
Qed.
```

Listing 6: auxiliary lemmas

In Listing 6, three auxiliary lemmas can be seen. For readability, only the most important steps of the proofs have been left. The first proof, ForallTTtreeSplit, states that for any two natural numbers `value1` and `value2` and any 2-3 tree `tree`, the predicate `ForallTTtree (fun x => x > value1 / \ x < value2) tree` holds if and only if both `ForallTTtree (fun  x => x > value1) tree` and `ForallTTtree (fun x => x < value2) tree` hold. In other words, checking that all elements in the tree are between value1 and value2 is equivalent to separately checking that all elements are greater than value1 and that all elements are less than value2. This proof is powerful for splitting the ordered invariant in the case of the middle child of a 3-Node.

The proof steps are the following: first, we split the main goal into two goals, which are the two directions of the "if and only if" logical equivalence. For both goals, we apply structural induction on the tree. As we previously mentioned, it is important to notice that we have one base case, where the tree is empty, and two inductive steps, corresponding to the node types of 2-3 trees.

While this is the only proof where we have the `<->` relation, the structural induction will be used in almost all our proofs, including the other 2 lemmas we have in this listing. The second lemma, ForallTTtreeTrans, states that if a property P holds for all elements of a tree t, and if P implies another property Q, then Q also holds for all elements of the tree t. The third lemma, ForallTTtreeLookup, asserts that if a property P holds for all elements in a tree t, and if an element x is found in the tree, then the property P must hold for x.

Using these simple subproofs, we can now develop more complicated lemmas. For instance, in the invariant maintenance proofs, we will need to demonstrate that if a value is not present in a node of a tree, then it is also not present in any of that node's children. These lemmas can be seen in Listing 7. The two lemmas are very similar. Lemma lookupNode2False states that if a 2-Node is ordered and the lookup of x in this node is false, then x cannot be found in `l`, `r`, and x is not equal to the value at the node. Then, lemma lookupNode3False states that if a 3-Node is ordered and the lookup of x in this node is false, then x cannot be found in `l`, `m`, `r`, and x is not equal to the values `v1` and `v2`.

The proofs for these lemmas are very similar, and they use something that we did not have in the previous lemmas. They apply case analysis to compare the lookup value `x` to the values in the nodes. For instance, for the lookupNode2False lemma, we have the following cases:

1. `x` is equal to `v`
2. `x` is not equal to `v` and `x` is smaller than `v`
3. `x` is not equal to `v` and `x` is greater than `v`

This case analysis based on the values is very much caused by the fact that we know that the tree is ordered, and that gives us a lot of hypotheses to work with regarding the values of the node itself and how they can relate to the lookup value `x`.

```
Lemma lookupNode2False : forall (x value : nat) (l r : TTtree),
  ordered (Node2 l value r) ->
  lookup x (Node2 l value r) = false ->
  lookup x l = false /\ lookup x r = false /\ x <> value.
Proof.
  intros x value l r Hordered Hlookup.
  simpl in Hlookup.
  bdestruct (x =? value) as Heq.
  - (* Case: x = value *)

  - (* Case: x <> value *)
    bdestruct (x <? value) as Hlt.
    + (* Case: x < value *)

    + (* Case: x > value *)
Qed.

Lemma lookupNode3False : forall (x v1 v2 : nat) (l m r : TTtree),
  ordered (Node3 l v1 m v2 r) ->
  lookup x (Node3 l v1 m v2 r) = false ->
  lookup x l = false /\ lookup x m = false /\ lookup x r = false /\ x <> v1 /\ x <> v2.
Proof.
  intros x v1 v2 l m r Hordered Hlookup.
  simpl in Hlookup.
  bdestruct (x =? v1) as Heq1.
  - (* x = v1 *)

  - (* x <> v1 *)
    bdestruct (x =? v2) as Heq2.
    + (* x = v2 *)

    + (* x <> v1 *)
      bdestruct (x <? v1) as Hlt1.
      * (* x < v1 *)

      * (* x > v1 *)
        bdestruct (v2 <? x) as Hlt2.
        -- (* x > v2 *)

        -- (* x < v2 *)
Qed.
```

Listing 7: auxiliary lemmas

3.2 Insertion

3.2.1 Implementation

The insertion procedure introduces new challenges, both in the specification and proving. This is because, aside from a few specific cases, the tree often needs to be rebalanced multiple times during an insertion. An example of this issue can be seen in Figure 3. The primary strategy for the insert operation involves traveling down the tree until the correct spot for insertion is found at the terminal nodes. If the value can be absorbed into the terminal node (if a small node is encountered, it can be turned into a large node), the operation is complete. If the node cannot be absorbed, it needs to be pushed back up the tree to find an appropriate spot.

To manage this change in traversal direction, an extra data type with two configurations is needed. One configuration, Normal, simply wraps a valid 2-3 node, used during the backtracking phase once the value to be added has been absorbed. The other configuration, KickUp, handles cases where the value is being pushed up. To rebalance the tree, the KickUp stores the value to be added along with two sub-trees, corresponding to the children of a small node with the value at that level. This data type can be seen in Listing 8.

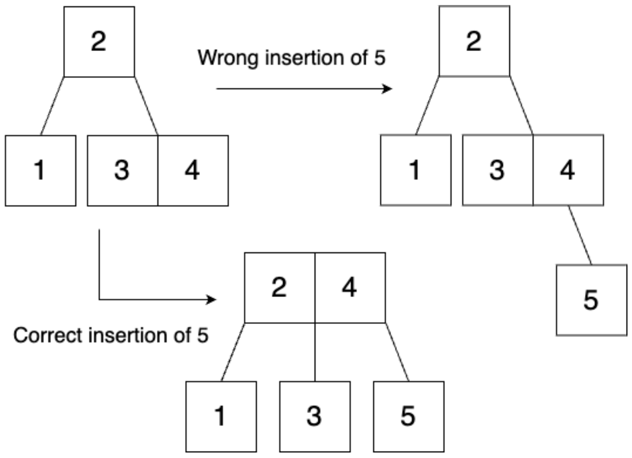

Figure 3: Insertion operation

```
Inductive InsertionTTtree : Type :=
  | Normal (t: TTtree)
  | KickUp (l : TTtree) (v: nat) (r : TTtree)
.
```

Listing 8: InsertionTTtree data type

Because this additional data type is used alongside the already defined 2-3 nodes, it is crucial to ensure that the insert operation always returns a correct tree. To achieve this, a wrapper function is necessary. This function, called upward, manages both valid nodes and kicked-up configurations. Specifically, it takes a 2-3 node as input and returns a kicked-up configuration. The function uses recursion to push the value down the tree until a terminal node is reached, and then the inserted value becomes a KickUp. The next step is to move the KickUp tree up until it is absorbed. This is performed by always looking at the parent of the KickUp. If the parent of the KickUp is a 2-Node, then the KickUp and its parent are converted into a 3-Node, however, if the parent is a 3-Node, then the KickUp and its parent become another KickUp.

A kicked-up configuration is not a valid 2-3 tree, so another function, convertRootKickUp, is needed to convert this configuration once the procedure is complete. Also, since the tree only accepts distinct values, the insertion main function includes a check for the value to be inserted. If the value is already stored in the tree, the function terminates immediately. The complete insertion implementation can be seen in Listing 9.

```
Fixpoint upward (x : nat) (t : TTtree) : InsertionTTtree :=
  match t with
  | Empty => KickUp Empty x Empty
  | Node2 l v r =>
      if x <? v then (* When the KickUp is coming from the left child *)
        match upward x l with
        | Normal t' => Normal (Node2 t' v r)
        | KickUp l' v' r' => Normal (Node3 l' v' r' v r)
        end
      else (* When the KickUp is coming from the right child *)
        match upward x r with
        | Normal t' => Normal (Node2 l v t')
        | KickUp l' v' r' => Normal (Node3 l v l' v' r')
        end
  | Node3 l v1 m v2 r =>
      if x <? v1 then (* When the KickUp is coming from the left child *)
        match upward x l with
        | Normal t' => Normal (Node3 t' v1 m v2 r)
        | KickUp l' v' r' => KickUp (Node2 l' v' r') v1 (Node2 m v2 r)
        end
      else if v2 <? x then (* When the KickUp is coming from the right child *)
        match upward x r with
        | Normal t' => Normal (Node3 l v1 m v2 t')
        | KickUp l' v' r' => KickUp (Node2 l v1 m) v2 (Node2 l' v' r')
        end
      else (* When the KickUp is coming from the middle child *)
        match upward x m with
        | Normal t' => Normal (Node3 l v1 t' v2 r)
        | KickUp l' v' r' => KickUp (Node2 l v1 l') v' (Node2 r' v2 r)
        end
  end
.

Definition convertRootKickUp (tempTTtree : InsertionTTtree) : TTtree :=
  match tempTTtree with
  | Normal t => t
  | KickUp l v r => Node2 l v r
  end
.

Definition insert (x : nat) (t : TTtree) : TTtree :=
  match lookup x t with
  | true => t (* If the element is already in the tree *)
  | false => convertRootKickUp (upward x t) (* If not, we insert it in the tree *)
  end
.
```

Listing 9: Insertion function

3.2.2 Proofs

As we specified in the problem statement, for this project we considered two kinds of proofs, invariant maintenance and algebraic proofs. For invariant maintenance, we have to show that if we had an ordered or balanced tree, the tree would still be balanced after an insertion operation. But before we look at the proofs, we have to show another auxiliary lemma that helped us with invariant maintenance. This lemma can be seen in Listing 10. In simpler terms, this lemma ensures that the property `P` is preserved throughout the tree, even after modifying the tree using the specified functions. If `P` is true for the initial value `v` and all elements in the original tree `t`, it will still be true for all elements in the new tree resulting from the operations upward and convertRootKickUp. The most important thing to notice is the proof structure of this lemma, which is going to occur in almost all proofs for insertion. The proof uses structural induction and case analysis like the previous proofs, but it also does case analysis to look at what type of configuration we have for the InsertionTTtree that is pushed up the tree, Normal or KickUp.

```
Lemma ForallTTtreeUpward : forall (P : nat -> Prop) (v : nat) (t : TTtree),
  P v ->
  ForallTTtree P t ->
  ForallTTtree P (convertRootKickUp(upward v t)).
Proof.
  intros P v t Hp HForall.
  induction t as [| l IHl value r IHr | l IHl value1 m IHm value2 r IHr].

  - (* Case: t = Empty *)

  - (* Case: t = Node2 l value r *)
    bdestruct (v<?value) as Hlessvalue.

    * (* v < value *)
      destruct (upward v l) eqn:HupL.
      -- (* upward v l = Normal t *)

      -- (* upward v l = KickUp l0 v0 r0 *)

    * (* v >= value *)
      destruct (upward v r) eqn:HupR.
      -- (* upward v r = Normal t *)

      -- (* upward v r = KickUp l0 v0 r0 *)
  - (* Case: t = Node3 l value1 m value2 r *)
    bdestruct (v<?value1) as Hvalue1.

    * (* v < value1 *)
      destruct (upward v l) eqn:HupL.
      -- (* upward v l = Normal t *)

      -- (* upward v l = KickUp l0 v0 r0 *)

    * (* v >= value1 *)
      bdestruct (value2<?v) as Hvalue2.
      -- (* v > value2 *)
         destruct (upward v r) eqn:HupL.
         ** (* upward v r = Normal t *)

         ** (* upward v r = KickUp l0 v0 r0 *)

      -- (* v <= value2 *)
         destruct (upward v m) eqn:HupM.
         ** (* upward v m = Normal t *)

         ** (* upward v m = KickUp l0 v0 r0 *)
Qed.
```

Listing 10: ForallTTtreeUpward lemma

In Listing 11 can be found the proof for the ordered invariant. Knowing that any 2-3 tree called `tree` is ordered, we must show that "tree" is still ordered after we insert any element `v`. We start the proof by checking whether `v` is or isn't in the tree already. If it is, then the `tree` does not change at all so the proof is easy, but if `v` is not in the tree, then we call orderedLookupFalseInsert, which is just a long proof that makes use of most of the auxiliary lemmas we had so far, and which has the same structure as the ForallTTtreeUpward lemma.

```
Theorem orderedInsert : forall (tree : TTtree) (v : nat),
    ordered tree -> ordered (insert v tree).
Proof.
  intros tree v Hordered.
  unfold insert.
  destruct (lookup v tree) eqn:Hlookup.
  - (* lookup v tree = true *)
    assumption.
  - (* lookup v tree = false *)
    apply orderedLookupFalseInsert; try assumption.
Qed.
```

Listing 11: ordered invariant maintenance

In Listing 12 we can see the theorems for the balanced invariant. The proofs of these theorems were not added due to the fact that their structure is exactly the same as for the ForallTTtreeUpward lemma. The main theorem for the balanced invariant maintenance is the last lemma in this listing, while the other two are helper lemmas. Both lemmas together help ensure that the upward function, regardless of whether it results in a KickUp or Normal tree, maintains the balance and height properties of the original tree.

The balancedUpwardKickup lemma states that if a given tree is balanced and the result of applying the upward function to a value `v` and this tree is a KickUp containing subtrees `l` and `r` with a new value `v'`, then the heights of the original tree and the subtrees `l` and `r` are equal, and both `l` and `r` are balanced.

The balancedUpwardNormal lemma states that if a given tree is balanced and the result of applying the upward function to a value `v` and this tree is Normal, resulting in a new tree `t`, then the new tree `t` is also balanced, and the height of the original tree is the same as the height of the new tree `t`.

```
Lemma balancedUpwardKickup : forall tree v l v' r,
  balanced tree ->
  upward v tree = KickUp l v' r ->
  height tree = height l /\ height tree = height r /\ balanced l /\ balanced r.
```

```
Lemma balancedUpwardNormal : forall (tree t : TTtree) (v : nat),
  (balanced tree ->
  upward v tree = Normal t ->
  balanced t /\ height tree = height t).

Theorem balancedInsert : forall (t : TTtree) (v : nat),
  balanced t -> balanced (insert v t).
```

Listing 12: balanced invariant maintenance

The algebraic specifications we defined between the lookup and the insert operation can be seen in Listing 13. The lookupInsert theorem states that if you insert a value `x` into a 2-3 tree, then performing a lookup for `x` in the resulting tree will always return true. In other words, after you add `x` to the tree using the insert function, `x` will definitely be found in the tree. The second theorem, lookupInsertNeq, addresses the scenario where a different value `v'` is looked up in the tree after inserting a value `v`. In essence, if you insert a value `v` into the tree and then look for a different value `v'`, the presence or absence of `v'` in the tree remains unchanged. This theorem ensures that inserting a value `v` does not affect the lookup results for other distinct values `v'`.

The last algebraic specification, commutativityInsert, states that for any values `x`, `y`, and `z`, and for any ordered 2-3 tree, the result of looking up `z` after inserting `x` and then `y` into the tree is the same as the result of looking up `z` after inserting `y` and then `x`. The order in which you insert the values `x` and `y` into the tree does not affect the presence or absence of another value `z` in the tree. This theorem guarantees that the insert operations are commutative with respect to the lookup function, provided the tree maintains its ordered property.

```
Theorem lookupInsert : forall (x : nat) (tree : TTtree),
  lookup x (insert x tree) = true.

Theorem lookupInsertNeq : forall (t : TTtree) (v v' : nat),
  ordered t ->
  v <> v' ->
  lookup v' (insert v t) = lookup v' t.

Theorem commutativityInsert : forall x y z tree,
  ordered tree ->
  lookup z (insert x (insert y tree)) = lookup z (insert y (insert x tree)).
```

Listing 13: balanced invariant maintenance

3.3 Deletion

3.3.1 Implementation

The deletion function is the most difficult operation in the 2-3 tree implementation. Similar to the insertion procedure, an additional type, DeletionTTtree, which can be seen in Listing 14, is used to represent either a NormalTree node or a Hole. The algorithm is structurally similar to the insertion process, consisting of both a downward phase and an upward phase. However, unlike insertion, the downward phase of deletion does not always reach the leaves of the tree, as the value to be deleted might reside in an internal node. This scenario is particularly complex and requires additional handling. From now on we will refer to the value to be deleted as `x`.

```
Inductive DeletionTTtree : Type :=
  | NormalTree (t : TTtree)
  | Hole (t : TTtree)
.
```

Listing 14: DeletionTTtree data type

In cases where `x` is found in a terminal node, `x` is replaced with a Hole wrapper, which is then carried up the tree until it can be appropriately placed. This placement can occur either at the root or during the upward traversal of the tree. The principle is very similar to the KickUp. The Hole looks at the parent node, and based on that the Hole is either absorbed and becomes a NormalTree or together with its parent it becomes a new Hole to be pushed up. However, the Hole has to also look at the sibling(s) in order to determine whether it can be absorbed or not.

A significant challenge arises when `x` is located in an internal node, necessitating a swap with its in-order successor (or predecessor). Consequently, the delete function now includes a check for `x`'s position in the tree. If `x` is internal, its successor is deleted, and `x` is replaced with this successor in the final result.

The deletion function can be seen in Listing 15. First, a lookup operation is performed to see whether the value `x` is in the tree or not, if it is not then there is nothing to delete. If `x` is in the tree, then we perform another check to see if `x` is located in an internal node or not. This check is performed by the leaf function. If `x` is a terminal node, we call the upwardDelete function, which does the exact same thing as the upward function from the insertion operation, but is more complex, because the upwardDelete function has to check the sibling(s) as well. After the upwardDelete function, we need to convert the DeletionTTtree back to a 2-3 tree, and we do this using the convertRootHole function. In case `x` is not a terminal node, then we store the value of its successor in `y` using the successor function. It is important to understand that the successor of a value in any tree that does not contain duplicates is always a leaf. Knowing that we can delete `y` just as we did `x` in the first case, and after that, we can replace `x` with `y` using the replace function.

```
Definition delete (x : nat) (t : TTtree) : TTtree :=
  match lookup x t with
  | false => t (* If the value is not in the tree, then we have nothing to delete *)
  | true =>
      match leaf x t with (* Based on whether the value we want to delete is in a terminal
          node or not, we have 2 cases *)
      | true => convertRootHole (upwardDelete x t)
      | false => let y := successor x t in replace x y (convertRootHole (upwardDelete y t))
      end
  end
.
```

Listing 15: Deletion function

3.3.2 Proofs

Due to time constraints, we were unable to complete the proofs for the deletion operation. This omission is primarily caused by the complexity of the deletion process, which involves numerous functions. However, we have left the definitions of the theorems for invariant maintenance and algebraic specifications of deletion as Admitted. This allows those who wish to continue this project to build upon our work and finalize the proofs. The Admitted proofs can be seen in Listing 16.

```
Theorem deleteOrderedTTree : forall x t,
  ordered t ->
  ordered (delete x t).
Proof.
Admitted.

Theorem deleteBalancedTTree : forall x t,
  balanced t ->
  balanced (delete x t).
Proof.
Admitted.

Theorem lookupDelete : forall (x : nat) (tree : TTtree),
  lookup x (delete x tree) = false.
Proof.
Admitted.

Theorem lookupDeleteNeq : forall (t : TTtree) (v v' : nat),
  ordered t ->
  v <> v' ->
  lookup v' (delete v t) = lookup v' t.
Proof.
Admitted.

Theorem commutativityDelete : forall x y z tree,
  ordered tree ->
  lookup z (delete x (delete y tree)) = lookup z (delete y (delete x tree)).
Proof.
Admitted.
```

Listing 16: Deletion unfinished proofs

4 Future Work

Overall, there are several routes for further research. The most obvious direction would be to complete the maintenance of invariants and the algebraic specifications for the delete function. Another interesting addition would be the join function, which merges two trees.

Furthermore, the proofs could be expanded by incorporating model-based specifications. Model-based specifications describe the desired behavior of a system using abstract models that represent the system's state and operations. For 2-3 trees, a model-based specification would involve defining an abstract model of the tree's structure and behavior, such as a list, then specifying how operations like insertion, and deletion affect this model. This approach helps ensure that the implementation meets the desired properties and can be used to formally verify the correctness of the tree's operations.

Lastly, as a longer-term project, this work could serve as a foundation for understanding B-trees. B-trees, with their variable number of values in nodes, present a greater challenge in both specification and proving. Expanding this work to B-trees would enhance its applicability to practical scenarios, as B-trees are more suitable for current machine architectures where a node degree of 3 is insufficient to fully leverage their properties.

5 Conclusion

We would like to acknowledge that our specification followed a document by Turbak[6], which describes a functional pseudo-implementation of 2-3 trees. As a template for the proofs, we referred to the relevant chapters from the Software Foundations series, specifically the chapter on Search Trees[3]. We extend our gratitude to these resources for providing a solid foundation and guidance for our project.

In conclusion, formal verification in Coq was a rewarding experience, thanks to its interactive environment and the use of its specialized command language. This setup made the process of constructing proofs intuitive and engaging. One of the most interesting aspects was understanding that to prove certain properties, it is sometimes necessary to generalize aspects of the specification. This need for generalization often becomes apparent during the proving process, highlighting the depth and intricacy involved in formal verification. Overall, working with Coq not only facilitated rigorous proof development but also deepened our understanding of the underlying theoretical concepts. The 2-3 trees implementation and proofs in Coq can be browsed at the following link [1].

References

[1] `https://github.com/CorneaHoriaMatei/TwoThreeTreesCoq`.

[2] Ram Chandra Bhushan and Dharmendra K Yadav. Verification of Virtual Machine Architecture in a Hypervisor through Model Checking. *Procedia computer science*, 167:67–74, 1 2020.

[3] Benjamin C. Pierce et al. *Software Foundations.* Electronic textbook, 2023. `https://softwarefoundations.cis.upenn.edu/`.

[4] Coq Development Team. About coq. `https://coq.inria.fr/about-coq`, 2023. Accessed: 2024-07-02.

[5] Coq Development Team. Coqide - the coq integrated development environment. `https://coq.inria.fr/doc/V8.19.2/refman/practical-tools/coqide.html`, 2023. Accessed: 2024-07-02.

[6] Franklyn Turbak. A functional pseudo-implementation of 2-3 trees. `https://www.cs.princeton.edu/~dpw/courses/cos326-12/ass/2-3-trees.pdf`, 2005. Accessed: 2024-07-15.