# Formalizing 2-3 trees in Isabelle/HOL

Silvio Bolognani

Department of Computer Science
Vrij Universiteit Amsterdam
Netherlands

9 June 2021

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

This thesis delves into a growing field in computer science, formal verification. Formal verification is dedicated to providing rigorous mathematical proofs of executable code, thus giving the guarantee that our specified models will always behave as intended given certain constraints [1]. Specifically the focus will be on 2-3 trees, a search tree. The reasons for this choice were multiple:

First of all, trees can intuitively be defined recursively and similarly for the operations that operate on them (which is going to come in handy in the proving), so given the lack of experience with proof assistants and formal verification, this kind of tree should bring the right amount of challenge and learning opportunity. Aside from their recursive nature trees are central to computer science permeating all of its fields, so achieving a deeper understanding of such structures will likely come in handy in the future. Therefore the choice of 2-3 trees was the result of a combination of practicality and usefulness.

## 1.2 Problem Statement

The aim of this thesis is to provide a formal proof of 2-3 trees. Secondarily the goal is also to gain insight in formal verification, particularly with Isabelle/HOL, the proof assistant chosen for this project.

What does it mean to verify a data structure? Logical reasoning is applied to a functional specification of such data structure in order to mathematically reason about the operations that can be performed on such data structure.

In this case, for operations that modify the tree, a proof would consist of showing that under any allowed manipulation of a 2-3 tree the result achieved is still a valid 2-3 tree, while for lookup (which has no effect on the tree), correctness of the operation will need to be shown. In order to accomplish this goal, several steps are needed. To start with, a specification of the tree and its functions will have to be produced, using Isabelle's specification language. Once a working specification has been reached the various properties that our data structure has to follow need to be identified and expressed in theorems, such that through the application of logic rules they can be proved correct.

Finally after these theorems are formulated, the proving phase starts.

## 1.3 Outline

The following chapters in the document will explore the specification and proving process in its entirety. Starting from a brief introduction to Isabelle/HOL and 2-3 trees to provide some necessary knowledge before moving into the specification of the tree structure itself, and the invariants that the tree needs to maintain. Once these basics are established, the operations acting on the tree will be examined both in the specification and in the proving process. The thesis will then compare this work to previously conducted research on the same topic and their relevant differences. Closing, potential improvements and expansions to the conducted work will be discussed, together with a brief summary of the work done.

# Chapter 2

# Background

## 2.1 Isabelle/HOL

Isabelle/HOL is a proof assistant, originally developed by researchers at University of Cambridge and Technical University of Munich its foundations are built upon weak type theory (provided by Isabelle) upon which a higher-order-logic framework is built (HOL). Such proof assistants provide a logical framework inside which theorems can be safely proven [2]. This safety is given by the framework provided by Isabelle which only allows the user to take valid steps in the proof process (this does not mean correct steps, just avoids incorrect rule applications). While this structure also brings proof assistants' usefulness also shows when dealing with large and constantly changing proofs and specifications, since dependencies between the components are automatically tracked and eventual issues are flagged. In the following paragraph the main functionalities used in this thesis are explained.

The first and most basic functionality needed is the specification language itself which is needed to model the specification of the tree. For this purpose, Isabelle uses a modified version of the Standard ML language [3] (a functional language), it is to be noted that while the syntax of this language is based on SML, since it needs to interact with other components of Isabelle the syntax is quite modified from pure SML. A nice feature of this language is that it provides different ways of specifying recursive functions, with each way resulting in slightly different proving frameworks. All of the functions will be declared as primrec (primitive recursive) or fun, which have the benefit of automatically looking for a termination order, so it does not have to be manually proven.

For when termination cannot be proved automatically the HOL framework of Isabelle/HOL then will be used to formulate lemmas and theorems about the structure previously specified. When a lemma or theorem is declared Isabelle will enter proof mode. Once prove mode is entered Isabelle will prompt us with the current sub-goals that need to be proved in order to complete the proof. While in prove mode different techniques can be applied in order to simplify the sub-goals or split them into smaller sub-goals.

Isabelle provides users with a gamma of different automated tools, including: rewriting tools, logical FOL provers and access to full-fledged automated theorem provers (through Sledgehammer [4]).

**Isabelle/HOL terminology for future reference**
**simp**: Executes term rewriting on one sub-goal.
**auto**: Runs simp, arithmetical simplification, and some logic, works on all sub-goals.
**blast**: First-order-logic prover, with practical limitations.
**Sledgehammer**: Constructs proofs from external ATPs.

Another useful functionality provided is proof terminators, which end prove mode for theorems with different effects. The oops keyword simply closes prove mode in order to allow the user to continue his work with other proofs, while the sorry keyword allows us to consider the theorem as proven, which in turn means it can be used to prove other theorems allowing for easier forward reasoning. Isabelle/HOL provides plenty of other features, such as code generation, but for brevity's sake, only the more used features were mentioned.

**NB**: Throughout the text Isabelle/HOL will be referred to as Isabelle, this is to be read as Isabelle/HOL.

## 2.2 2-3 Trees

2-3 trees are a type of balanced search tree, more specifically a special case of B-trees (namely a B-tree with order 3). B-trees were originally created by Bayer and McCreight [5] in order to minimize disk access requests and they are identified by a order parameter. As later defined by Knuth [6] this order $d$ indicates the maximum number of children possible for any node, with the minimum being $\lceil d/2 \rceil$. Each node also always has one child more than the number of values it contains.

How is this used to minimize disk accesses? By setting the degree of the tree equal to the disk size. This way the seek time of the disk is reduced to a minimum through locality exploitation. While general B-trees are a bit too complex for the scope of this thesis, 2-3 trees allow to maintain the relevant properties while reducing the complexity. Since only B-trees of order 3 are considered, each node has either 1 value and 2 children or 2 values and 3 children.

Given that this is a search tree the values stored are sorted. For a small node containing value $x$, this means that all values in the left subtree are smaller than $x$ and all values in the right subtree $> x$. Large nodes work in a similar fashion, the 2 values stored inside the node are sorted such that the left value is smaller than the right one. The left and right children of such nodes follow the same principle of the small node (with, for each side the corresponding value) while the middle child will contain only values between lv and rv. Another fundamental property of 2-3 trees (and more generally B-trees) is that they are complete, so for all valid trees the distance from every leaf to the root is the same (i.e. all the leaves are at the same level). This means that every operation that modifies the structure of the tree also has to rebalance it in order to maintain its invariants.

For this thesis, the tree will contain values maintaining a strict total order, that is an asymmetric relation as follows, $x$ and $y$ are values in the tree and $T$ is the set of elements of an arbitrary 2-3 tree:

$$\forall x \in T, count\ x == 1$$

For this specification, a document by Turbak [7] describing a functional psuedo-implementation of 2-3 trees has been followed. It is also worth noting that while this specification deals with nodes that store a single value, this could be replaced with a key-value pairing in order to give the tree more flexibility in usage. An example of a 2-3 tree is given below.

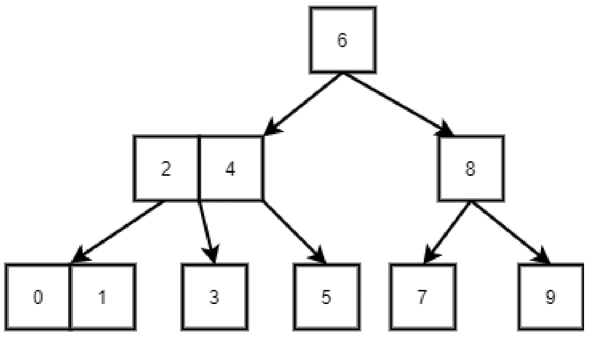

Figure 2.1: Result of inserting naturals from 0 to 9 in an Empty tree

# Chapter 3

# Specification of the tree

## 3.1 Type Definition

Specifying the tree is fairly straightforward an empty leaf type, a small node and a large node are needed. Like for simple binary trees, the leaf will contain no values,and the small node holds one value and two children. The main difference comes from the large node, which contains 2 values and 3 children.

Listing 3.1: Type definition in Isabelle

```
datatype 'a tttree =
          Empty |
          SmallNode "'a tttree" 'a  "'a tttree" |
          LargeNode "'a tttree"  'a  "'a tttree"  'a "'a tttree"
```

This difference in the number of constructors will play a role in the proving process; this is obvious when looking at how the structural induction technique operates. Applying induction to a theorem goal (on which it is applicable) will generate two types of sub-goals, a base case, and one or more inductive hypothesis. In this case, there will be two inductive steps because of the two different constructors that are defined recursively. In order to be more succinct in later chapters some abbreviations for the components of the tree are needed. The left and right subtrees will be called L and R. In the small node the value is shortened to V, while in the large node the values are identified by LV and RV (where L and R stand for the respective positions), and the middle subtree will be referred to as M.

The 2-3 tree datatype is not the only one that will be necessary. To anticipate slightly the next chapter, both the insert and the delete function have a downward and upward phase. In the upward phase the function will be working with a wrapped tree node indicating whether the value carried up has been absorbed or not. Since the value to be absorbed is dependent on the function two more datatypes are needed.

Listing 3.2: Temporary types for deletion and insertion

```
datatype 'a TempDel = Hole "'a tttree" |Nor "'a tttree"

datatype 'a TempIns = Norm "'a tttree" | KickUp "'a tttree" 'a "
    'a tttree"
```

## 3.2 Invariants

In order to be able to formulate theorems from which properties of the system can be proved the invariants need to be identified and specified in Isabelle such that they can be later used in the proofs. As identified by Knuth, B-trees satisfy 5 invariants:

1. Every path from the root to a leaf has the same length.
2. If a node has n children, it contains $n - 1$ *keys*.
3. Every node (except the root and leaves) is at least half full.
4. The elements stored in a given subtree all have keys of ordered between the keys in the parent node on either side of the subtree pointer.
5. The root has at least two children if it is not a leaf.

Since the tree is specified functionally, invariants 2, 3 and 5 are already satisfied. This is because the nodes have a fixed structure, so:

2. For both possible non-leaf nodes the children are always $keys + 1$.
3. The LargeNode construct is equivalent to the order of the tree; therefore it's full, and $2 > \lceil 3/2 \rceil$.
5. If the root is not a leaf it is either a small node or a large node. In both cases, the children are $\geq 2$.

The more obvious invariant is that no matter how the tree is modified, the resulting tree should still operate as a search tree. In other words by doing an in-order traversal of the tree every element should be larger than the predecessor. While it would be possible to write a function that immediately checks whether this in-order traversal of the tree maintains the ordering, it is more practical to instead write a function that from a given tree it returns a list containing the elements of the tree as encountered during an in-order traversal. The benefit of doing this is that Isabelle lists have a comprehensive set of facts already proven about them, which will come in handy in later proofs.

At last, the more complex invariant to prove is that every leaf has to be at the same level. In order to verify this statement, a boolean function is needed. This function intuitively will return true if the tree maintains this equivalent distance from leaves to root and false otherwise. With this new method, it is possible to finally represent the height invariant, which will come in handy on both the lhs and rhs of different theorems.

Listing 3.3: The height invariant function

```
primrec TT :: "'a tttree ⇒ bool" where
"TT Empty = True"|
"TT (SmallNode l _ r) = ((heightTree l = heightTree r) & (TT l)
  & (TT r) )"|
"TT (LargeNode l _ m _ r) = (((heightTree l = heightTree r) & (
   heightTree l = heightTree m)) & (TT l) & (TT m) & (TT r))"
```

# Chapter 4

# Operations on the tree

## 4.1 Lookup

In order to determine whether an element is part of our tree, the process is very similar to a more standard binary search tree, matter of fact it is identical if only looking at the small node and leaf constructors. The difference come when dealing with large nodes, but even with them, a straightforward adaptation jumps to the eye. Since the tree is still sorted it can be adapted to the strategy used for the smaller node by adding a third comparison case that checks for values in between the left and right values.

The lookup function is ideal to start our proving process since the tree is not modified by this operation. This means that invariants maintenance is always maintained. This leaves us to prove the correctness. What does it mean for a querying operation to behave correctly? If the value queried is inside the tree the function should return true otherwise false should be returned. This is equivalent to the question of belonging in a set. Since sets are well developed (plenty of available facts that can be used in our proving) in Isabelle, with these two statements the conclusion for the theorem to prove correctness can be formulated. Now the assumptions needed for proving the theorem must be identified. What premises are needed to prove the equivalence statement? The invariants for completeness are not necessary to prove, because the querying only inspects elements. Then the only premise needed is that the tree is sorted, since the function does not change the tree structure, but only the values stored are examined. Obviously, since complete and sorted trees are a subset of sorted trees, this theorem also holds for complete trees.

Listing 4.1: Theorem for contains' correctness

```
theorem cont_corr_set : "sorted(elements t) ⟹
                         contains a t = (a ∈ set (elements t))"
```

Like most of the proofs present in this thesis, it started with structural induction on the tree. The base case is fairly straightforward and can be directly deduced from lookup function itself. Now various theorem provers were put to use in order to help with simplifying the goals This first proof was mostly to get used to the various automatic solvers provided by Isabelle. The base case like most cases covered in this thesis can easily be taken care of with simp.

Now let's try to see how the different automated tools break down the subgoals. Run-

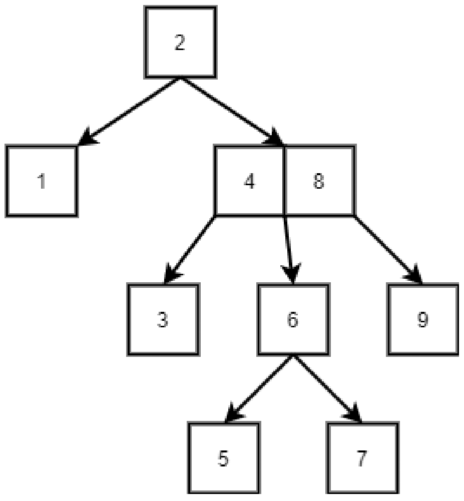

Figure 4.1: Example of uneven tree

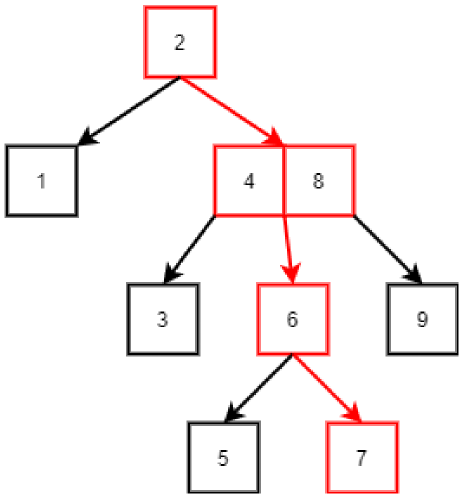

Figure 4.2: Contains run succesfully regardless

ning auto will already break down our 2 subgoals in 22 more specific subgoals. At this point, even running try0, an Isabelle procedure which runs most automated solvers won't find a proof, and both simp and auto are out of their depth. Isabelle though has another tool to find automated proofs, Sledgehammer.

Listing 4.2: Subgoal for a small node pre-simplification

```
(sorted(elements t1) ⇒ contains a t1 = (a ∈ set(elements t1)))⇒
(sorted(elements t2) ⇒ contains a t2 = (a ∈ set(elements t2)))⇒
sorted (elements (SmallNode t1 x2 t2)) ⇒
contains a (SmallNode t1 x2 t2) = (a ∈ set (elements (SmallNode
    t1 x2 t2)))
```

Running Sledgehammer immediately yields no result, so *simp* is run to convert the subgoal in a form that provers might be more familiar with. Now Sledgehammer finds a proof to the first subgoal, looking at the other subgoals it gets fairly clear that also the other goals can likely be solved in the same fashion. Trying to manually apply Sledgehammer to every goal confirms this initial suspicion. Working with this proof is quite abstract, so let's see how it can be improved upon in its legibility and conciseness. Looking at the proofs provided by sledgehammer it's possible to see what lemmas were used in the proof. Looking up these lemmas in the Isabelle libraries is useful in order to get a better idea of how sledgehammer has formulated the proof.

In this case Sledgehammer uses a few different facts coming from the *List_Ins_Del* theory which establishes different facts regarding sorted lists and sets. Inspecting this page the lemmas used by Sledgehammer and a collection of lemmas are displayed. These facts equate a sorted list in set terminology under the name *isin_simps*. Now let's see if this statements can simplify the previously substantial proof. Let's add *isin_simps* as a simplification rule. The resulting sub-goals immediately seems in a more intuitive form.

Listing 4.3: Subgoal for a small node post-simplification.

```
(x2 < a →(a ∈ set(elements t2)) = (a = x2 ∨ a ∈ set (elements
    t1)) ∨ a ∈ set (elements t2)) ∧
(¬ x2 < a → a < x2 → (a ∈ set(elements t1)) = (a = x2 ∨ a ∈ set
(elements t1)∨ a ∈ set (elements t2)))
```

The conclusion is now split into a conjunction of two terms, one term covers the case for which the value queried for is larger than $V$, while the other covers the negation of the first. Now the conclusion of these cases has become more trivial. Depending on the case, the left hand side of the conclusion has become a isin set (element t1/t2) while the right hand side has been simplified by *sorted_lems*. This form is easily provable and even auto completes it quickly. The inductive step regarding the large node constructor is simplified in the same fashion but this time auto cannot resolve the sub-goal, Sledgehammer though quickly finds a fix. Now that we got some familiarity with the Isabelle's proof process let's see the more tricky operations.

## 4.2 Insertion

The insertion procedure brings new challenges, both in the specification and proving. This is because aside from a few specific cases the tree needs to be re-balanced several times per insertion. The main reasoning behind the insert operation is to travel down the tree until the correct spot for insertion is found at the level of the terminal nodes. If now the value can be absorbed into the terminal node (that is to say, if a small node is encountered it can be turned into a large node) the operation is done. If the node cannot be absorbed it needs to be pushed back up the tree in order to find a spot for it. In order to achieve this change of direction in the traversal of the tree an extra data type that has two configurations is needed. One of these configurations is simply a wrapping of a valid 2-3 node that will be used in the backtracking phase once the value to be added has been absorbed. The other constructor instead is for when the value is being kicked up, in order to be able to re-balance the tree some knowledge about the tree is needed therefore a kicked up configuration is made, where the value to be added is stored, together with two sub-trees which correspond with the trees that would be children of a small node with v as its value at that level. Since another datatype is being used in addition to our already defined 2-3 nodes some attention needs to be paid to making sure that the insert operation always returns a correct tree. In order to achieve this, a small wrapper function is also needed.

This is because the tree traversal component needs to work with both valid nodes and kicked up configurations, more specifically it takes as input a 2-3 node and returns a kick-up, but such KickUp is not a valid 2-3 tree, so a small function is needed to convert this configuration once the procedure is complete. Since the tree only accepts distinct values the wrapper function also includes a check for the value to be inserted, therefore if the value is indeed already stored in the tree the function is terminated immediately.

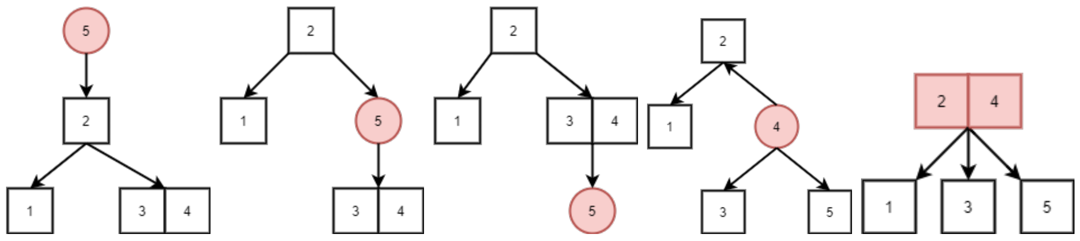

Figure 4.3: Image 1 introduces the tree and the value inserted in red. Each subsequent image represents a call of the recursive function. Images 2 and 3 are the downward phase of the function, while images 4 and 5 picture the upward phase

Getting into the proving part of the insertion operation, thanks to the invariants discussed before we know that there are two main theorems that need to be proven: maintenance of in-order sorting and maintenance of completeness. Let's start by looking at the maintenance of the ordering.

How can this be proved? One method is to compare the result of an insertion in the tree to an equivalent data structure. As seen before such data structure is the list, as previously seen for lookup the sorted function will be used. Lists also come with an $ins_list$ function that inserts an element in the list while maintaining it sorted.
Considering a tree *t* and the list *l* obtained from *elements t*. The result of applying *ins_list x l* should be identical to converting the result of (*insert a t*) to a list. For this theorem several minor lemmas are needed. To begin with, the main theorem has to deal with the wrapping function which covers two cases, one in which the element is already present in the tree and one where this isn't the case. The idea here is to have two smaller lemmas each covering a case. The first case is the simpler one since the tree does not change. There is one minor trick though; the tree does not change, but there is no lemma showing that if we *ins_list* an element in a list which contains it already the list remains unchanged. Luckily this is a fairly trivial statement to prove. The second case is the more complex one since it deals with the proper recursive function.

The result of *ins_list* has to be equal to the elements of the conversion of the tempIns structure obtained as a result of inserting a node in the tree. For this lemma a similar approach to the contain method proof has been followed. The proof is by structural induction, with the base case easily showed by simplification. At this point the inductive steps need to be broken down in their relative subcases. The application of the *ins_list*() function can be simplified with lemmas provided by the list theory. To simplify the ins function on the tree the definitions on which the cases are split are used. This will rewrite the two current sub-goals into the various cases connected by and operators. Now with auto this CNF term is broken down splitting each clause in its own subgoal. By exploding the sub-goals like this a better idea can be formed on what facts are needed to complete the proof. In this case what is needed is to show that if a tree has height 0, the elements of t correspond to []. Proving this is fairly trivial and can be accomplished with simp.

Now by using this new lemma as a simplification rule several subgoals can be completed and the remaining ones can be handled by Sledgehammer. Now that a working proof has been reached, similarly to what was done for lookup we want to make the proof neater, more readable and if possible faster. To simplify the goals as much as possible, let's add to the simplification step all the facts that have been used by Sledgehammer to provide a proof. This should break down the goals at an earlier stage, with the purpose of identifying a more direct proof. Looking at the conclusions left after this simplification it can be seen how some of them have been brought to the base case (dealing with an empty tree or list); this is generally good, because such cases are easier to reason about and often easily automated. This intuition proved right when auto is applied as the first 4 goals are solved.

Listing 4.4: Insertion theorems

```
theorem insert_order: "⟦sorted(elements t); distinct(elements(t))
    ; TT t⟧ ⟹ (elements(ins a t) = ins_list a (elements t)"

theorem complete_insert : "⟦sorted(elements t); TT t ⟧ ⟹
                           TT (ins a t) "
```

A function was previously constructed which tells us if a certain 2-3 tree is properly structured. Let's try to adapt the theorem for sortedness to also work with the shape. Since our interest is only in the shape of the tree and not the contents the sorted and distinct requirements can be removed from the premises. Like the previous theorem, this one will also consist of a general theorem for the ins function and an auxiliary lemma which covers the recursive insert function. From here the process proceeds much in the same as before. Using the automated tools it's possible to see what Isabelle has trouble completing: for an arbitrary small node n the result of ins corresponding to one of its children has the same height as the other child which was not modified. This final step of the Isabelle tools though is not very practical since showing that a part of the tree which has not been explored is equal to a new certain tree obtained from a temporary node is not very intuitive. A new approach was needed, so some intermediary lemmas were made, showing that no matter the temporary node the result of its conversion will be, it maintains the TT property. The validity of this invariant turned out to be by far the more challenging one.

This is because, as it turns out that the TT invariant by itself is not sufficient for this proof as it does not hold in some cases, explicitly when the tree grows in size (i.e. when the upward going node is absorbed at the root. This indicates the need for another proposition which combined with our TT property is congruent with all cases. This can be achieved by unifying the heights of the trees, by not counting the height of the final KickUp, which corresponds to the increase in height caused by the root absorption that does not influence the TT property. This addition also serves another fundamental purpose. Structural induction generates assumptions based on the constructors. This means that now in the premises of the subgoals there is the fact $heightTree\ t1 = heightTree\ x1a$. This is a key step, as this fact can now be used to simplify the conclusion to the form TT (convertKick ( some TempIns)). Once the correct lemma is identified the proof is much simpler to previous attempts simply by simplifying and splitting cases over the data-types involved in the function, the theorem can be proved. This property brought a different challenge as the more contrived aspect of it was realizing the error in the formulation of the theorem. As for sortedness the proof with our wrapper function also has to be proven. Differently from the lemma created for the recursive function, the conclusion of the theorem is limited to $TT(ins\ a\ t)$ as at the completion of the function the resulting tree has to follow the invariant.

## 4.3 Deletion

The deletion function is the most convoluted operation and even the specification comes with its challenges. Similarly to our previous procedure an additional type is used, TempDel, which wraps either a node or a hole. The terminology $D$ will be used to refer to the value to be deleted. Structurally the algorithm follows the insertion process, with a downward phase and an upward one. In this case though, the downward phase does not necessarily reach the leaves of the tree, as $D$ can reside in an internal node. This is quite a complex case that requires some complexity to be resolved. In other cases (i.e. when the target is in a terminal node), $D$ is replaced with a TempDel wrapper which is now carried up the tree until it can be placed. This can happen either at the root or on the way up the tree. The specific cases that are dealt with are given at the end of the study as they are not much relevance.

| Functions for delete specification | |
|---|---|
| del | 'a tttree ⇒ ('a::linorder ) ⇒ 'a tttree |
| nxt | 'a list ⇒ ('a::linorder) ⇒ a May |
| replace | ('a::linorder) ⇒ ('a::linorder) ⇒ 'a tttree ⇒ 'a tttree |
| delete | ('a::linorder) ⇒ 'a tttree ⇒ 'a TempDel |

The remaining question is dealing with $D$ as an internal node, as it requires the swap of itself with its in-order successor (or predecessor), which is then deleted. This slightly changes the approach used in previous operations, as having to find the predecessor of a value while iterating through the tree is not possible. The wrapping function this time will also perform a check on the position of $D$ in the tree. If $D$ is internal, its successor is deleted and $D$ is replaced with the successor in the result. The recursive function follows from the insertion one. At this point the proofs will revolve around proving preservation of invariants in the recursive function and replace functions. These will then be used in the general theorem which will cover the function.
The recursive part of the function though was also more involved than the earlier counterparts. Because of how the hole is absorbed in the tree, access to both children of the current node and the node itself is needed, for which there is no neat implementation trick. The first approach was to nest case separations on the needed node, but this lead to a blow up of goals to prove. Alternatively a placeholder datatype was created simulating the tttree datatype but with separate split facts. This solution was also unsatisfying as it produced poorly-formed premises. For this reason the smaller balancing have been moved to separate functions on which some auxiliary lemmas can more easily be proven.

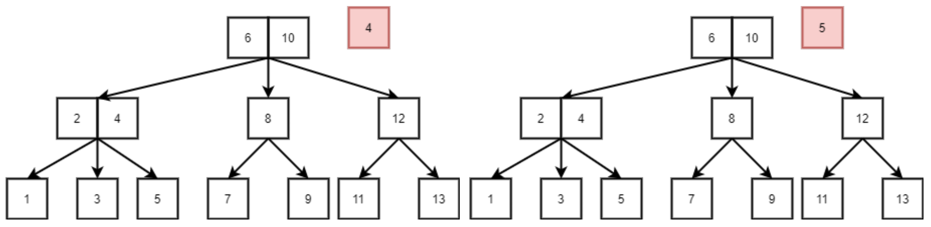

Figure 4.4: In this example D (in red) is an internal node, so when *del* is called it is called on the next value in order

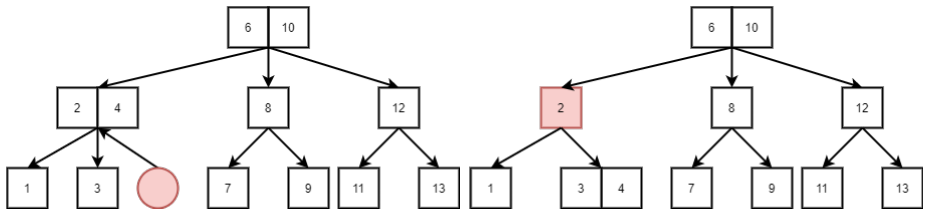

Figure 4.5: Here the upward phase of the deletion can be seen. The hidden complexity in this step is the choice of node restructuring.

Since a successful structure had been previously established for the proofs, it will still be used until failure. The main theorem this time is more complex as the wrapper function also has a more involved job. The lemma which formulation is simulates what was done in the corresponding insertion proof (only this time an element is deleted from the list).

The lemma which covers the case equivalent to the a call to the recursive function proved to be very challenging. Its formulation was straightforward with the only addition being the premise of $D$ being a in a leaf node. This is because any internal node deletion is solved at the wrapper level. Proving this theorem came with different challenges. Like was done for most main proofs until now, the first step was to isolate specific cases by splitting the goal over the various case splits in the function. From here different techniques were used to finalize the proof, without much success. The key issue here, which was common to all attempts at a solution is the following step in listing 4.3.

Listing 4.5: Maintenance of order in deletion

```
⟦...  ;delete a t1 = Hole x1a;  ...⟧ ⟹
elements (convertHole (small_small t2 x1a a)) = del_list a (
    elements t1 @ a # elements t2)
```

While this is trivial to prove for specific values of $t1$ , $t2$, the general concept relies on $x1a$ being equal to $t1$, $x1a$ being the result of the auxiliary functions on the specific value shuffling needed. Proving that these two are the same proved to be too much, as there is no clear way to extract infer much from the premise generating x1a. This issue was common to all goals of the same form regardless of which auxiliary function was used. Trying to solve this at the auxiliary function level also proved fruitless, as the presence of x1a in the premise as $delete\ a\ t1 = Hole\ x1a$ renders it impossible to gather information about what x1a looks like and how it was generated.

Chapter 5

# Conclusion

## 5.1 Related work

As of the time of this thesis, the only other 2-3 tree Isabelle/Hol implementation has been conducted by Tobias Nipkow [8]. He has made a map and a set implementation using 2-3 trees. In addition Nipkow provides a plethora of other search trees implementations from which starting lemmas were inspired (such as [9]). Nipkow also bases his implementation on Turbak's work[7]. Nipkow's implementation is more elegant and concise, while the implementation presented here has some optimizations that should speed up execution in certain cases. His implementation handles duplicate values differently in case of duplicates by nullifying them directly in the recursive function. The delete function also varies greatly in its different handling of deletion of internal nodes. Nipkow's proofs are more complete as he also provides full proofs for the delete function. Aside from this specification, no other was found for 2-3 trees. Expanding the search to other interactive theorem provers such as Coq, only specifications dealing insert with were found. Looking at a wider scope, some work was conducted with B-trees, particularly the specification provided by Mündler [10], but the general nature of B-trees combined with the imperative nature of the specification lead to a quite different proof structure.

## 5.2 Summary of contributions

Overall the work conducted in this thesis has provided a verified specification of 2-3 trees in Isabelle, after the one provided in the HOL library, with the exception of the completeness invariant for the delete function. Isar proofs are also provided for most theorem proved. The tree is implemented as a basis for a set ADT, given it's property of only storing distinct values. The thesis also hopes to give some insight in a first user experience with Isabelle/HOL. The Isabelle/HOL theory (around 400 lines) can be browsed at the following link [11].

## 5.3 Future Work

Overall there are several lines of research that could be expanded upon. The more obvious path would be to complete the maintenance of invariants for the delete function. Another line of continuation of the project would be to provide full structured Isar proofs for all the sequential proofs presented in the thesis. The tree could be expanded into more data structures based on 2-3 trees, like Nipkow has done in with his aforementioned work, using the tree to implement a set, list and map. Another interesting function to add to the specification is the join function, which allows to merge two trees. As a longer project, this work could be used a basis to understand a simpler case of B-trees, which having a variable amount of values in nodes offer a higher degree of difficulty in both specification and proving [10]. This would expand the utility of the exercise to the applied field, compared to the more theoretical exercise done here. This is because to properly take advantage of B-trees properties in current machine architectures a node degree of 3 is not sufficient.

# Bibliography

[1] Alok Sanghavi. "What is formal verification?" In: (May 2010).

[2] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic.* Vol. 2283. Springer Science & Business Media, 2002.

[3] Robin Milner et al. *The definition of standard ML: revised.* MIT press, 1997.

[4] Sascha Böhme and Tobias Nipkow. "Sledgehammer: judgement day". In: *International Joint Conference on Automated Reasoning.* Springer. 2010, pp. 107–121.

[5] R. Bayer Mathematical et al. *Organization and maintenance of large ordered indices.* Nov. 1970. URL: `https://dl.acm.org/doi/10.1145/1734663.1734671`.

[6] Donald E Knuth. *The art of computer programming, 2nd edn. Sorting and searching, vol. 3.* 1998.

[7] Lyn Turbak. *2-3 Trees.* URL: `https://www.cs.princeton.edu/~dpw/courses/cos326-12/ass/2-3-trees.pdf`.

[8] Tobias Nipkow. *Functional Data Structures.* 2020. URL: `https://isabelle.in.tum.de/library/HOL/HOL-Data_Structures/document.pdf`.

[9] Tobias Nipkow. *2-3 Trees.* URL: `https://isabelle.in.tum.de/library/HOL/HOL-Data_Structures/Tree_Set.html`.

[10] Niels Mündler. *A Verified Imperative Implementation of B-Trees.* URL: `https://mediatum.ub.tum.de/doc/1596550/1596550.pdf`.

[11] URL: `https://github.com/sbolognani/2-3Tree`.