Formal Verification of AVL Trees in HOL4

Sineeha Kodwani

October 24, 2024

Abstract

Interactive Theorem Proving (ITP) enables researchers to build reusable libraries of formalized mathematics, algorithms, and data structures. This thesis contributes to the formalization of AVL trees in the HOL Theorem Prover (HOL4), an interactive theorem-proving system based on higher-order logic. The formalization covers the theoretical foundations of AVL trees, focusing on essential properties such as height constraints, node counts, and balancing operations through rotations after insertions and deletions. A recursive relationship between the number of nodes in an AVL tree and the Fibonacci sequence is established, demonstrating that minimal AVL trees achieve optimal performance with a height logarithmic to the number of nodes. Additionally, design adaptations made to integrate the formalization into HOL4's framework are discussed.

We also compared with some related work, such as Isabelle's AVL tree formalization, and the existing balanced binary search trees (`balanced_map`) in HOL4. The present work aims to become a part of HOL4's official examples, adding to its valuable collection of verified data structures for future developers.

1 Introduction

Interactive Theorem Proving (ITP) involves the formal verification of mathematical theorems and data structures through human-guided proof systems. The HOL theorem prover (HOL4) is one such system based on Higher-Order Logic (HOL). The formal verification of algorithms within HOL4 helps ensuring their correctness, and contributes to building a reusable, verified library of algorithms and data structures.

AVL trees are self-balancing binary search trees, where the height difference of left and right subtrees for any tree node is less than or equal to one. This balancing requirement ensures logarithmic height with respect to the number of nodes, making search, insertion, and deletion operations on AVL trees more efficient (in $O(\log n)$) than those on linear data structures ($O(n)$) such as arrays and linked lists.

The goal of this thesis is to provide a formal verification of AVL trees using HOL4. This involves defining AVL trees in HOL4, proving fundamental properties such as balance conditions and operations (insertion, deletion, rotation), and comparing this formalization to existing formalizations in systems like Isabelle.

2 Higher-Order Logic and HOL4

In this section, we briefly introduce Higher-Order Logic (HOL) and its implementation, HOL4.

2.1 Introduction to Higher-Order Logic

Higher-Order Logic (HOL) extends first-order logic by enabling quantification over individual objects, predicates, and functions, providing an expressive framework for formal reasoning. This expressiveness is crucial for formal verification tasks, where one needs to specify and prove properties of complex systems such as algorithms, data structures, and hardware designs.

Unlike first-order logic, which only allows quantification over variables that represent objects, HOL allows quantification over higher-level entities like functions and sets of functions. This feature is useful when formalizing sophisticated mathematical structures and operations, such as the present work for AVL trees, where recursive functions are used for defining balance conditions and other properties.

2.2 HOL Theorem Prover

The HOL Theorem Prover (HOL4) [G+08], a prominent implementation of HOL, offers a robust, interactive, and automated proof environment. It is built on an ML-based foundation, originally developed by Mike Gordon in the 1980s as an adaptation of the Edinburgh LCF system, and has since evolved into a powerful tool for formal verification tasks [GM93]. HOL4's entire software system is distinguished by its small trusted kernel, which encapsulates the core logical rules, ensuring the soundness of the proofs derived from it. This architecture allows HOL4 to remain flexible and reliable while offering an extensive suite of derived rules and proof tools that support advanced features like quotient types, mutual recursion, and automated reasoning.

HOL4's libraries are extensive, covering a wide range of mathematical and computational domains—from arithmetic and number theory to program logics and hardware models like the ARM architecture [Har09]. These libraries are not only persistent but also customizable, allowing users to build their proofs atop established formalizations or create new ones tailored to their specific needs. This makes HOL4 especially suitable for verifying both theoretical constructs and practical implementations. Its ability to mix automated reasoning with human-guided proof development further enhances its versatility, enabling efficient handling of complex proofs with both precision and flexibility [NPW02].

For tasks involving AVL trees, HOL4's capabilities are particularly valuable. The formal verification of AVL trees in HOL4 involves defining the tree structure, specifying invariants like balance conditions, and proving the correctness of operations such as insertion, deletion, and tree rotations. Additionally, HOL4 supports reasoning about recursive relationships, which is critical in establishing the connection between AVL trees and the Fibonacci sequence—a key factor in demonstrating that AVL trees maintain logarithmic height relative to the number of nodes.

HOL4 integrates interactive proof guidance with automated reasoning tools, such as SAT solvers and first-order solvers, allowing researchers to handle both small-scale

formal verifications and large-scale system validations. The mix of automation and interactivity makes HOL4 an indispensable tool in formal methods, particularly when verifying correctness properties for data structures and algorithms, as is the case with AVL trees in this thesis.

Moreover, HOL4's libraries and its proven track record in hardware and software verification projects make it an essential tool for both academic and industrial verification tasks. For example, HOL4 has been used to verify the correctness of hardware architectures and safety-critical software systems [AH14], underscoring its importance in the broader context of formal verification.

HOL4 is especially useful in this thesis due to its integration of higher-order logic and its extensive libraries for formal verification tasks, which allow the formal proof of properties such as the balance conditions and recursive relationships in AVL trees.

3 AVL Trees in HOL

3.1 AVL Trees

An AVL tree is defined as a binary search tree that maintains a balance condition, which ensures that for any node, the height of its left and right subtrees differs by at most one. An AVL tree stores finite number of key-value pairs, where all keys must be distinct.

In HOL4, AVL trees can be represented by an inductive, algebraic datatype, where the type of data to be stored in the tree as a type variable. Without loss of generality, we consider keys as natural numbers.

3.2 Properties of AVL Trees

In theory, a complete binary tree of height n (the height of single node is 1) has $2^n - 1$ nodes, and thus the height of complete binary trees is in logarithmic (more precisely, in log 2) of the number of nodes. Note that, in the worst case, a valid AVL tree may be unbalanced (i.e. the height different of two subtree is 1) at every non-leave node, and it is not immediate that the height of such unbalanced trees is still in logarithmic of the number of nodes.

In fact, the maximal height of an AVL tree can be proved to be logarithmic to the number of nodes, ensuring efficient operations. AVL trees use rotations to maintain their balance after insertion or deletion operations. This proof here, however, is only given indirectly, by a smart connection to Fibonacci series, which is known to have logarithmic upper and lower bounds.

4 Formalization of AVL Trees in HOL4

In this section, we present the formalization of AVL trees in the HOL4 theorem prover. The following definitions and theorems outline the construction and properties of AVL trees as verified within HOL4. The precise code of these formalizations is available as part of the appendix and has been verified using HOL4.

4.1 Datatype and Definitions

In this section, we present the fundamental definitions and data structures used in the formalization of AVL trees in the HOL4 theorem prover. AVL trees are a specific type of self-balancing binary search tree that ensures logarithmic height by maintaining a balance between the heights of its subtrees. These definitions provide the foundation for verifying the correctness and properties of AVL trees, which are crucial for efficient data storage and retrieval.

Datatype An AVL tree is a self-balancing binary search tree where the heights of the two child subtrees of any node differ by at most one. This balance ensures that search, insertion, and deletion operations remain efficient by maintaining a logarithmic height relative to the number of nodes in the tree. When the tree becomes unbalanced after an insertion or deletion, rebalancing operations are performed to restore the AVL property.

In HOL4, the datatype of an AVL tree is formally defined as:

```
avl_tree = Tip | Bin int num 'a avl_tree avl_tree
```

Here, the AVL tree can be either a `Tip`, representing an empty tree, or a `Bin` node, which contains the following (Note that a singleton AVL tree is constructed by a `Bin` node with two `Tip`s):

- **int**: The balance factor of the node, which is the difference between the heights of the left and right subtrees.
- **num**: The key associated with the node.
- **'a avl_tree**: The left subtree.
- **'a avl_tree**: The right subtree.

This recursive structure allows to define the properties of AVL trees and the operations performed on them. The balance factor is a cached value calculated by the height of right subtree subtracts the height of left subtree. It is crucial for quickly determining whether the tree remains balanced after insertion or deletion operations.

Definition of Height The `height` function computes the height of an AVL tree, which is defined as the number of edges from the root to the furthest leaf. The height is essential for maintaining the AVL property, as it is used to calculate the balance factor of a node. It is formally defined in HOL4 as follows:

```
[height_def]
height Tip ≝ 0
height (Bin h k v l r) ≝ MAX (height l) (height r) + 1
```

The height of an empty tree is 0, and for a non-empty tree, the height is determined by the maximum height of its left and right subtrees, incremented by 1. This ensures that the height of a balanced AVL tree grows logarithmically with the number of nodes.

Definition of Singleton AVL Tree A singleton AVL tree consists of a single node. This is a base case used in insertion operations when creating new nodes in the tree. The formal definition in HOL4 is:

```
[singleton_avl_def]
singleton_avl k v ≝ Bin 0 k v Tip Tip
```

This creates a node with a balance factor of 0, a key `k`, a value `v`, and two empty subtrees (`Tip`). The balance factor of 0 indicates that the node is perfectly balanced with no children.

AVL Predicate The predicate `avl` is used to check whether a tree satisfies the AVL property. For a tree to be an AVL tree, the height difference between its left and right subtrees must be at most 1. This ensures that the tree remains balanced. The formal definition of the AVL predicate in HOL4 is:

```
[avl_def]
avl Tip ≝ T
avl (Bin bf k v l r) ≝
  (height l = height r ∨ height l = height r + 1 ∨
   height r = height l + 1) ∧ bf = &height r − &height l ∧
  avl l ∧ avl r
```

This definition recursively checks the balance condition at every node, ensuring that both the left and right subtrees also satisfy the AVL property. The balance factor, calculated from the heights of the left and right subtrees, ensures that the AVL property holds after each operation.

Node Count The `node_count` function computes the number of nodes in an AVL tree. This function is important for evaluating the structure of the tree and ensuring that the tree is minimally balanced. The formal definition is:

```
[node_count_def]
node_count Tip ≝ 0
node_count (Bin bf k v l r) ≝ node_count l + node_count r + 1
```

The node count of an empty tree is zero, while for a non-empty tree, the node count is the sum of the node counts of its left and right subtrees, plus one for the root node.

Minimal AVL Trees A minimal AVL tree is defined as one that satisfies the AVL property while having the smallest possible number of nodes for a given height. This is essential for ensuring that AVL trees maintain optimal structure and performance. The formal definition in HOL4 is:

```
[minimal_avl_def]
minimal_avl t ≝
  avl t ∧
  ∀t'.
    avl t' ∧ height t' = height t ⇒
    node_count t ≤ node_count t'
```

This predicate ensures that for any AVL tree t' of the same height, the number of nodes in t is less than or equal to the number of nodes in t', thus ensuring minimality.

Complete AVL Tree A complete AVL tree is one in which all levels are fully filled, except possibly for the last level, which is filled from left to right. A complete AVL tree is balanced and maintains the AVL property. The formal definition of a complete AVL tree in HOL4 is:

```
[complete_avl_def]
complete_avl 0 ≝ Tip
complete_avl (SUC n) ≝
  Bin 0 0 ARB (complete_avl n) (complete_avl n)
```

This recursive definition constructs a balanced AVL tree of height n by creating left and right subtrees of height n. A complete AVL tree is useful for understanding the optimal balance and node distribution in AVL trees of varying heights.

4.2 Theorems on AVL Trees

The formal verification of key properties of AVL trees is essential to ensuring that operations such as insertion, deletion, and rotation maintain the necessary balance constraints and achieve optimal performance. The following theorems have been rigorously proven within the HOL4 framework, focusing on core properties such as height, balance factors, node counts, and recursive relationships between these properties. These theorems are fundamental to proving the correctness of AVL trees and ensuring that they exhibit the desired logarithmic time complexity for operations.

The proof of these theorems within HOL4 serves as a guarantee that AVL trees behave optimally and maintain their essential balancing properties across various operations. Each theorem is accompanied by a description of its significance in the context of AVL trees.

- **Theorem: height_eq_0**
  This theorem asserts that the height of a tree is zero if and only if the tree is empty, represented by a `Tip`. This result is crucial as it establishes the base case for many proofs involving AVL trees, particularly those related to height and balance. The formal statement of the theorem is:

  ```
  [height_eq_0]
  ⊢ (height t = 0 ⟺ t = Tip) ∧ (0 = height t ⟺ t = Tip)
  ```

  This theorem ensures that for any AVL tree t, if the height of t is zero, it must be an empty tree. This result is foundational for verifying AVL tree operations, as many structural properties depend on correctly identifying the base case of an empty tree. It also plays a critical role in inductive proofs where the height of a tree is reduced step-by-step.

- **Theorem: avl_complete_avl**
  This theorem proves that a complete AVL tree satisfies the AVL property. A complete AVL tree is a tree where all levels are fully filled except possibly the last, which is filled from left to right. This structure guarantees that the tree is balanced. The formal statement is:

  ```
  [avl_complete_avl]
  ⊢ avl (complete_avl k) ⟺ T
  ```

  This theorem is significant as it confirms that the construction of a complete AVL tree results in a valid AVL tree. Complete AVL trees are commonly used as examples of balanced trees and serve as a useful model for understanding how the AVL property is maintained as trees grow in height.

- **Theorem: minimal_avl_exists**
  This theorem establishes the existence of minimal AVL trees for any given height. A minimal AVL tree is defined as the tree that satisfies the AVL property while containing the smallest possible number of nodes for that height. The formal statement is:

  ```
  [minimal_avl_exists]
  ⊢ ∀k. ∃t. minimal_avl t ∧ height t = k
  ```

  The significance of this theorem lies in its demonstration that AVL trees can be constructed in an optimal manner with respect to node count. Minimal AVL trees are important because they represent the most compact tree structure that still satisfies the AVL property. This guarantees that AVL trees can be kept as small as possible, improving space efficiency while maintaining balance and performance.

- **Theorem: minimal_avl_node_count**
  This theorem relates the node count of a minimal AVL tree to the Fibonacci sequence, establishing a deep connection between AVL tree structure and the Fibonacci series. Specifically, it states that the number of nodes in a minimal AVL tree of height k is equal to $N(k)$, where $N(k)$ is the number of nodes in a Fibonacci tree of height k. The formal statement is:

  ```
  [minimal_avl_node_count]
  ⊢ ∀k t. minimal_avl t ∧ height t = k ⇒ node_count t = N k
  ```

  The Fibonacci sequence plays a critical role in AVL trees because it captures the optimal relationship between height and node count. A minimal AVL tree of height k has a node count that grows in a Fibonacci-like fashion, which ensures that the tree remains as compact as possible while still allowing logarithmic height growth. This result highlights the efficiency of AVL trees in terms of both space and time complexity, as it demonstrates that the height of an AVL tree grows logarithmically with the number of nodes.

- **Theorem: N_fibonacci_relation**
  This theorem builds on the previous result by explicitly stating the recursive relationship between the number of nodes in an AVL tree and the Fibonacci sequence. It shows that the number of nodes in a minimal AVL tree of height k follows the Fibonacci sequence. The formal statement is:

  ```
  [N_fibonacci_relation]
  ⊢ ∀k. N k = Fibonacci (k + 2) − 1
  ```

  This theorem provides a more detailed characterization of the relationship between the number of nodes in an AVL tree and its height. The Fibonacci sequence governs the growth of AVL trees, and this theorem demonstrates that the number of nodes in a minimal AVL tree of height k is equal to the $(k + 2)$-th Fibonacci number minus one. This result is essential for understanding the recursive structure of AVL trees and their efficiency, as it proves that the trees grow in a balanced and optimal manner according to the Fibonacci series.

- **Theorem: height_complete_avl**
  This theorem states that the height of a complete AVL tree of height n is indeed n. The formal definition ensures that the structure of the tree follows the expected pattern for a balanced binary tree. The formal statement is:

  ```
  [height_complete_avl]
  ⊢ height (complete_avl n) = n
  ```

  This theorem is significant because it guarantees that the height of a complete AVL tree is correctly calculated, which plays a crucial role in operations such as insertion and balancing. Knowing the height of a tree is necessary for maintaining the balance condition after modifications.

- **Theorem: avl_insert_aux**
  This theorem verifies that the insertion of a new element into an AVL tree preserves the AVL property. When a new element is added to the tree, the AVL property must be checked and potentially restored through rebalancing operations. The formal statement is:

  ```
  [avl_insert_aux]
  ⊢ ∀k v t.
      avl t ⇒
      avl (insert_avl k v t) ∧
      (height (insert_avl k v t) = height t ∨
       height (insert_avl k v t) = height t + 1)
  ```

  This theorem ensures that after inserting a key-value pair (k, v) into the tree, the resulting tree still satisfies the AVL property. This is critical for the correctness of insertion operations, as it guarantees that the tree remains balanced after an insertion.

These theorems, together, form the foundation of the formal verification of AVL trees in HOL4. They ensure that key properties such as height, balance factors, and node counts are maintained across various operations. The recursive relationship between the number of nodes and the Fibonacci sequence is particularly important, as it highlights the optimality of AVL trees with respect to both time and space efficiency. By proving these theorems, we can guarantee that AVL trees are implemented in a provably correct manner, ensuring their effectiveness in real-world applications. Furthermore, the Fibonacci-based structure ensures that AVL trees can handle a large number of elements while maintaining efficient operations.

4.3 Insertion and Balancing Operations

One of the key properties of AVL trees is their ability to maintain balance after insertions. After adding a new element to the tree, it is possible that the balance factor (the difference between the heights of the left and right subtrees) exceeds the allowed limit of 1. When this happens, the tree must be rebalanced to restore the AVL property. The balancing operations ensure that the tree maintains its logarithmic height, which is essential for efficient searching, insertion, and deletion operations.

In this section, we define two core balancing operations, `balanceL` and `balanceR`, which handle cases where the left or right subtree becomes too tall, respectively. We also describe the `insert_avl` function, which is responsible for inserting elements into the tree while ensuring that the AVL property is preserved through rebalancing.

- **balanceL**
  The function `balanceL` is responsible for rebalancing the tree when the left subtree is taller than the right subtree by more than one level. This situation can occur after an insertion into the left subtree. If the left subtree becomes imbalanced, `balanceL` performs a rotation to restore balance. The formal definition in HOL4 is:

```
[balanceL_def]
balanceL k v l r ≝
  if height l = height r + 2 then
    case l of
      Tip ⇒ tree k v l r
    | Bin v1 lk lv ll lr ⇒
      if height ll < height lr then
        case lr of
          Tip ⇒ tree lk lv ll (tree k v lr r)
        | Bin v6 lrn lrv lrl lrr ⇒
          tree lrn lrv (tree lk lv ll lrl) (tree k v lrr r)
      else tree lk lv ll (tree k v lr r)
  else tree k v l r
```

  This operation works by comparing the heights of the left and right subtrees of the current node. If the left subtree is too tall (i.e., the height difference is greater than one), a rotation is performed. The precise rotation depends on whether the

imbalance is caused by the left or right child of the left subtree. The function handles both single and double rotations to restore balance. The balancing of the left subtree is critical for maintaining the AVL property and ensuring that the tree remains balanced after an insertion on the left side.

- **balanceR**
  Similarly, the `balanceR` function rebalances the tree when the right subtree becomes too tall compared to the left subtree. This can occur after an insertion into the right subtree. If the right subtree becomes imbalanced, `balanceR` restores balance by performing a rotation. The formal definition in HOL4 is:

```
[balanceR_def]
⊢ balanceR k v l r =
  if height r = height l + 2 then
    case r of
      Tip ⇒ tree k v l r
    | Bin v1 rk rv rl rr ⇒
      if height rl > height rr then
        case rl of
          Tip ⇒ tree rk rv (tree k v l rl) rr
        | Bin v6 rln rlv rll rlr ⇒
          tree rln rlv (tree k v l rll) (tree rk rv rlr rr)
      else tree rk rv (tree k v l rl) rr
  else tree k v l r
```

  The logic for `balanceR` is analogous to `balanceL`, but it handles the case where the right subtree is imbalanced. Similar to `balanceL`, this function performs either a single or double rotation depending on the structure of the right subtree. Rebalancing the right subtree is essential for maintaining the AVL property after an insertion into the right child of the tree.

- **insert_avl**
  The function `insert_avl` is responsible for inserting a new key-value pair into an AVL tree while preserving the AVL property. After inserting the new element, the tree may become unbalanced, so rebalancing operations such as `balanceL` or `balanceR` are invoked as necessary. The formal definition in HOL4 is:

```
[insert_avl_def]
insert_avl x v Tip ≝ singleton_avl x v
insert_avl x v (Bin bf k kv l r) ≝
  if x = k then Bin bf k kv l r
  else if x < k then balanceL k kv (insert_avl x v l) r
  else balanceR k kv l (insert_avl x v r)
```

  In the `insert_avl` function, the key-value pair (x, v) is inserted into the correct position in the tree according to the binary search tree property. If x is smaller than the current key k, the insertion proceeds in the left subtree. If x is larger than

k, the insertion proceeds in the right subtree. After the insertion, the balance of the tree is checked. If the tree becomes imbalanced, either `balanceL` or `balanceR` is applied to restore balance.

The key feature of the `insert_avl` function is that it ensures the AVL property is maintained at all levels of the tree. By performing rebalancing operations as needed, the function guarantees that the height of the tree remains logarithmic with respect to the number of nodes. This is crucial for ensuring that search, insertion, and deletion operations can be performed efficiently in $O(\log n)$ time.

The `balanceL` and `balanceR` operations are vital for maintaining the AVL property after modifications to the tree. Without these operations, the tree could become unbalanced, leading to poor performance in subsequent operations. The `insert_avl` function ensures that new elements can be added to the tree while keeping it balanced. The careful handling of rotations in both the left and right subtrees allows AVL trees to maintain their optimal height and performance.

4.4 Key Theorems for Insertion

The insertion of elements into an AVL tree must maintain the AVL property, the binary search tree property, and ensure that the keys are updated correctly. The following theorems formally prove the correctness of the insertion algorithm in HOL4. Each theorem plays a critical role in verifying that after an element is inserted, the resulting tree remains balanced, its height is properly adjusted, and the set of keys is correctly updated. These proofs are essential for demonstrating that AVL trees continue to function optimally after an insertion operation.

- **Theorem: keys_insert**
  This theorem proves that after inserting a key x into an AVL tree t, the set of keys in the resulting tree is the union of the set of keys in t and the singleton set $\{x\}$. The formal statement of the theorem is:

  ```
  [keys_insert]
  ⊢ ∀x v t. keys (insert_avl x v t) = keys t ∪ {x}
  ```

  This theorem demonstrates that the insertion of a key into an AVL tree correctly updates the set of keys stored in the tree. Specifically, if x is inserted into the tree t, the resulting set of keys will be the same as the set of keys in t, with the addition of x. The proof proceeds by structural induction on the tree, verifying that the insertion preserves the binary search tree property (i.e., that keys are correctly placed in the left or right subtrees according to their values) and that no duplicate keys are introduced.

  The correctness of key updates is a fundamental aspect of AVL trees. This ensures that after any insertion operation, the tree continues to function as a valid search tree, allowing for efficient lookup operations. By showing that the set of keys is correctly updated, this theorem guarantees that the AVL tree remains a proper binary search tree after insertion.

- **Theorem: height_insert_avl**
  This theorem ensures that after inserting a key x into an AVL tree, the height of the tree is either preserved or increased by one. The formal statement is:

```
!k v t. height(insert_avl k v t) = height t \/ height(insert_avl k v t) = heigh
```

  The significance of this theorem lies in its guarantee that the height of an AVL tree is maintained within strict bounds after an insertion operation. If the tree remains balanced after the insertion, its height will not change. However, if the insertion causes the tree to become imbalanced, a rotation (as handled by the `balanceL` or `balanceR` functions) will restore the balance, possibly increasing the height by at most one.

  This theorem is crucial for ensuring that the AVL tree continues to exhibit logarithmic height relative to the number of nodes, which is key to maintaining efficient search, insertion, and deletion operations.

- **Theorem: avl_insert_avl**
  This theorem proves that the AVL property is preserved after the insertion of a new element into the tree. Specifically, it shows that if the tree satisfies the AVL property before insertion, it will continue to satisfy this property after the insertion. The formal statement is:

```
!k v t. avl t ==> avl(insert_avl k v t)
```

  The proof of this theorem involves showing that after inserting a new element into the tree, the balance factor at each node is either maintained or adjusted correctly through the balancing operations. The `balanceL` and `balanceR` functions ensure that any imbalance is resolved, restoring the AVL property to the tree.

  This theorem is fundamental for proving the correctness of the insertion operation in AVL trees. It guarantees that after an insertion, the tree remains balanced, which is essential for ensuring that the height of the tree remains logarithmic with respect to the number of nodes. Without this guarantee, the tree could become unbalanced, leading to degraded performance in subsequent operations.

- **Theorem: avl_tree_preserves_avl**
  This theorem establishes that the insertion of a new node into an AVL tree preserves the AVL property at all levels of the tree. It is formally stated as:

```
[avl_tree_preserves_avl]
 ⊢ avl l ∧ avl r ∧
   (height l = height r ∨ height l = height r + 1 ∨
    height r = height l + 1) ⇒
   avl (tree k v l r)
```

This theorem is a more general form of the previous theorem, which ensures that the AVL property is preserved not just at the root of the tree, but at every node where balancing occurs. The condition on the heights of the left and right subtrees ensures that the balance factor is within the allowed range of $[-1, 1]$, maintaining the AVL property throughout the entire tree.

The importance of this theorem lies in its guarantee that the AVL tree remains balanced after any insertion, deletion, or rebalancing operation. By proving that the AVL property is preserved at all levels, this theorem ensures that AVL trees maintain their logarithmic height and efficient performance characteristics.

The proofs of these theorems collectively demonstrate that the insertion operation in AVL trees is both correct and efficient. By ensuring that the AVL property is preserved, the height is properly adjusted, and the set of keys is correctly updated, these theorems guarantee that AVL trees continue to function optimally after insertion. Furthermore, the balancing operations (handled by `balanceL` and `balanceR`) ensure that any imbalance introduced by the insertion is corrected, preserving the logarithmic height of the tree and ensuring efficient search and insertion times.

5 Recursive Relationship Between Nodes and Fibonacci Sequence

In this section, we explore the deep connection between AVL trees and the Fibonacci sequence, a well-known sequence that plays a crucial role in ensuring the optimal structure of AVL trees. AVL trees are designed to maintain a balanced height relative to the number of nodes they contain, and this balance can be directly tied to the growth pattern of the Fibonacci sequence. Specifically, the number of nodes in an AVL tree of height k is closely related to the $(k+2)$-th Fibonacci number. This relationship ensures that the height of an AVL tree grows logarithmically with respect to the number of nodes, making AVL trees highly efficient for operations such as search, insertion, and deletion.

The recursive nature of the Fibonacci sequence mirrors the recursive structure of AVL trees, where each node is balanced based on the heights of its left and right subtrees. The Fibonacci sequence thus provides a mathematical framework for understanding the growth and balance properties of AVL trees. In this section, we formalize the Fibonacci sequence in HOL4 and demonstrate how it is used to prove important properties of AVL trees.

5.1 Formalization of Fibonacci Sequence

The Fibonacci sequence is a recursive sequence where each term is the sum of the two preceding ones, starting from 0 and 1. The Fibonacci sequence is formally defined as:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

This recursive structure of the Fibonacci sequence mirrors the recursive nature of balanced binary trees, making it a natural fit for analyzing the properties of AVL trees.

In HOL4, the Fibonacci sequence is formalized as follows:

```
[Fibonacci_def]
Fibonacci n ≝
  if n = 0 then 0
  else if n = 1 then 1
  else Fibonacci (n − 1) + Fibonacci (n − 2)
```

This definition of the Fibonacci sequence allows us to compute Fibonacci numbers recursively. Starting with the base cases $F(0) = 0$ and $F(1) = 1$, the function recursively computes each subsequent Fibonacci number by summing the previous two values. This recursive definition plays a pivotal role in establishing the relationship between the height of an AVL tree and the number of nodes it contains.

5.2 Relationship Between Fibonacci Sequence and AVL Trees

The relationship between the Fibonacci sequence and AVL trees is rooted in the fact that minimal AVL trees (i.e., AVL trees with the smallest number of nodes for a given height) exhibit a structure similar to that of Fibonacci trees. Minimal AVL trees grow in a pattern that can be described by Fibonacci numbers, which means that the number of nodes in a minimal AVL tree of height k is closely related to the $(k+2)$-th Fibonacci number.

The following theorem formally establishes this relationship in HOL4:

- **Theorem: N_fibonacci_relation**
  This theorem proves that the number of nodes in a minimal AVL tree of height k is given by $F(k+2)-1$, where F represents the Fibonacci sequence. The formal statement of the theorem is:

  ```
  [N_fibonacci_relation]
  ⊢ N k = Fibonacci (k + 2) − 1
  ```

  The proof of this theorem involves showing that the recursive structure of minimal AVL trees corresponds exactly to the recursive structure of the Fibonacci sequence. Specifically, the number of nodes in a minimal AVL tree of height k is the sum of the number of nodes in the left and right subtrees (which are minimal AVL trees of smaller heights), plus one for the root node. This recursive structure aligns perfectly with the Fibonacci sequence, where each term is the sum of the two preceding terms.

  This theorem is significant because it demonstrates that the height of a minimal AVL tree grows logarithmically with respect to the number of nodes. The Fibonacci sequence grows exponentially, meaning that the height of the tree increases slowly as the number of nodes increases. This property is what allows AVL trees to maintain efficient operations, as the logarithmic height ensures that search, insertion, and deletion operations can be performed in $O(\log n)$ time.

5.3 Theorem: Fibonacci Recursive Properties

Another important result in the formalization of AVL trees is the recursive nature of the Fibonacci sequence itself. This recursive property plays a critical role in the inductive proofs used to establish the correctness of the AVL tree algorithms. The following theorem captures the recursive definition of the Fibonacci sequence:

- **Theorem: Fibonacci_thm**
  This theorem states that the Fibonacci sequence satisfies the following recursive relationship:

  ```
  [Fibonacci_thm]
  ⊢ Fibonacci (k + 2) = Fibonacci (k + 1) + Fibonacci k
  ```

  The proof of this theorem follows directly from the definition of the Fibonacci sequence. This recursive relationship is central to many of the inductive proofs involving AVL trees, as it provides a framework for reasoning about the growth of the tree as nodes are added or removed. By leveraging the recursive nature of the Fibonacci sequence, we can prove that AVL trees maintain their logarithmic height across all operations.

5.4 The Importance of Fibonacci Sequence in AVL Trees

The recursive relationship between the number of nodes in an AVL tree and the Fibonacci sequence is fundamental to understanding the efficiency of AVL trees. Since the Fibonacci sequence grows exponentially, while the height of an AVL tree grows logarithmically with respect to the number of nodes, this ensures that AVL trees are highly efficient in terms of both time and space complexity.

The Fibonacci sequence helps to establish that minimal AVL trees are optimally balanced, with the smallest possible number of nodes for a given height. This balance is what allows AVL trees to maintain their logarithmic height, ensuring that operations such as search, insertion, and deletion can be performed in $O(\log n)$ time. By formally proving the relationship between the Fibonacci sequence and AVL trees in HOL4, we can guarantee the correctness and efficiency of AVL tree algorithms.

6 Design and Integration in HOL4

The formal verification of data structures in HOL4 requires not only defining the structure and properties of the data structure but also ensuring that it integrates seamlessly with the theorem prover's existing framework. In this section, we focus on the design considerations, adaptations, and extensions made to formalize AVL trees within HOL4, a higher-order logic theorem prover. These adaptations were necessary to ensure that the formalization of AVL trees fits into HOL4's framework, allowing future developers to build upon this work easily and enabling the reuse of existing libraries within HOL4.

6.1 Adapting the AVL Formalization to HOL4's Framework

The HOL4 theorem prover has a robust framework for formalizing mathematical structures and reasoning about their properties. However, any new formalization must be carefully integrated into this existing ecosystem. In the case of AVL trees, several key design decisions were made to ensure that the formalization fits smoothly into HOL4's libraries and adheres to its conventions. Below, we discuss the key design adaptations and the rationale behind them.

6.2 Choosing the Appropriate Datatype Representation

HOL4 provides a rich environment for defining custom datatypes, which is essential for formalizing data structures like AVL trees. The choice of datatype representation is crucial because it affects how the formalization interacts with existing tools and libraries within HOL4. AVL trees, as a recursive data structure, are defined in HOL4 using the following datatype:

```
avl_tree = Tip | Bin int num 'a avl_tree avl_tree
```

This representation captures the essential structure of AVL trees:

- `Tip`: Represents an empty tree.
- `Bin`: Represents a node in the tree, containing:
  - **int**: The balance factor, which stores the difference in height between the left and right subtrees.
  - **num**: The key associated with the node.
  - **'a avl_tree**: The left subtree.
  - **'a avl_tree**: The right subtree.

This datatype was chosen because it allows easy integration with HOL4's built-in tools for recursive data structures and pattern matching. By using a recursive datatype, we ensure that the formalization is compatible with HOL4's proof automation tools, such as $Induct_on$, which are commonly used for reasoning about recursive functions and data structures.

6.3 Seamless Integration with HOL4's Existing Libraries

A key design consideration was ensuring that the AVL tree formalization could reuse existing libraries within HOL4, such as the arithmetic, list, and set theory libraries. Reusing these libraries reduces redundancy, simplifies the formalization process, and enables more robust proof automation.

For example, the existing list and set libraries in HOL4 were leveraged for defining and proving properties about the keys in an AVL tree. The 'keys' function, which retrieves the set of keys in an AVL tree, is integrated with HOL4's existing set theory library:

```
[keys_def]
keys Tip ≝ ∅
keys (Bin v₀ k v l r) ≝ {k} ∪ keys l ∪ keys r
```

This integration allows us to make use of HOL4's extensive set-theoretic reasoning capabilities, enabling efficient proofs about the contents of the tree, such as proving that a key exists in the tree after an insertion $keys_insert$ or that the set of keys is updated correctly after an operation.

By designing the formalization to interface with HOL4's existing libraries, we ensure that future work can build upon this foundation without needing to reimplement basic functionality. This modularity and reusability are key strengths of the HOL4 framework, and they were critical design considerations during the development of the AVL tree formalization.

6.4 Handling Recursive Functions and Theorem

Recursive functions are essential for formalizing AVL tree operations, such as height calculation, insertion, and balancing. In HOL4, recursive functions must be carefully defined to ensure that they terminate and are well-formed. The formalization of AVL trees makes extensive use of recursion to define operations like $insert_avl$, $balanceL$, and $balanceR$.

For example, the function $insert_avl$ is defined recursively to ensure that it can insert a new element into an AVL tree while preserving the AVL property. This function must handle both the base case (inserting into an empty tree) and the recursive case (inserting into a non-empty tree). To integrate this recursive function with HOL4's framework, termination must be guaranteed:

```
[insert_avl_def]
⊢ (∀x v. insert_avl x v Tip = singleton_avl x v) ∧
  ∀x v bf k kv l r.
    insert_avl x v (Bin bf k kv l r) =
    if x = k then Bin bf k kv l r
    else if x < k then balanceL k kv (insert_avl x v l) r
    else balanceR k kv l (insert_avl x v r)
```

In this recursive definition, the height of the tree decreases with each recursive call, which ensures that the recursion terminates. HOL4's support for reasoning about recursive functions allows us to prove that $insert_avl$ correctly maintains the AVL property and terminates for all inputs. This recursive approach is also extended to other operations, such as $balanceL$ and $balanceR$, which are used to restore the AVL property after an insertion.

6.5 Designing Efficient Proof Strategies

To verify the correctness of the AVL tree formalization, a series of theorems must be proven about the properties of AVL trees, such as their height, balance, and the set of keys. HOL4 provides various proof strategies, such as induction, case analysis, and automated tactics, to facilitate the proof process.

One of the key challenges in designing the formalization was choosing efficient proof strategies that could handle the complexity of recursive functions and the AVL property. For example, proving that the height of the tree is logarithmic with respect to the number of nodes requires an inductive proof that leverages the recursive structure of the AVL tree and its relationship with the Fibonacci sequence.

Another challenge was proving that the *balanceL* and *balanceR* functions correctly restore the AVL property after an insertion. These proofs require reasoning about the height of the subtrees and ensuring that the balance factor remains within the allowed range. The use of HOL4's automated proof tactics, such as $REWRITE_TAC$ and $INDUCT_TAC$, allowed these proofs to be constructed efficiently.

6.6 Exporting the Formalization

Once the formalization was completed, it was exported as part of HOL4's growing library of verified data structures. The export process ensures that the formalization can be easily reused and extended by other developers working within the HOL4 framework. By integrating the AVL tree formalization into HOL4's existing libraries, we ensure that future work can build upon this foundation, whether for educational purposes, research, or practical applications involving verified data structures.

6.7 Summary of Design Adaptations

In summary, the formalization of AVL trees in HOL4 required several key design adaptations to fit seamlessly into HOL4's existing framework. These adaptations included choosing a suitable recursive datatype representation, integrating with existing HOL4 libraries, designing recursive functions with termination in mind, and employing efficient proof strategies to verify the correctness of AVL tree operations. By making these design choices, we ensure that the AVL tree formalization is both robust and reusable, providing a solid foundation for future work in HOL4.

7 Comparison with Existing Theories

7.1 Isabelle's AVL Tree Implementation

Isabelle/HOL [Isa22] provides a formalization of AVL trees that is widely regarded as one of the most complete and influential formalizations in the field of theorem proving. Isabelle's formalization of AVL trees follows a monolithic approach, where all the components—data type definitions, auxiliary functions, and theorems—are tightly integrated into one comprehensive theory. This contrasts with the more modular structure that HOL4 typically encourages, where components are often distributed across different theories for reusability and composability.

Datatype Representation in Isabelle In Isabelle, the AVL tree is defined as a recursive data structure using a datatype declaration. Here is the definition from Isabelle's formalization:

```
datatype (set_of: 'a) tree = ET | MKT 'a "'a tree" "'a tree" nat
```

This datatype closely mirrors the one used in HOL4 but includes an additional ‘nat‘ parameter in each node, which explicitly stores the height of the tree as part of the node's structure. This explicit height parameter differs from the HOL4 approach, where the height of a tree is computed recursively rather than being stored within the tree structure. While storing the height as part of the node structure simplifies some operations, such as balancing and checking AVL properties, it increases the complexity of maintaining the correct height values during tree mutations (insertions and deletions).

In HOL4, by contrast, the height is computed dynamically using a recursive function. The AVL tree in HOL4 is defined as follows:

```
Datatype :
  avl_tree = Tip | Bin int num 'a avl_tree avl_tree
End
```

This definition uses a balance factor (an integer representing the height difference between the left and right subtrees) instead of explicitly storing the height as in Isabelle. This design decision aligns with the traditional AVL tree structure and maintains the dynamic computation of tree properties rather than encoding them directly in the data structure.

Height Calculation The height of an AVL tree in Isabelle is calculated using a primitive recursive function:

```
primrec height :: "'a tree => nat" where
"height ET = 0" |
"height (MKT x l r h) = max (height l) (height r) + 1"
```

In HOL4, a similar function calculates the height of the tree, but it uses a recursive definition rather than reading the height from the tree's structure. The HOL4 height function is:

$$\texttt{height Tip} \stackrel{\text{def}}{=} 0$$
$$\texttt{height (Bin } h\ k\ v\ l\ r\texttt{)} \stackrel{\text{def}}{=} \texttt{MAX (height } l\texttt{) (height } r\texttt{)} + 1$$

While both functions ultimately compute the height of the tree, Isabelle's approach benefits from faster height lookups, as the height is stored in each node. However, this comes at the cost of maintaining this value during tree mutations, increasing the complexity of operations like insertion and deletion.

AVL Predicates Both Isabelle and HOL4 define a predicate to ensure that a tree satisfies the AVL property. In Isabelle, the AVL property is enforced as follows:

```
primrec avl :: "'a tree => bool" where
"avl ET = True" |
"avl (MKT x l r h) =
 ((height l = height r \/ height l = height r + 1 \/ height r = height l + 1) /\
  h = max (height l) (height r) + 1 /\ avl l /\ avl r)"
```

This function checks that the tree is balanced by ensuring the heights of the left and right subtrees differ by at most one, and it also verifies that the stored height value is consistent with the actual height of the tree.

In HOL4, the AVL property is similarly enforced, but using the balance factor instead of explicitly stored heights:

```
avl Tip = T /\
avl (Bin bf k v l r) =
 ((height l = height r \/ height l = height r + 1 \/ height r = height l + 1) /\
  bf = &height r - &height l /\ avl l /\ avl r)
```

Both implementations ensure that the AVL property is maintained, but the HOL4 version dynamically calculates the height difference between subtrees using the balance factor, while the Isabelle version relies on pre-computed height values.

Insertion and Balancing The insertion operation in Isabelle is more complex due to the explicit height field in each node. Here's the definition for inserting a new element into the tree:

```
primrec insert :: "'a::order => 'a tree => 'a tree" where
"insert x ET = MKT x ET ET 1" |
"insert x (MKT n l r h) =
   (if x=n
    then MKT n l r h
    else if x<n
      then mkt_bal_l n (insert x l) r
      else mkt_bal_r n l (insert x r))"
```

This function balances the tree after an insertion using two auxiliary functions, mkt_bal_l and mkt_bal_r, which handle left and right rotations, respectively. These balancing functions are more complex in Isabelle because they need to maintain the height field after rebalancing.

In HOL4, the insertion function does not need to maintain an explicit height field. Instead, it focuses on maintaining the balance factor, which simplifies the balancing operations:

```
insert_avl x v Tip = singleton_avl x v /\
insert_avl x v (Bin bf k kv l r) =
  if x = k then Bin bf k kv l r
  else if x < k then balanceL k kv (insert_avl x v l) r
  else balanceR k kv l (insert_avl x v r)
```

Here, 'balanceL' and 'balanceR' handle left and right rotations, ensuring the AVL property is restored after insertion. Since the height is calculated recursively, there is no need to explicitly update it during these operations.

Efficiency and Trade-offs The primary trade-off between the two implementations lies in the efficiency of height lookups and the complexity of maintaining balance. Isabelle's approach, which stores the height in each node, allows for faster lookups during insertion and deletion but increases the complexity of maintaining correct height values. In contrast, HOL4's implementation dynamically calculates the height and uses a balance factor, simplifying the maintenance of tree invariants but potentially making height lookups slightly slower.

Overall, the HOL4 implementation is more in line with the traditional AVL tree design, focusing on balance factors, while Isabelle's version opts for an explicit height field, which influences both performance and complexity.

Proof Techniques and Automation Isabelle's AVL tree formalization leverages Isabelle's powerful proof automation tools, such as 'simp' and 'auto', to simplify the proof of correctness for insertion and deletion operations. For example, the following lemma ensures that the AVL property is maintained after insertion:

```
theorem avl_insert_aux:
  assumes "avl t"
  shows "avl(insert x t)"
        "(height (insert x t) = height t \/ height (insert x t) = height t + 1)"
```

HOL4 employs similar proof strategies but relies more heavily on explicit recursion and case analysis to establish the same properties. The modularity of HOL4's proof tactics, such as rw (rewrite) and $induct_o n$, allows for fine-grained control over the proof process.

Summary of Differences In summary, Isabelle's AVL tree formalization takes a more explicit approach by storing the height of each tree node, which simplifies height lookups but complicates insertion and deletion operations. HOL4's formalization, on the other hand, maintains the traditional AVL tree structure by dynamically calculating heights and using a balance factor to ensure the tree remains balanced. Both approaches have their advantages and trade-offs, with Isabelle's approach favoring efficient height lookups and HOL4's approach favoring simplicity in tree mutations.

7.2 Balanced_bst in HOL4

In contrast to Isabelle's formalization of AVL trees, HOL4 [HOL] provides a modular and flexible formalization of balanced binary search trees (BSTs), known as `balanced_bst`. HOL4's approach is characterized by its focus on modularity, where different components such as balance operations, key comparison, and height management are separated into reusable theories. This modular design encourages compositional reasoning and reusability of core components across different theories.

Data Type Representation in HOL4 In HOL4, the `balanced_bst` type follows the typical structure of binary search trees, with nodes that store keys, values, and left and right subtrees. This differs from Isabelle's AVL tree formalization, which includes an explicit height parameter stored in each node. HOL4 avoids encoding height directly

into the datatype, opting instead for recursive functions to manage balancing operations and height computations.

The HOL4 definition for `balanced_bst` is as follows:

```
balanced_bst = Tip | Bin size key value balanced_bst balanced_bst
```

In this definition, the tree is either empty (`Tip`) or a binary node (`Bin`) that stores the size of the subtree, a key, a value, and references to left and right subtrees. Unlike Isabelle, which stores an explicit height value, HOL4 calculates properties such as height recursively as needed.

Balancing and Height Calculation HOL4's balancing functions dynamically compute the balance factor using the height difference between subtrees. This contrasts with Isabelle's approach, where the height is stored within each node.

The height of a tree in HOL4 is computed as follows:

```
height Tip = 0 /\ height (Bin _ _ _ l r) = MAX (height l) (height r) + 1
```

In this recursive definition, the height of an empty tree (`Tip`) is zero, while the height of a node is the maximum height of its subtrees plus one. This approach mirrors the classic AVL tree structure, maintaining balance based on dynamic height calculations rather than pre-stored height values.

Balancing Operations The balancing operations in HOL4 focus on restoring the AVL property by applying left and right rotations when needed. The `balance_L` and `balance_R` functions are used to rebalance the tree after an insertion or deletion.

```
balanceL k v l r = if height l > height r + 1 then rotateR k v l r else Bin _ k v l r
balanceR k v l r = if height r > height l + 1 then rotateL k v l r else Bin _ k v l r
```

These functions ensure that the balance factor remains within acceptable limits by rotating the tree when the height difference between left and right subtrees exceeds 1. Unlike Isabelle, which recalculates and stores the height explicitly in each node, HOL4's balancing is based solely on recursive height calculations.

Trade-offs and Design Considerations The primary difference between HOL4's and Isabelle's implementations lies in the handling of height. While Isabelle stores the height explicitly within each node, HOL4 opts for recursive height calculations. This decision simplifies the node structure in HOL4 but may lead to slightly slower height lookups compared to Isabelle's approach.

Moreover, HOL4's modular design promotes code reuse and flexibility, allowing developers to extend or adapt the core balanced tree structure without tightly coupling all components into a single monolithic theory. In contrast, Isabelle's formalization integrates all components into a single theory, which can be beneficial for comprehensive reasoning but may limit modularity.

Proof Techniques and Automation The proof strategies employed in HOL4 leverage its modularity and tactics such as `rw` (rewrite) and `induct_on` for case analysis and recursion. HOL4 requires more explicit handling of recursion and case distinctions compared to Isabelle, where powerful automation tools like `simp` and `auto` play a central role in proofs. The modularity of HOL4's proof system allows for precise control over individual components of the proof, making it a preferred choice for fine-grained formal reasoning.

Summary of Differences In summary, HOL4's `balanced_bst` formalization maintains the traditional AVL tree structure by dynamically calculating heights and focusing on balance factors. This contrasts with Isabelle's more explicit approach, where the height is stored within each node, allowing for faster height lookups but requiring more complex maintenance of tree properties. HOL4's design favors modularity, flexibility, and dynamic height computation, whereas Isabelle's approach favors efficiency in height lookups and monolithic integration.

8 Conclusion

This thesis presents a comprehensive formalization of AVL trees within the HOL4 theorem prover, contributing to the broader effort of enriching HOL4's library of verified data structures. The formalization covers key aspects of AVL trees, including recursive data structures, balance conditions, height constraints, and the associated balancing operations. Through this work, we have formalized the core properties that ensure AVL trees maintain their logarithmic height and efficiency for search, insertion, and deletion operations.

A central achievement of this thesis is the detailed specification and proof of AVL tree correctness, particularly focusing on the invariant that ensures the height difference between subtrees remains bounded. By formalizing the recursive relationships between nodes, this work reinforces the importance of AVL trees in guaranteeing optimal performance for balanced binary search trees.

In comparing the HOL4 formalization with other theorem proving environments such as Isabelle/HOL, we identified significant differences in the design decisions that impact efficiency, maintainability, and complexity. While Isabelle opts for an explicit height parameter stored within each node, HOL4 dynamically computes the height during recursive operations. This leads to a trade-off between the simplicity of the AVL tree structure and the performance of height lookups during rebalancing operations. The comparative analysis highlights both the strengths and limitations of each approach, providing insights into different formalization strategies.

This formalization is not only an academic exercise but also a practical contribution to the ongoing development of verified data structures in HOL4. The methods and techniques developed here can be extended to other self-balancing tree structures or more complex data structures, further enriching the HOL4 ecosystem. Future work can build on this foundation, integrating AVL trees into more advanced applications such as formalized compilers, automated reasoning tools, or verified databases.

In conclusion, this thesis advances the state of the art in formal verification by providing a rigorous, reusable, and modular formalization of AVL trees in HOL4. It

serves as a stepping stone for future developments, encouraging further exploration and refinement of formally verified data structures within theorem proving systems.

References

[AH14] Jeremy Avigad and John Harrison. Formally verified mathematics. *Communications of the ACM*, 57(4):66–75, 2014.

[G+08] Mike Gordon et al. A brief overview of hol4. *LNCS*, 2008.

[GM93] Michael J.C. Gordon and Tom F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Har09] John Harrison. Handbook of practical logic and automated reasoning. Cambridge University Press, 2009.

[HOL] HOL. Hol/examples/balanced_bst/balanced_mapscript.sml at kodwani_-dev · sineeha-kodwani/hol. Accessed: 2024-10-22.

[Isa22] Isabelle. Avl. theory · master · isabelle / afp · gitlab, 2022. Accessed: 2022-08-19.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, 2002.

Appendix: AVL_trees Theory

Parent Theories: comparison, alist, integerRing, int_arith, Omega, EVAL_numRing

8.1 Definitions

[avl_def]

```
⊢ (avl Tip ⟺ T) ∧
  ∀bf k v l r.
    avl (Bin bf k v l r) ⟺
    (height l = height r ∨ height l = height r + 1 ∨
     height r = height l + 1) ∧ bf = &height r − &height l ∧
    avl l ∧ avl r
```

[balanceL_def]

```
⊢ ∀k v l r.
    balanceL k v l r =
    if height l = height r + 2 then
      case l of
        Tip ⇒ tree k v l r
      | Bin v1 lk lv ll lr ⇒
```

```
      if height ll < height lr then
        case lr of
          Tip ⇒ tree lk lv ll (tree k v lr r)
        | Bin v6 lrn lrv lrl lrr ⇒
          tree lrn lrv (tree lk lv ll lrl) (tree k v lrr r)
      else tree lk lv ll (tree k v lr r)
  else tree k v l r
```

[balanceR_def]

```
⊢ ∀k v l r.
    balanceR k v l r =
    if height r = height l + 2 then
      case r of
        Tip ⇒ tree k v l r
      | Bin v1 rk rv rl rr ⇒
        if height rl > height rr then
          case rl of
            Tip ⇒ tree rk rv (tree k v l rl) rr
          | Bin v6 rln rlv rll rlr ⇒
            tree rln rlv (tree k v l rll) (tree rk rv rlr rr)
        else tree rk rv (tree k v l rl) rr
    else tree k v l r
```

[complete_avl_def]

```
⊢ complete_avl 0 = Tip ∧
  ∀n. complete_avl (SUC n) =
      Bin 0 0 ARB (complete_avl n) (complete_avl n)
```

[delete_avl_def]

```
⊢ (∀x. delete_avl x Tip = Tip) ∧
  ∀x bf k kv l r.
    delete_avl x (Bin bf k kv l r) =
    if x = k then
      case (l,r) of
        (Tip,Tip) ⇒ Tip
      | (Tip,Bin v27 v28 v29 v30 v31) ⇒ r
      | (Bin v12 v13 v14 v15 v16,Tip) ⇒ l
      | (Bin v12 v13 v14 v15 v16,Bin v37 v38 v39 v40 v41) ⇒
        (let
           (pred_k,pred_v,l') = remove_max l
         in
           balanceR pred_k pred_v l' r)
    else if x < k then balanceR k kv (delete_avl x l) r
    else balanceL k kv l (delete_avl x r)
```

[height_def]

```
⊢ height Tip = 0 ∧
  ∀h k v l r.
    height (Bin h k v l r) = MAX (height l) (height r) + 1
```

[insert_avl_def]

$$\vdash (\forall x\ v.\ \texttt{insert\_avl}\ x\ v\ \texttt{Tip} = \texttt{singleton\_avl}\ x\ v) \wedge$$
$$\forall x\ v\ bf\ k\ kv\ l\ r.$$
$$\texttt{insert\_avl}\ x\ v\ (\texttt{Bin}\ bf\ k\ kv\ l\ r) =$$
$$\textbf{if}\ x = k\ \textbf{then}\ \texttt{Bin}\ bf\ k\ kv\ l\ r$$
$$\textbf{else if}\ x < k\ \textbf{then}\ \texttt{balanceL}\ k\ kv\ (\texttt{insert\_avl}\ x\ v\ l)\ r$$
$$\textbf{else}\ \texttt{balanceR}\ k\ kv\ l\ (\texttt{insert\_avl}\ x\ v\ r)$$

[keys_def]

$$\vdash \texttt{keys Tip} = \{\} \wedge$$
$$\forall v_0\ k\ v\ l\ r.\ \texttt{keys (Bin}\ v_0\ k\ v\ l\ r) = \{k\} \cup \texttt{keys}\ l \cup \texttt{keys}\ r$$

[lookup_avl_def]

$$\vdash (\forall x.\ \texttt{lookup\_avl}\ x\ \texttt{Tip} = \texttt{NONE}) \wedge$$
$$\forall x\ v_0\ k\ kv\ l\ r.$$
$$\texttt{lookup\_avl}\ x\ (\texttt{Bin}\ v_0\ k\ kv\ l\ r) =$$
$$\textbf{if}\ x = k\ \textbf{then}\ \texttt{SOME}\ kv$$
$$\textbf{else if}\ x < k\ \textbf{then}\ \texttt{lookup\_avl}\ x\ l$$
$$\textbf{else}\ \texttt{lookup\_avl}\ x\ r$$

[minimal_avl_def]

$$\vdash \forall t.\ \texttt{minimal\_avl}\ t \iff$$
$$\texttt{avl}\ t \wedge$$
$$\forall t'.$$
$$\texttt{avl}\ t' \wedge \texttt{height}\ t' = \texttt{height}\ t \Rightarrow$$
$$\texttt{node\_count}\ t \leq \texttt{node\_count}\ t'$$

[N_def]

$$\vdash \forall k.\ \texttt{N}\ k =$$
$$\texttt{MIN\_SET (IMAGE node\_count}\ \{x\ \texttt{|}\ \texttt{height}\ x = k \wedge \texttt{avl}\ x\})$$

[node_count_def]

$$\vdash \texttt{node\_count Tip} = 0 \wedge$$
$$\forall bf\ k\ v\ l\ r.$$
$$\texttt{node\_count (Bin}\ bf\ k\ v\ l\ r) =$$
$$\texttt{node\_count}\ l + \texttt{node\_count}\ r + 1$$

[singleton_avl_def]

$$\vdash \forall k\ v.\ \texttt{singleton\_avl}\ k\ v = \texttt{Bin}\ 0\ k\ v\ \texttt{Tip Tip}$$

[tree_def]

$$\vdash \forall k\ v\ l\ r.\ \texttt{tree}\ k\ v\ l\ r = \texttt{Bin (\&height}\ r - \texttt{\&height}\ l)\ k\ v\ l\ r$$

8.2 Theorems

[avl_balL]

$\vdash \forall k\ v\ l\ r.$
avl $l \wedge$ avl $r \wedge$
(height l = height $r \vee$ height l = height $r + 1 \vee$
height r = height $l + 1 \vee$ height l = height $r + 2$) $\Rightarrow$
avl (balanceL $k\ v\ l\ r$)

[avl_balR]

$\vdash \forall k\ v\ l\ r.$
avl $l \wedge$ avl $r \wedge$
(height r = height $l \vee$ height r = height $l + 1 \vee$
height l = height $r + 1 \vee$ height r = height $l + 2$) $\Rightarrow$
avl (balanceR $k\ v\ l\ r$)

[avl_complete_avl]

$\vdash$ avl (complete_avl k) $\iff$ T

[avl_insert_aux]

$\vdash \forall k\ v\ t.$
avl $t \Rightarrow$
avl (insert_avl $k\ v\ t$) $\wedge$
(height (insert_avl $k\ v\ t$) = height $t \vee$
height (insert_avl $k\ v\ t$) = height $t + 1$)

[avl_t1]

$\vdash$ avl (Bin 0 3 3 Tip Tip)

[avl_t2]

$\vdash$ avl (Bin 1 3 3 Tip (Bin 0 5 5 Tip Tip))

[avl_t3]

$\vdash$ avl (Bin 0 3 3 (Bin 0 2 2 Tip Tip) (Bin 0 5 5 Tip Tip))

[avl_t4]

$\vdash$ avl
(Bin 1 3 3 (Bin 0 2 2 Tip Tip)
(Bin (-1) 5 5 (Bin 0 4 4 Tip Tip) Tip))

[avl_t5]

$\vdash$ avl
(Bin 1 3 3 (Bin 0 2 2 Tip Tip)
(Bin 0 5 5 (Bin 0 4 4 Tip Tip) (Bin 0 13 13 Tip Tip)))

[avl_t6]

$\vdash$ avl
(Bin 0 5 5
(Bin 0 3 3 (Bin 0 2 2 Tip Tip) (Bin 0 4 4 Tip Tip))
(Bin 1 13 13 Tip (Bin 0 14 14 Tip Tip)))

[avl_t7]

```
⊢ avl
    (Bin (−1) 5 5
       (Bin (−1) 3 3 (Bin (−1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
          (Bin 0 4 4 Tip Tip))
       (Bin 1 13 13 Tip (Bin 0 14 14 Tip Tip)))
```

[avl_tree_case_eq]

```
⊢ avl_tree_CASE x v f = v' ⟺
  x = Tip ∧ v = v' ∨
  ∃i n a1 a a0. x = Bin i n a1 a a0 ∧ f i n a1 a a0 = v'
```

[avl_tree_preserves_avl]

```
⊢ ∀l r k v.
    avl l ∧ avl r ∧
    (height l = height r ∨ height l = height r + 1 ∨
     height r = height l + 1) ⇒
    avl (tree k v l r)
```

[children_of_minimal_avl]

```
⊢ ∀bf k v l r.
    minimal_avl (Bin bf k v l r) ⇒
    minimal_avl l ∧ minimal_avl r
```

[complete_avl_compute]

```
⊢ complete_avl 0 = Tip ∧
  (∀n. complete_avl <..num comp'n..> =
       Bin 0 0 ARB (complete_avl (<..num comp'n..> − 1))
         (complete_avl (<..num comp'n..> − 1))) ∧
  ∀n. complete_avl <..num comp'n..> =
      Bin 0 0 ARB (complete_avl <..num comp'n..> )
        (complete_avl <..num comp'n..> )
```

[Fibonacci_def]

```
⊢ ∀n. Fibonacci n =
      if n = 0 then 0
      else if n = 1 then 1
      else Fibonacci (n − 1) + Fibonacci (n − 2)
```

[Fibonacci_ind]

```
⊢ ∀P. (∀n. (n ≠ 0 ∧ n ≠ 1 ⇒ P (n − 1)) ∧
           (n ≠ 0 ∧ n ≠ 1 ⇒ P (n − 2)) ⇒
           P n) ⇒
      ∀v. P v
```

[Fibonacci_mono]

```
⊢ ∀n. Fibonacci n ≤ Fibonacci (n + 1)
```

[Fibonacci_mono_transitive]

$\vdash \forall m\ n.\ m \leq n \Rightarrow$ Fibonacci $m \leq$ Fibonacci n

[Fibonacci_thm]

$\vdash \forall k.$ Fibonacci $(k + 2)$ = Fibonacci $(k + 1)$ + Fibonacci k

[height_balL]

$\vdash \forall k\ v\ l\ r.$
height l = height $r + 2 \wedge$ avl $l \wedge$ avl $r \Rightarrow$
height (balanceL $k\ v\ l\ r$) = height $r + 2 \vee$
height (balanceL $k\ v\ l\ r$) = height $r + 3$

[height_balL2]

$\vdash \forall k\ v\ l\ r.$
avl $l \wedge$ avl $r \wedge$ height $l \neq$ height $r + 2 \Rightarrow$
height (balanceL $k\ v\ l\ r$) = $1 +$ MAX (height l) (height r)

[height_balR]

$\vdash \forall k\ v\ l\ r.$
height r = height $l + 2 \wedge$ avl $l \wedge$ avl $r \Rightarrow$
height (balanceR $k\ v\ l\ r$) = height $l + 2 \vee$
height (balanceR $k\ v\ l\ r$) = height $l + 3$

[height_balR2]

$\vdash \forall k\ v\ l\ r.$
avl $l \wedge$ avl $r \wedge$ height $r \neq$ height $l + 2 \Rightarrow$
height (balanceR $k\ v\ l\ r$) = $1 +$ MAX (height l) (height r)

[height_complete_avl]

$\vdash$ height (complete_avl n) = n

[height_eq_0]

$\vdash$ (height t = 0 $\iff$ t = Tip) $\wedge$ (0 = height t $\iff$ t = Tip)

[height_of_minimal_avl_diff_1]

$\vdash \forall bf\ k\ v\ l\ r.$
minimal_avl (Bin $bf\ k\ v\ l\ r$) $\Rightarrow$
l = Tip $\wedge$ r = Tip $\vee$ height l = height $r + 1 \vee$
height r = height $l + 1$

[keys_balanceL]

$\vdash \forall k\ v\ t_1\ t_2.$
keys (balanceL $k\ v\ t_1\ t_2$) = $\{k\} \cup$ keys $t_1 \cup$ keys t_2

[keys_balanceR]

$\vdash \forall k\ v\ t_1\ t_2.$
keys (balanceR $k\ v\ t_1\ t_2$) = $\{k\} \cup$ keys $t_1 \cup$ keys t_2

[keys_insert]

```
⊢ ∀x v t. keys (insert_avl x v t) = keys t ∪ {x}
```

[lookup1]

```
⊢ lookup_avl 3
    (Bin 0 5 5
       (Bin (−1) 3 3 (Bin (−1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
          (Bin 0 4 4 Tip Tip))
       (Bin (−1) 13 13 (Bin 1 6 6 Tip (Bin 0 7 7 Tip Tip))
          (Bin 0 14 14 Tip Tip))) =
  SOME 3
```

[lookup2]

```
⊢ lookup_avl 14
    (Bin 0 5 5
       (Bin (−1) 3 3 (Bin (−1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
          (Bin 0 4 4 Tip Tip))
       (Bin (−1) 13 13 (Bin 1 6 6 Tip (Bin 0 7 7 Tip Tip))
          (Bin 0 14 14 Tip Tip))) =
  SOME 14
```

[lookup3]

```
⊢ lookup_avl 4
    (Bin 0 3 3 (Bin (−1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
       (Bin 0 7 7 (Bin 0 6 6 Tip Tip) (Bin 0 13 13 Tip Tip))) =
  NONE
```

[lookup4]

```
⊢ lookup_avl 6
    (Bin 0 5 5
       (Bin (−1) 3 3 (Bin (−1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
          (Bin 0 4 4 Tip Tip))
       (Bin (−1) 13 13 (Bin 1 6 6 Tip (Bin 0 7 7 Tip Tip))
          (Bin 0 14 14 Tip Tip))) =
  SOME 6
```

[lookup5]

```
⊢ lookup_avl 10
    (Bin 0 5 5
       (Bin (−1) 3 3 (Bin (−1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
          (Bin 0 4 4 Tip Tip))
       (Bin (−1) 13 13 (Bin 1 6 6 Tip (Bin 0 7 7 Tip Tip))
          (Bin 0 14 14 Tip Tip))) =
  NONE
```

[minimal_avl_exists]

```
⊢ ∀k. ∃t. minimal_avl t ∧ height t = k
```

[minimal_avl_l_is_avl]

$\vdash \forall t.$ minimal_avl $t \Rightarrow$ avl t

[minimal_avl_node_count]

$\vdash \forall k\ t.$ minimal_avl $t \wedge$ height $t = k \Rightarrow$ node_count $t =$ N k

[N_0]

$\vdash$ N $0 = 0$

[N_1]

$\vdash$ N $1 = 1$

[N_fibonacci_relation]

$\vdash \forall k.$ N $k =$ Fibonacci $(k + 2) - 1$

[N_k]

$\vdash \forall k.$ N $(k + 2) =$ N $(k + 1) +$ N $k + 1$

[remove_max_def]

$\vdash (\forall v_0\ v\ l\ k.$ remove_max (Bin $v_0\ k\ v\ l$ Tip) = $(k,v,l)) \wedge$
$\forall v_9\ v_8\ v_7\ v_{11}\ v_{10}\ v_1\ v\ l\ k.$
remove_max (Bin $v_1\ k\ v\ l$ (Bin $v_7\ v_8\ v_9\ v_{10}\ v_{11}$)) =
(**let**
(max_k, max_v, r') = remove_max (Bin $v_7\ v_8\ v_9\ v_{10}\ v_{11}$)
in
$(max_k, max_v,$balanceL $k\ v\ l\ r')$)

[remove_max_ind]

$\vdash \forall P.\ (\forall v_0\ k\ v\ l.\ P$ (Bin $v_0\ k\ v\ l$ Tip)) $\wedge$
$(\forall v_1\ k\ v\ l\ v_7\ v_8\ v_9\ v_{10}\ v_{11}.$
P (Bin $v_7\ v_8\ v_9\ v_{10}\ v_{11}) \Rightarrow$
P (Bin $v_1\ k\ v\ l$ (Bin $v_7\ v_8\ v_9\ v_{10}\ v_{11}$))) $\wedge\ P$ Tip $\Rightarrow$
$\forall v.\ P\ v$

[t1]

$\vdash$ insert_avl 3 3 Tip = Bin 0 3 3 Tip Tip

[t10]

$\vdash$ delete_avl 14
(Bin 0 5 5
(Bin (-1) 3 3 (Bin (-1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
(Bin 0 4 4 Tip Tip))
(Bin (-1) 13 13 (Bin 1 6 6 Tip (Bin 0 7 7 Tip Tip))
(Bin 0 14 14 Tip Tip))) =
Bin (-1) 5 5
(Bin (-1) 3 3 (Bin (-1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
(Bin 0 4 4 Tip Tip))
(Bin 0 7 7 (Bin 0 6 6 Tip Tip) (Bin 0 13 13 Tip Tip))

[t11]

```
⊢ delete_avl 5
    (Bin (−1) 5 5
       (Bin (−1) 3 3 (Bin (−1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
          (Bin 0 4 4 Tip Tip))
       (Bin 0 7 7 (Bin 0 6 6 Tip Tip) (Bin 0 13 13 Tip Tip))) =
  Bin 0 4 4 (Bin 0 2 2 (Bin 0 1 1 Tip Tip) (Bin 0 3 3 Tip Tip))
    (Bin 0 7 7 (Bin 0 6 6 Tip Tip) (Bin 0 13 13 Tip Tip))
```

[t12]

```
⊢ delete_avl 4
    (Bin 0 4 4
       (Bin 0 2 2 (Bin 0 1 1 Tip Tip) (Bin 0 3 3 Tip Tip))
       (Bin 0 7 7 (Bin 0 6 6 Tip Tip) (Bin 0 13 13 Tip Tip))) =
  Bin 0 3 3 (Bin (−1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
    (Bin 0 7 7 (Bin 0 6 6 Tip Tip) (Bin 0 13 13 Tip Tip))
```

[t2]

```
⊢ insert_avl 5 5 (insert_avl 3 3 Tip) =
  Bin 1 3 3 Tip (Bin 0 5 5 Tip Tip)
```

[t3]

```
⊢ insert_avl 2 2 (insert_avl 5 5 (insert_avl 3 3 Tip)) =
  Bin 0 3 3 (Bin 0 2 2 Tip Tip) (Bin 0 5 5 Tip Tip)
```

[t4]

```
⊢ insert_avl 4 4
    (insert_avl 2 2 (insert_avl 5 5 (insert_avl 3 3 Tip))) =
  Bin 1 3 3 (Bin 0 2 2 Tip Tip)
    (Bin (−1) 5 5 (Bin 0 4 4 Tip Tip) Tip)
```

[t5]

```
⊢ insert_avl 13 13
    (insert_avl 4 4
       (insert_avl 2 2 (insert_avl 5 5 (insert_avl 3 3 Tip)))) =
  Bin 1 3 3 (Bin 0 2 2 Tip Tip)
    (Bin 0 5 5 (Bin 0 4 4 Tip Tip) (Bin 0 13 13 Tip Tip))
```

[t6]

```
⊢ insert_avl 14 14
    (insert_avl 13 13
       (insert_avl 4 4
          (insert_avl 2 2 (insert_avl 5 5 (insert_avl 3 3 Tip))))) =
  Bin 0 5 5 (Bin 0 3 3 (Bin 0 2 2 Tip Tip) (Bin 0 4 4 Tip Tip))
    (Bin 1 13 13 Tip (Bin 0 14 14 Tip Tip))
```

[t7]

```
⊢ insert_avl 1 1
    (insert_avl 14 14
       (insert_avl 13 13
          (insert_avl 4 4
             (insert_avl 2 2
                (insert_avl 5 5 (insert_avl 3 3 Tip))))))) =
  Bin (−1) 5 5
    (Bin (−1) 3 3 (Bin (−1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
       (Bin 0 4 4 Tip Tip))
    (Bin 1 13 13 Tip (Bin 0 14 14 Tip Tip))
```

[t8]

```
⊢ insert_avl 6 6
    (insert_avl 1 1
       (insert_avl 14 14
          (insert_avl 13 13
             (insert_avl 4 4
                (insert_avl 2 2
                   (insert_avl 5 5 (insert_avl 3 3 Tip)))))))) =
  Bin (−1) 5 5
    (Bin (−1) 3 3 (Bin (−1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
       (Bin 0 4 4 Tip Tip))
    (Bin 0 13 13 (Bin 0 6 6 Tip Tip) (Bin 0 14 14 Tip Tip))
```

[t9]

```
⊢ insert_avl 7 7
    (insert_avl 6 6
       (insert_avl 1 1
          (insert_avl 14 14
             (insert_avl 13 13
                (insert_avl 4 4
                   (insert_avl 2 2
                      (insert_avl 5 5 (insert_avl 3 3 Tip)))))))) =
  Bin 0 5 5
    (Bin (−1) 3 3 (Bin (−1) 2 2 (Bin 0 1 1 Tip Tip) Tip)
       (Bin 0 4 4 Tip Tip))
    (Bin (−1) 13 13 (Bin 1 6 6 Tip (Bin 0 7 7 Tip Tip))
       (Bin 0 14 14 Tip Tip))
```

[trees_of_height_exist]

```
⊢ ∀k. ∃t. avl t ∧ height t = k
```