B+ Tree Formal Verification: Insertion

MATTHEW MEYER, EPFL
BODONG JIA, EPFL

The B+ tree is a fundamental data structure widely used in database and file systems to maintain ordered data and support efficient insertion, deletion, and search operations. Verifying the correctness of B+ tree operations is crucial for ensuring data integrity and performance. This project focuses on the formal verification of the insertion operation in B+ trees, utilizing stainless proof code to ensure that all properties of the B+ tree are preserved post-insertion. Our approach provides a robust proof for verifying data structure integrity.

1 INTRODUCTION

Data structures with logarithmic lookup complexities are crucial in systematic software design. B+ trees stand out as balanced search trees frequently used in databases and file systems to handle large datasets efficiently. In most commercial systems that rely on B+ trees, performance is paramount, leading to implementations predominantly in imperative languages like C/C++. Several efforts have already been made to verify these B+ tree systems[1, 3]. However, with the rise of blockchain and similar systems, the demands for data consistency, reliability, and transparency have significantly increased. Therefore, a purely functional B+tree implementation system for business will have a promising future.

To fill the gap in the verification of pure functional B+tree, we have implemented a purely functional B+ tree in Scala and first utilized the Stainless[2] framework to verify its lookup and insertion operations. Due to time constraints and the inherent complexity of multi-way trees, such as frequent node splitting and promotion, verifying a B+ tree is considerably more complex than verifying a red-black tree. Consequently, the deletion operation and range iterator have not yet been verified. This work aims to provide stronger safety and reliability insertion assurances for B+ trees in emerging systems.

2 BACKGROUND

2.1 B+ Tree Structure

The B+ tree consists of internal nodes and leaf nodes. Internal nodes guide the search, while leaf nodes store the actual data entries. A B+tree of order d satisfies the following properties:

(1) **Node Capacity**: Each internal node (except the root) contains between $\lceil d/2 \rceil$ and d children.
(2) **Leaf Nodes**: All leaf nodes are at the same depth, ensuring a balanced tree structure.
(3) **Sorted Keys**: Keys within nodes are maintained in a sorted order, facilitating efficient search operations.
(4) **Pointer Consistency**: Internal nodes store pointers to child nodes, maintaining the hierarchical structure of the tree.

These properties ensure that the tree remains balanced, providing consistent performance for search and update operations.

Authors' addresses: Matthew Meyer, EPFL; Bodong Jia, EPFL.

2.2 Fundamental Operations

The insertion operation involves locating the appropriate leaf node for the new key and inserting it in sorted order. If the leaf node exceeds its capacity, a split operation is performed, propagating changes up the tree as necessary to maintain balance.

2.2.1 Lookup. The lookup operation in a B+tree involves traversing the tree form the root to the appropriate leaf to locate a specific key. The process is as follows:

(1) **Start at the Root**: Begin the search at the root node of the tree.
(2) **Internal Node Traversal**: For each internal node encountered, perform a binary search to find the range that may contain the target key.
(3) **Child Selection**: Based on the search, select the appropriate child to continue the traversal.
(4) **Leaf Node Access**: Upon reaching a leaf node, perform a linear search to locate the key.

The efficiency of the lookup operation is attributed to the tree's balanced nature and the ordered arrangement of keys within nodes, resulting in $O(\log n)$ time complexity, where n is the number of keys in the tree.

2.2.2 Insertion. Insertion into a B+tree must maintain the tree's invariants to ensure its balanced structure. The insertion process involves the following steps:

(1) **Search for Insertion Point**: Traverse the tree to locate the appropriate leaf node where the new key should be inserted.
(2) **Key Insertion**: Insert the key in the correct position within the leaf node, maintaining the sorted order.
(3) **Node Split**: If the leaf node exceeds its capacity after insertion, split the node into two, dispersing the keys evenly and promoting the middle key to the parent node.
(4) **Propagate Splits**: If the parent node also exceeds its capacity due to the promoted key, recursively perform splits up the tree until the root is either adjusted or split, increasing the tree's height.

The formal verification ensures that each step of the insertion process preserves the B+tree properties, preventing structural anomalies such as unbalanced trees or inconsistent key ordering.

3 OUR IMPLEMENTATION

In this section we present the types we defined, and the main methods on them.

We start with the class `Tree` and two extensions of it `Leaf` and `Internal`. These are, as may be expected, classes for leaf nodes and internal nodes. Note that we are blurring the boundary between trees and nodes, as a node encodes all the information of the subtree rooted in it. When discussing our implementation and subsequent verification, we go back and forth between nodes and their associated subtrees somewhat interchangeably.

```
sealed abstract class Tree
case class Leaf(keys: List[BigInt], values: List[BigInt]) extends Tree {
    require(keys.size == values.size) // Ensures keys and values are always in sync
  }
  case class Internal(keys: List[BigInt], children: List[Tree]) extends Tree {
    require(keys.nonEmpty && children.size == keys.size + 1)
  }
```

We implement methods on `Tree`, as follows:

- `Tree.content: List[BigInt]` This function is used to prove the validity of `contains`. It returs the list of all keys in the `Tree`.
- `contains(tree: Tree, key: BigInt, isRoot: Boolean): Boolean` As its name suggests, this function tests whether a key is in `tree`. Its postcondition states that it returns `True` if and only if `key` is in `tree.content`.
- `Tree.size: BigInt` Returns the size of the tree, is ensured to be non-negative. This will be important in our discussion of invariants later on.
- `insert(tree: Tree, key: BigInt, value: BigInt, isRoot: Boolean): Tree` The main function we implement. It successfully returns a new B+ tree with the (key,value) pair inserted, and we ensure that all tree invariants (see next section) are preserved. We also write several helper functions and lemmas to support this.

4 VERIFICATION APPROACH

We wish to verify that insertion into the tree preserves certain invariants, listed below.

4.1 Tree invariants

4.1.1 Node sizes. In a B+ tree, the number of keys and children in nodes is strictly regulated. Here is a table showing the various rules that must be followed (note: n is the order of the tree):

	Max keys	**Min keys**	**Max children**	**Min children**
Root Node	$n-1$	1	n	$\lceil n/2 \rceil$
Non-root Node	$n-1$	$\lceil n/2 \rceil - 1$	n	$\lceil n/2 \rceil$
Leaf Node	$n-1$	$\lceil n/2 \rceil$	-	-

Table 1. Node size restrictions in a B+ tree

Moreover, for an internal node (i.e. not leaf), the number of children is exactly one more than the number of keys. All these conditions are captured in the function `isValidTree(t: Tree, isRoot: Boolean): Boolean`. This function has a postcondition that helped greatly with verifying this invariant: when it returns true, then all of the children of the input tree (assuming it is an internal node) are also valid trees.

4.1.2 Key ordering. In order to permit fast searching in the tree, it is important that the keys be ordered adequately. This property is also verified in `isValidTree`.

4.1.3 All branches have the same length. A B+ tree is only truly useful if all its branches have the same length, ensuring that the height, and search time, are in $O(\ln m)$, where m is the total number of nodes in the tree. In order to verify this, we implement the function `sameLengths(t: Tree, isRoot: Boolean): Boolean`. This function has a misleading name: it does not directly check if all the branches rooted in t have the same length - instead, it checks that all its children have the same height, and that this property also holds for said children (and thus, for all internal nodes below t). This is the definition of a tree being *good* given in definition 2. Now, we prove that this function returning true in fact implies that all branches have the same length.

Definition 1 (Good Node)**.** Given a node N, we say that N is *good* if all of its children nodes have the same height.

Definition 2 (Good Tree)**.** Given a tree T, we say that T is *good* if all of its nodes are good.

Definition 3 (Height of a tree)**.** The height h of a node N is given by

- $h(N) = 1$ if N is a leaf node
- $h(N) = 1 + \max_{\text{child} \in N} h(\text{child})$ otherwise.

For a tree T, we define its height $h(T)$ to be the height of the its root node.

Remark 1. Notice that if a N is good, all of its children (assuming they exist) must have height $h(N) - 1$.

Definition 4 (Length of a branch)**.** In a tree T, the length of a branch to a leaf node is given by the number of edges on the path connecting that leaf node to the root.

Remark 2. It is more traditional to define the height of a tree as the length of its longest branch, but the definitions here are more in line with our implementation.

Lemma 1 (Good trees have same-length branches)**.** *Let T be a good tree with height n. Then, all branches of T have the same length, and that length is in fact $n - 1$.*

Proof. We proceed by induction on the height of T. In the base case, T is a leaf, and the assertion is trivially true. Now, suppose that the lemma is true for good trees with height equal to n, and let T be a good tree with height $n + 1$. Consider $T_1, T_2, \ldots, T_r$ the subtrees rooted in the children of the root (i.e., the subtrees just below the root). As they are subtrees of T, all of their nodes are good, and they as trees are themselves good. Moreover, each of their heights must be n, by remark 1. By the induction hypothesis, all of their branches have the same length, and that length is precisely $n - 1$.
Now, consider any branch of T. It can be decomposed as an edge from the root of T to the root of one of $T_1, T_2, \ldots, T_r$, and a branch of that subtree, which as said before, has length $n - 1$. Thus, the entire branch has length n. This branch of T was chosen arbitrarily, so we are done. ■

4.2 Finding an adequate measure for recursion on trees

Recursively verifying properties on trees presented a major difficulty, that we struggled for a long time to solve: when doing top-down recursion on trees, we need a non-negative decreasing measure in order to prove program termination. It is natural to use the height of the tree as a measure, but we run into a problem: when defining the height, we need to prove certain postconditions, such as, notably, that the height of a tree is strictly greater than that of its subtrees, in order to claim that the measure is decreasing. However, in order to verify this, we need a measure! We thus end up in a sort of circular train of thought.
Our solution starts with defining the function `insertMeasure(t: Tree, isRoot: Boolean): BigInt`, which effectively returns the hight of the node t. We compute it in the following way: for each child of t, we compute its height, and return one plus the maximum between this value and `insertMeasure` applied to the tree composed of the keys and children of t with the first elements of these lists removed (with some special consideration for special cases, such as when `t.children` is empty for example). The observation that we made is that with this reasoning, we first explore the first child of t, the second, and so on. Thus, it makes sense to use a measure with a *lexicographic* order. That is, in a first coordinate, we measure how far along the node we are, going from right to left, and in the second, we see how far down the subtree we have explored. We implemented this as follows:

```
decreases( measureHelper(t), t.size ) //this is the measure we use,
//with measureHelper defined as:
```

```
def measureHelper(t: Tree): BigInt = {
    t match {
      case Internal(_, children) => children.length
      case Leaf(_,_) => BigInt(0)
    }
  }
```

This works, as it is indeed decreasing, and in defining `size`, we force it to be non-negative:

```
def size: BigInt = {

      this match {
      case Leaf(keys, _) => keys.size
      case Internal(_, children) => max(0,1 + children.map(_.size).foldLeft(BigInt(0))(_ + _))
    }
  }.ensuring(res => res >= 0)
```

Thus, `insertMeasure` looks like this, in the end:

```
def insertMeasure(t: Tree, isRoot: Boolean): BigInt = {
  require(isValidTree(t, isRoot))
  decreases(

      measureHelper(t), t.size
    )

  t match {
    case Leaf(keys, values) => BigInt(1)
    case Internal(keys, children) =>
      (keys, children) match {
        case (_, Nil()) => BigInt(1)
        case (keyss, Cons(head, tail)) =>
          val newKeys = if(keyss.length==1){Nil[BigInt]()}else{keys.init}
          val headMeasure = insertMeasure(head, false)
          val tempNode = Internal(newKeys, tail)
          max(headMeasure, insertMeasure(tempNode, isRoot))+1
      }
  }
}.ensuring(res => res >= 1  &&
 (!t.isInstanceOf[Internal] || t.asInstanceOf[Internal].children.forall(c => insertMeasure(c, false)
```

Notice the postcondition, which is crucial since a measure must be non-negative and strictly decreasing.

This was not the only instance in which we used this kind of reasoning - it was also called for in proving the postcondition of `sameLengths(t: Tree, isRoot: Boolean): Boolean`, which was that if it returns true, then it returns true for all of its children as well, which is a very powerful assertion.

4.3 Operation Verification

4.3.1 Other Helper Functions and Lemmas. Supporting the primary operations are helper functions and lemmas that reinforce tree invariants:

Lemma 2 (lengthsLemma)**.** *Ensures all children of an internal node have consistent insert measures.*

Proof. Given that the internal node ‘t‘ is valid and maintains the ‘sameLengths‘ invariant, each child ‘c‘ must have an ‘insertMeasure‘ exactly one less than that of ‘t‘. This ensures that the measure consistently decreases down the tree, maintaining the well-foundedness necessary for termination. ■

Lemma 3 (internalChildrenCountLemmaCorrect)**.** *Verifies that internal nodes have the correct number of children.*

Proof. By the definition of the ‘Internal‘ class, each internal node must have exactly one more child than the number of keys. This is enforced by the requirement ‘children.size == keys.size + 1‘ in the class constructor, ensuring structural integrity. ■

Lemma 4 (findPosition)**.** *Determines the appropriate position within a node's keys for insertion or search, ensuring ordered key placement.*

Proof. Since the keys within a node are sorted, ‘findPosition‘ recursively locates the correct insertion point by comparing the target key with existing keys. Each recursive call reduces the problem size by one, guaranteeing termination and maintaining the sorted order. ■

Lemma 5 (insertIntoSorted)**.** *Inserts a new key into a sorted list, maintaining order without duplicates.*

Proof. The function checks if the new key is less than the head of the list. If so, it inserts the key at that position. Otherwise, it recursively inserts the key into the tail of the list. This ensures that the resulting list remains sorted and contains the new key exactly once, as duplicates are prevented by the precondition. ■

4.3.2 Insertion Function. The `insert` function is responsible for adding new key-value pairs to the B+ tree while preserving all tree invariants, such as node size constraints and balanced branch lengths. The insertion process includes:

(1) **Validation:** Ensures the current tree meets the invariants using `isValidTree` and `sameLengths`. It also checks that `insertMeasure` is non-negative and that the value does not already exist in the tree's content.
(2) **Insertion Logic:**
   - **Leaf Node Handling:** If the target node is a `Leaf`:
     - *Direct Insertion:* If there is space (`keys.size < ORDER`), the pair is inserted using `insertIntoLeaf`.
     - *Node Splitting:* If full (`keys.size == ORDER`), `splitLeaf` is invoked to divide the leaf, maintaining balance and size constraints.

- **Internal Node Handling:** For `Internal` nodes, `findPosition` determines the appropriate child subtree. The function recursively calls `insert` on the selected child and then invokes `balanceInternal` to maintain structural invariants, handling necessary adjustments.

(3) **Post-Insertion Guarantees:** Ensures the resulting tree remains valid (`isValidTree`), branches maintain equal lengths (`sameLengths`), and `insertMeasure` remains non-negative.

4.3.3 BalanceInternal Operation. The `balanceInternal` function maintains the structural integrity of `Internal` nodes after insertion by:

(1) **Precondition Verification:** Ensures that the `Internal` node and the newly inserted child comply with the tree's invariants, including key order and child counts.
(2) **Balancing Logic:**
   - *Simple Merge:* If capacity allows (`node.keys.size < ORDER - 1`), performs a merge by integrating `splitKeys` and `splitChildren`, maintaining balance.
   - *Node Splitting:* If over capacity, splits the internal node by aggregating keys and children, determining a midpoint, and dividing into two new `Internal` nodes to preserve balance and size constraints.
(3) **Post-Balancing Guarantees:** Ensures the internal node structure remains valid (`isValidTree`) and branches retain equal lengths (`sameLengths`).

4.3.4 SplitLeaf Operation. The `splitLeaf` function handles overfull `Leaf` nodes by:

(1) **Precondition Verification:** Ensures the leaf adheres to invariants, including sorted keys, no duplicates, and the correct number of keys (`keys.size == ORDER`).
(2) **Splitting Logic:**
   - *Key Insertion and Sorting:* Inserts the new key into the sorted keys list using `insertIntoSorted`.
   - *Determining Split Point:* Calculates the midpoint (`mid`) based on the tree's order to divide keys and values evenly.
   - *Creating New Nodes:* Creates two new `Leaf` nodes from the split keys and values and forms an `Internal` node containing the middle key and the new leaves.
(3) **Post-Split Guarantees:** Ensures the resulting `Internal` node is structured correctly, each child leaf maintains integrity and size constraints, `insertMeasure` accurately reflects updated heights, and the newly inserted value is present in the tree's content.

Through these operations and supporting functions, the implementation achieves a robust, formally verified B+ tree structure, maintaining correctness of insertion operations.

5 CONCLUSION

Our verification successfully identifies and rectifies scenarios where tree properties could be violated. The proof code provides a relatively comprehensive approach to ensuring the correctness of B+ tree insertions.
What is novel in our approach is the functional and immutable implementation of such a data structure. By nature, it is more natural to do imperatively and with a mutable type, as done by [3], for example. We also learned how implementing and verifying immutable trees is quite challenging. Finding adequate measures was difficult, but pushed us to ask deep questions about recursively reasoning with tree structures.
Our work is only the beginning of what could be done. Possible overtures are:

- Implement and verify deletion: we had originally planned to do this, but realized we had underestimated the difficulty and time needed to do so, so we decided to focus on insertion. However, with the setup we have and the ideas we thought of while working on insertion, we feel confident that given more time, we could explore these avenues.
- Verify range iterator: Verifying a range iterator for a B+ tree is both challenging and meaningful[3]. The difficulty lies in ensuring that the iterator correctly traverses the elements in sorted order without missing any elements or revisiting them, even in the presence of concurrent modifications or complex tree structures. Successfully verifying this aspect is crucial, as it guarantees that the tree's iterators function reliably, which is essential for many applications relying on ordered data retrieval
- The general order case: In our code, the order of the tree is fixed, and for values other than 2 or 3, it takes a very large amount of time to run (`ORDER=2` takes 1 minute, `ORDER=3` takes 5 minutes, and for `ORDER=4` we stopped the program after an hour because of the verification complexity by now). One possible solution could involve an extra induction proof demonstrating that if the properties hold for order `N`, they also hold for order `N+1`. This would significantly reduce the computational effort required for higher orders by leveraging the established properties for smaller orders.

LINK TO OUR GITHUB REPOSITORY

https://github.com/Matthew2357/formal-verification

REFERENCES

[1] Aurèle Barrière. 2018. Verified B+ Trees: Formal Verification of Balanced Tree Algorithms. https://aurele-barriere.github.io/papers/vstbtrees.pdf Accessed: 2025-01-05.

[2] Stainless Contributors. 2024. Stainless: Verification Framework for Scala. Available at https://stainless.epfl.ch/.

[3] Niels Mündler and Tobias Nipkow. 2022. A Verified Implementation of B+-Trees in Isabelle/HOL. arXiv:2208.09066 [cs.LO] https://arxiv.org/abs/2208.09066