Final Report: Formal Verification of AVL trees

Filip Carevic
filip.carevic@epfl.ch

Natalija Mitic
natalija.mitic@epfl.ch

Radenko Pejic
radenko.pejic@epfl.ch

January 9, 2023

Abstract

In order to guarantee logarithmic time complexity of the basic *lookup*, *insert* and *delete* list operations, as well as solve in-memory sorts of sets and dictionaries, balanced trees were developed. One of the first balanced trees is the AVL tree [2], with a worst-case lookup, insertion, and deletion time of $O(logN)$, where N is the number of nodes in the tree. In this work, we implement and formally verify the aforementioned basic operations of AVL trees. Additionally, we expand the basic interface with formal verification of *join* and *split* operations, the fundamental operations for several set operations (e.g., union, intersection) defined on AVl trees. Our implementation is available at the following *github repository*[1].

1 Introduction

Standard list allocations (*sequential* and *linked* allocations) cannot guarantee equally good performances for *lookup*, *insert* and *delete(by value)* operations[2] - there is an intrinsic trade-off between linear and constant computation complexity. In order to achieve logarithmic time complexity for all three operations, *balanced tree* representations of a *linear list* were introduced. One of the first and most significant balanced trees is the AVL tree [2], with a worst-case lookup, insertion, and deletion time of $O(logN)$, where N is the number of nodes in the tree.

In this work, we aim to implement and prove correctness of *lookup*, *insert* and *delete* operations of AVL trees. In addition, we expand the basic interface with implementation and formal verification of *join* and *split* operations performed on AVL trees. The operations are implemented in Scala [5]

[1]URL to source code repository: `https://github.com/natalijamitic/cs-550-formal-verification/tree/main/Project`

[2]Note here that *delete by position* is another characteristic operation of linear lists. However, it has been neglected since it is irrelevant to our work.

programming language, while the formal verification is performed using the Stainless [4] verification tool.

2 Preliminaries

This section lists the fundamental concepts needed to comprehend the topic.

2.1 Definitions

We follow the definitions presented in [3].

Definition 2.1 (Binary tree)**.** A binary tree is a tree-type, non-linear data structure where each node has a maximum of two children.

Definition 2.2 (Binary search tree - BST)**.** A binary search tree (BST) is a binary tree where each node satisfies following properties:

1. the node value is larger than the left-child value
2. the node value is smaller than the right-child value

A prerequisite for this is the existence of a pre-defined ordering of elements (e.g., relation $\leq$ on natural numbers).

Definition 2.3 (Height)**.** The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree.

Definition 2.4 (Balance factor)**.** The balance factor, or just balance, of a node represents the difference between the height of the right and left child. Balance is positive if the right subtree is deeper than the left subtree.

Definition 2.5 (AVL)**.** A BST is defined as a balanced BST (i.e., AVL) if the heights of the left and right subtree of every node differ by not more than 1. In other words, a BST is balanced if the balance of every node is -1, 0 or 1.

2.1.1 Balancing operations

Certain operations on AVL trees (insertion and deletion) may violate the *balance* property of the tree. Therefore, **balancing operations** (i.e., rotations) need to be performed. We differentiate 4 cases of rotations:

1. **Right rotation**
   This rotation is needed when there is an imbalance due to the left child's left subtree. The setting in which this operation is performed is illustrated in Figure 1. Node A is imbalanced (i.e., balanced factor is equal to -2). Readjustment is accomplished as follows:
   - node B takes place of its parent node A

    - node A becomes the right child of node B
    - node A takes B's right subtree as its left subtree
2. **Left rotation** - symmetrical to the right rotation
3. **Left-right rotation (double rotation)**
   This rotation is needed when there is an imbalance due to the left child's right subtree. The setting in which this operation is performed is illustrated in Figure 2. Node A is imbalanced (i.e., balanced factor is equal to -2). The readjustment is performed as follows:
    - left rotation on node B - node B's balance becomes 0, node C's balance becomes -2
    - right rotation on node C - node C's balance becomes 0
4. **Right-left rotation (double rotation)** - symmetrical to the left-right rotation

Figure 1: Single rotation - right

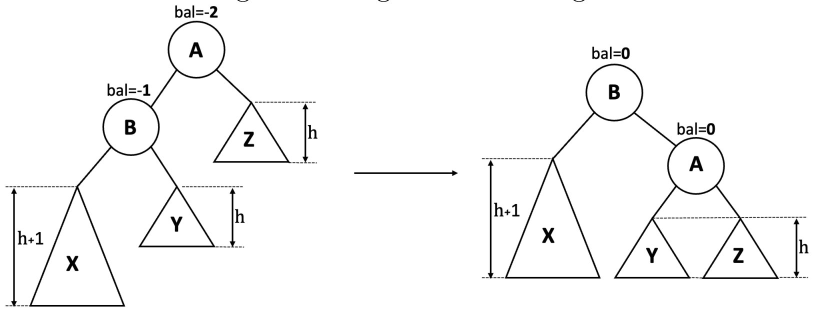
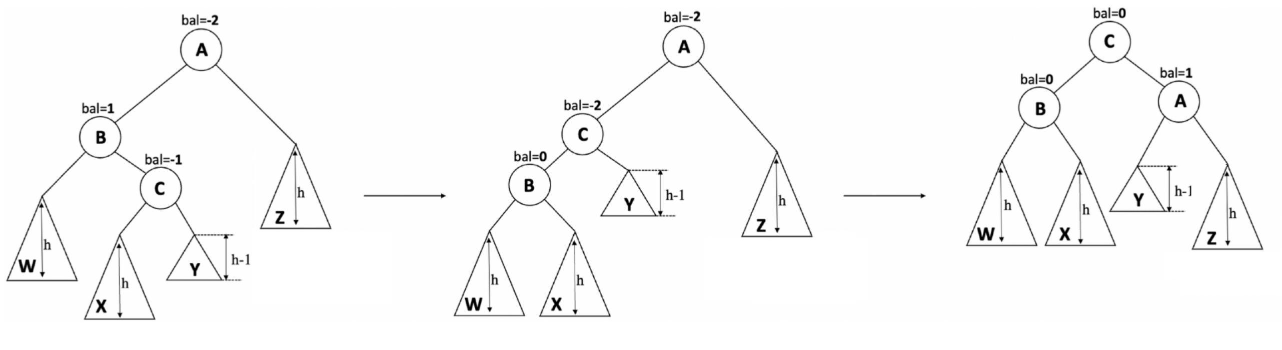


Figure 2: Double rotation - left + right



3 Implementation

In following subsections, we describe and analyse implementation of operations performed on AVL trees.

3.1 Tree class

The definitions used throughout our work are implemented in the sealed abstract class *Tree*. Apart from standard tree properties - **size** (i.e, the number of nodes in the tree), **isEmpty** and **height**, this class implements the following methods:

- **toSet** - This method constructs a set of all elements of type *BigInt* existing in the three.
- **toList** - This method constructs a list of all elements of type *BigInt* existing in the tree. The implementation of this function is based on the inorder tree traversal. After this function has been executed, it is guaranteed that the set of all elements in the obtained list is equal to set returned by the *toSet* function.
- **checkGreatest(v)** - This function returns an indicator if the value v (provided as a function argument) is greater than all of the elements in the tree. In particular, the function checks if the first order logic formula is satisfied: $\forall x \in dom(BigInt).x \in toSet \implies v > x$
- **checkSmallest(v)** - This function returns an indicator if the value v (provided as a function argument) is smaller than all of the elements in the tree. In particular, the function checks if the first order logic formula is satisfied: $\forall x \in dom(BigInt.x \in toSet \implies v < x$

. Lastly, **isBST**, **isBalanced** and **isAVL** methods implement corresponding definitions given in section 2.1. The pseudocode is given in Listing 1.

All functions implemented in the *Tree* class operate on the *this* object. Moreover, the properties of abstract class *Tree* are inherited by two concrete classes: *Empty* and *Node*, representing an empty and non-empty tree, respectively. It is important to note that the height of an empty tree is -1.

Remark: Our original plan was to include original definition of AVL tree in our analysis, but we encountered several challenges. Firstly, Stainless does not offer support for working with logarithms, which are prevalent in the analysis of this data structure. Secondly, Stainless has limited support for working with floating-point types. As a result, we had to follow an alternative definition of AVL trees in our analysis.

```
def isAVL: Boolean =
     isBalanced && isBST && right.isAVL && left.isAVL
def isBST: Boolean =
     left.checkGreatest(k) && left.isBST &&
     right.checkSmallest(k) && right.isBST
def isBalanced: Boolean =
    abs(left.height - right.height) <= 1
    && left.isBalanced && right.isBalanced
```

Listing 1: Main definitions

3.2 AVL implementation

An end-user can access the implementation of the AVL tree using ***AVLTree*** class that implements the ***AVLOperations*** interface and is supplied with a *tree:Tree* object. The *AVLOperations* interface specifies several methods that *AVLTree* must implement:

- **lookupAVL(tree, searched)**

This method returns an indicator if the *searched* element of type *BigInt* exist in the tree. The implementation recursively utilizes the ordering property of binary search tree. In other words, if the *searched* value does not mach the value v in the observed node, the lookup is continued in its left (right) subtree in case where the inequality $searched < v$ ($searched > v$) is satisfied. The only precondition of this function is that the tree is AVL. Finally, the return value of this operation is guaranteed to be true if the *searched* item exists in the list representation of the tree (i.e., *toList*), and false otherwise. The partial pseudocode is depicted in Listing 2.

```
def lookupAVL(tree: Tree, searched: BigInt): Boolean={
    require(tree.isAVL)
    ...
}.ensuring(isFound =>
    isFound == tree.toList.contains(searched))
```

Listing 2: Precondition and postcondition of lookupAVL operation

- **balanceLeft(value, left, right)**

The *balanceLeft* function is invoked in order to comply with the balancing property of AVL trees when the imbalance is caused by the left subtree. This function takes the following arguments: the *value* of the node that is violating the balancing property, as well as its left and right subtree. The return value is an AVL tree comprised of the *value*, left and right subtree elements. Moreover, *balanceLeft* can also be called on an already balanced tree in which case the left and right subtrees will be trivially combined with the *value*.

The preconditions for this operation are: (i) the left and right subtrees are AVL, (ii) the BST properties hold for *value* and left subtree, and *value* and right subtree, (iii) the absolute height difference between left and right subtree cannot exceed 1 or the left height is larger than right height by 2. After the execution of this function, it is ensured that: (i) the resulting tree is AVL, (ii) the set of elements in the resulting tree is equal to the union of *value* and elements in the left and right subtrees, (iii) the resulting tree has the height of either $max(left.height, right.height) + 1$ or $max(left.height, right.height)$. For a better overview of the pre- and post-conditions, see Listing 3.

The implementation of *balanceLeft* is based on the rotations explained in section 2. Both the single left rotation and the double right-left rotation are incorporated into this function. For the double-rotation proof, we utilize the *BST spreading to subparts* property. Specifically, take a look at the first part of figure 2, the spreading of the BST property can be conceptualized as

$$B.right.checkSmallest(B) \implies ((B < C) \textit{ and } X.checkSmallest(B) \textit{ and } Y.checkSmallest(B)).$$

```
def balanceLeft(n: BigInt, l:Tree, r:Tree): Tree = {
    require(
        l.checkGreatest(n) && left.isBST &&
        r.checkSmallest(n) && r.isBST &&
        ((abs(l.height - r.height) <= 1)
        || (l.height == r.height+2)))
    ...
}.ensuring (tree =>
    tree.isAVL  &&
    tree.toSet == union(l.toSet, n, r.toSet) &&
    (tree.size == l.size + r.size + 1)  &&
    (tree.height == max(l.height, r.height)+1 ||
    tree.height == max(l.height, r.height)))
```

Listing 3: Preconditions and postconditions of balanceLeft operation

- **balanceRight(value, left, right)**

The *balanceRight* function is analogous to *balanceLeft* and is used when an imbalance is caused by the right subtree.

- **insertAVL(tree, key)**

The insert operation of AVL trees is implemented in the *insertAVL* function (see Listing 4). Intuitively, this operation starts as a regular BST operation, by recursively traversing the tree and inserting the *key* as a leaf node. Considering that insertion can disrupt the balancing properties, balancing operations need to be performed.

The only argument of the *insertAVL* function is the value *key* of type *BigInt* that should be inserted into the tree. The insertion is unsuccessful if *key* is already in the tree.

The requirement for calling *insertAVL* is that the tree is an AVL tree, while after the execution of the function it is guarantees that: (i) the returned tree is AVL, (ii) the size of the returned size either remained the same or increased by 1, (iii) the height of the returned tree fluctuates by at most 1.

```
 def insertAVL(tree: Tree, key: BigInt): Tree = {
     require(tree.isAVL)
         ...
}.ensuring(newTree=> newTree.isAVL &&
     ((newTree.size == tree.size + 1) ||
     (newTree.size == tree.size)) &&
     abs(newTree.height - tree.height) <=1)
```

Listing 4: Preconditions and postconditions of insertAVL operation

- **deleteAVL(tree, key)**

The delete operation of AVL trees is implemented in the *deleteAVL* function. This function takes as an argument the value *key* to be deleted and returns the tree after the delete operation has been completed.

The only precondition for this function is that the tree is AVL. After the execution of this function, it is guaranteed that: (i) the resulting tree is AVL, (ii) the set of elements in the resulting tree is subset of the set of elements in the initial tree, (iii) if the operation was unsuccessful, the size of the resulting tree is equal to size of the initial tree, or smaller by 1 otherwise, and (iv) that the height property remains the same, or decreases by 1. It is important to note that the *deleteAVL* function is part of the public interface available to a client.

The implementation relies on two helper functions: *replaceRootWithMax* and *removeMax*. Intuitively, *deleteAVL* function locates and extracts the subtree with root node containing the *key* value. The extracted subtree is forwarded to *replaceRootWithMax* function, which, by leveraging helper function *removeMax*, locates and removes the node containing predecessor (with respect to inorder tree traversal) value *p* while replacing the root value of extracted subtree (i.e., the value *key*) with located value *p*. Lastly, each function performs balancing operation if required. The partial pseudocode depicting precondition and postcondition clauses is shown in Listing 5.

```
def deleteAVL(tree: Tree, key: BigInt): Tree = {
    require(tree.isAVL)
        ...
}.ensuring(res => res.isAVL &&
    (!tree.toSet.contains(key) ==>
    (res.size == tree.size)) &&
    (res.toSet.contains(key) ==>
    (res.size + 1 == tree.size)) &&
    res.toSet.subsetOf(tree.toSet) &&
    (tree.height == res.height ||
    tree.height == res.height+1) &&
    !res.toSet.contains(key))

def replaceRootWithMax(tree: Tree): Tree = {
    require(tree.size > 0 && tree.isAVL)
        ...
}.ensuring(newTree => newTree.isAVL &&
    (newTree.size + 1 == tree.size) &&
    newTree.toSet.subsetOf(tree.toSet) &&
    (tree.height == newTree.height ||
    tree.height == newTree.height + 1))

def removeMax(tree: Tree): (BigInt, Tree) = {
    require(tree.size > 0 && tree.isAVL)
    ...
}.ensuring((maxValue, leftTree) => leftTree.isAVL &&
    (leftTree.size + 1 == tree.size) &&
    (leftTree.height == tree.height ||
    leftTree.height == tree.height - 1) &&
    leftTree.checkGreatest(maxValue) &&
    tree.toSet.contains(maxValue) &&
    leftTree.toSet.subsetOf(tree.toSet))
```

Listing 5: Preconditions and postconditions of deleteAVL operation and related helper functions

- **joinAVL(t_1, k, t_2)**

The join operation of AVL trees has been implemented in *joinAVL* function. The function takes as arguments two AVL trees (t_1 and t_2) and a key k. The key k is used to used to join AVL trees $t1$ and $t2$. The function return the tree obtained after the join operation has been performed.

The *joinAVL* function requires that the value k is larger than all of the elements in the tree $t1$, and smaller than all of the elements in the tree $t2$. After the execution of this function, it is guaranteed that the resulting tree is AVL, as well as that the resulting tree is crafted using the elements provided in function arguments (i.e., trees t_1, t_2 and key k).

In its implementation, the function relies on the two symmetrical helper functions **joinLeftAVL** and **joinRightAVL**. The partial pseudocode is provided in the Listing 6.

Intuitively, the join operation works as follows: in the case where height difference between t_2 and t_1 is greater than 1, *join* function follows the left spine of t_2 until it reaches a node c such that the difference in heights of its subtree and $t2$ satisfy the balance criterion of AVL trees. A new node t^* is generated by adding t_1 as the left subtree, node c (and its descendants) as the right subtree, and setting key k to be the root value. Node t^* replaces the node c in $t2$. This could potentially disrupt the height properties of its ancestors, thus balancing operations have to be performed. It can be shown that at most one (singe or double) rotation is required. The case where the height difference between t_1 and t_2 is greater than 1 is symmetric.

Remark: In the edge cases, where absolute height difference between t_1 and t_2 is not greater than 1, *join* function returns a new tree with the root value of k left subtree t_1, and right subtree t_2.

```
def joinRightAVL(tl: Tree, k:BigInt, tr:Tree): Tree = {
    require(tl.size > 0 && tl.isAVL && tr.isAVL &&
    tl.checkGreatest(k) && tr.checkSmallest(k) &&
    tl.height > tr.height + 1)
    ...
}.ensuring(tree => tree.isAVL &&
    tree.height <= tl.height + 1 &&
    tree.height >= tl.height &&
    tree.height >= tr.height + 1 &&
    tree.size == tl.size + tr.size + 1 &&
    ((tl.toSet ++ tr.toSet)+ k) == tree.toSet)

def joinAVL(tl:Tree,k:BigInt,tr:Tree):Tree = {
    require(tl.size > 0 && tr.size > 0 &&
    tl.isAVL && tl.checkGreatest(k) &&
    tr.isAVL && tr.checkSmallest(k))
    ...
}.ensuring(tree => tree.isAVL &&
    tree.size == tl.size + tr.size + 1 &&
    tl.toSet.subsetOf(tree.toSet) &&
    tree.toSet.contains(k) &&
    tr.toSet.subsetOf(tree.toSet))
```

Listing 6: Preconditions and postconditions of joinAVL operation

- **splitAVL(t, k)**

The split operation of AVL trees is implemented in *splitAVL* function. In summary, given a value k, this operation splits AVL tree into two subtrees, t_1 and t_2, such that all of the elements in t_1 are smaller than k, while all of the elements in t_2 are greater than k.

The function takes as arguments tree t, and a value k used for the split. The function returns resulting trees t_1, t_2 and an indicator of success. The only precondition of this operation is that the tree t is AVL. After the execution of this function, it is guaranteed that: (i) resulting trees t_1 and t_2 are AVL trees, (ii) the value of k is greater than all of the elements in the t_1, and smaller than all of the elements in t_2, (iii) the elements in the resulting trees t_1 and t_2 are the subset of the elements in the initial tree t and (iv) in the case where the operation was successfully performed, the value of k exists in the initial tree t. The pseudocode is provided in Listing 7.

Remark: The implementation of split operation utilizes previously described *join* operation. For more details, refer to [1].

```
def splitAVL(tree:Tree, k:BigInt):(Tree,Boolean,Tree)={
    require(tree.isAVL)
    ...
}.ensuring((t1, success, t2)=>(t1.isAVL && t2.isAVL &&
    t1.checkGreatest(k) && t2.checkSmallest(k) &&
    t1.toSet.subsetOf(t.toSet) &&
    t2.toSet.subsetOf(t.toSet)) &&
    (success ==> t.toSet.contains(k)))
```

Listing 7: Preconditions and postconditions of splitAVL operation

4 Validation

In order to formally verify the implementation, we leveraged the Stainless verification tool (version 0.9.6) [4]. In particular, the validation has been performed using the following Stainless configuration:

1. *timeout*: 10 seconds
2. *solver*: nativez3

It requires approx. 36 seconds for the validation to be completed.

The complete **source code** for the AVL implementation and formal verification can be found on the following *github repository*[3].

[3]URL to source code repository: `https://github.com/natalijamitic/cs-550-formal-verification/tree/main/Project`

5 Conclusion and Future Work

In this paper, we presented and formally verified self-balancing AVL trees - a data structure that guarantees $O(logN)$ time complexity for insert, search and delete operations. Moreover, we expanded the basic AVL functionalities with the validation of *join* and *split* operations.

Split and join operations are fundamental building blocks for the implementation of several set operations defined on AVL trees: *union*, *intersection*, *set difference*. Implementation and formal verification of previously mentioned set operations we leave for future work.

The drawback of the current implementation is that it accepts only *BigInt* as the type of elements stored in the tree node. Thus, in future work, we plan to scale the framework to support arbitrary data types.

References

[1] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, jul 2016.

[2] Y.M. Landis G.M. Adelson-Velskii. An algorithm for the organization of information. *Doklady Akademia Nauk SSSR*, 1962.

[3] DE Knuth. Sorting and searching. third edn. volume 3 of the art of computer programming, 1997.

[4] EPFL IC LARA. Stainless: Formal verification for scala. *http://stainless.epfl.ch/*, 2019.

[5] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.