Vrije Universiteit Amsterdam

Bachelor Thesis

Logical Verification of AVL Trees in Lean

Author: Sofia Konovalova (2635220)

1st supervisor: Jasmin Blanchette
daily supervisor: Jannis Limperg

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

May 11, 2021

Contents

Chapter 1

Introduction

Chapter 2

Background

insert small description of the chapter

2.1 Lean Theorem Prover

2.2 AVL Trees

I begin by defining a binary search tree. A binary tree is a tree data structure where each node can have no more than two children. These two children are called the *left child* and the *right child* subtrees. In a *binary search tree* (BST), nodes are placed according to their key.

Where nodes are placed in a binary search tree is determined by what is often called the *binary search property*.

Definition 2.1 (Binary Search Property)**.** Given any node N in a binary search tree, all the keys in the left child subtree are smaller than that of N, and all keys in the right child subtree are greater than the key of N.

This allows for lookup and insertion to be done in $O \log n$ time in the worst case, as at any given node half of the tree is skipped.

Search, insertion and retrieval can be done recursively. Starting at the root node, the input key and the node key are compared: if the input key is smaller, the operation is done recursively on the left subtree; if the input key is larger, then the operation is done recursively on the right subtree.

An AVL tree is based on a binary search tree, with one very important distinction - it is *balanced*. To define what it means for a tree to be balanced, I will first define what the *height* of a tree is.

Definition 2.2 (Tree height)**.** The height of a tree is the length of the longest path from the root to a leaf.

Balance is reliant on this definition - an AVL tree is only balanced when the heights of any given left and right child subtrees does not differ by more than one [1]. By keeping balance, the structure ensures that there is a high ratio between the number of nodes in the tree and the height. This allows for retrieval and search operations to be done

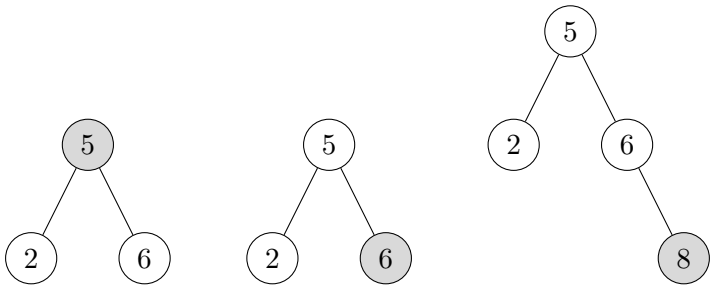

Figure 2.1: Example of an insertion operation with key 8

in $O(\log n)$ time in the worst case, with n being the amount of nodes in a tree [2]. A balancing factor can help define whether a tree is balanced or not.

Definition 2.3 (Balancing factor)**.** The balancing factor $BF(n)$ of a tree is the difference in height between the left child tree and the right child tree. A tree is defined to be an AVL tree (and therefore balanced) is the invariant $BF(n) \in \{-1, 0, 1\}$ holds for every node n in the tree.

During node insertion, the tree can become imbalanced, which can be mitigated with either a right rotation or a left rotation.

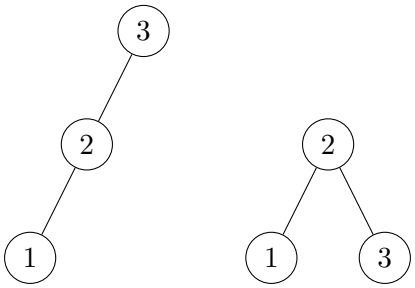

Figure 2.2: Example of a single right rotation

Chapter 3

Verification

This section will go through the verification of binary search trees and consequently AVL trees in Lean. First I will give some structural definitions for the tree and its properties, and then proceed with an explanation of the various proofs completed. The textbook released by the University of Pennsylvania *Verified Functional Algorithms* [3] and *Discrete Mathematics with a Computer* [2] were used as a reference point for some definitions and basic proof statements.

3.1 Binary Search Trees

Definition

I begin by defining the core of an AVL tree: a binary search tree. Even though Lean has a definition of a binary tree, `bin_tree`, the constructor does not include a left and right tree for a leaf node. In my implementation, shown in Figure 3.1, there is no constructor a leaf node, but there is no need since the left and right child trees of a node may be empty.

```
inductive btree (α : Type u)
| empty {} : btree
| node (l : btree) (k : nat) (a : α) (r : btree) : btree
```

Figure 3.1

I also create a definition for the binary search property as laid out in Definition 2.1. First a helper is defined, in order to express keys being smaller or larger than the root. This is shown in 3.2.

```
def forall_keys (p : nat → nat → Prop) : btree α → nat → Prop
| btree.empty x := tt
| (btree.node l k a r) x :=
  forall_keys l x ∧ (p x k) ∧ forall_keys r x
```

Figure 3.2

The binary search property can be laid out. An empty tree is a binary search tree why? and a non-empty tree is only a binary search tree if all the keys in the left child are smaller than the root and all keys in the right child are larger than the root, and if the left and right children satisfy the binary search property themselves. This is formalized in 3.3.

maybe explain why ordered is a definition and not inductive?

```
def ordered : btree α → Prop
| btree.empty := tt
| (btree.node l k a r) :=
  ordered l ∧ ordered r ∧ (forall_keys (>) l k) ∧ (forall_keys (<) r k)
```

Figure 3.3

Operations

There are three might add delete later basic operations on a binary search tree: insert, bound and lookup. Insert is straightforward; bound is checking if a key exists in the tree; and lookup is looking up a key in a binary search tree and returning the node value. The insert and lookup operations are similar; both compare the input key to the node key, and recursively perform the operation on the left or right subtree. The insert operation is shown in Figure 3.4.

```
def insert (x : nat) (a : α) : btree α → btree α
| btree.empty := btree.node btree.empty x a btree.empty
| (btree.node l k a' r) :=
  if x < k then btree.node (insert l) k a' r
  else if x > k then btree.node l k a' (insert r)
  else btree.node l x a r
```

Figure 3.4

explain the problem between the old bound and new bound

3.2 AVL Trees

Height and Balancing factor

I defined the height as per Definition 2.2. add more?

Instead of defining a balancing factor as a number, I defined balancing factor as a function that returns a boolean. If a node is not empty, to fit within Definition 2.3 then the difference in heights should not be less than 1 or greater than 1. This can be expressed with a simple absolute value of the difference in heights. This definition is shown in Figure 3.6.

```
def height : btree α → int
| btree.empty := 0
| (btree.node l k a r) :=
  1 + (max (height l) (height r))
```

Figure 3.5

```
def balanced : btree α → bool
| btree.empty := tt
| (btree.node l k a r) :=
  abs (height l - height r) ≤ 1
```

Figure 3.6

Rotations

Operations

3.3 Proofs

Chapter 4

Discussion

4.1 Comparison to Coq Implementation

4.2 Usability

4.3 Future Work

Chapter 5

Conclusion

Bibliography

[1] G. Adelson-Velskiy and E. Landis, "An algorithm for the organization of information," in *Soviet Mathematics Doklady*, vol. 3, 1962, pp. 1259–1263.

[2] J. O'Donnel, C. Hall, and R. Page, *Discrete Mathematics with a Computer*. Springer-Verlag, 2006, ch. 12, pp. 312–354.

[3] A. W. Appel, "Software foundations volume 3: Verified functional algorithms," University of Pennsylvania. [Online], 2017.