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 24, 2021

Contents

1 Introduction **3**

2 Lean Theorem Prover **4**
2.1 Dependent Type Theory 4
2.2 Interacting with Lean 4

3 AVL Trees **5**
3.1 Binary Search Trees 5
3.2 Balance and Rotation 7

4 Conclusion **8**

Bibliography **9**

Chapter 1

Introduction

Chapter 2

Lean Theorem Prover

write introduction about chapter

2.1 Dependent Type Theory

Lean is based on a version of dependent type theory called *Calculus of Constructions*, which is a higher-order typed lambda calculus. This allows for Lean to have a hierarchy of non-cumulative universes and inductive types [1]. On the top of the hierarchy are universes, which are denoted by `Type u`, where `u` is the level. Variables can be defined as part of a universe. For example, `variables {α : Type u}` defines a variable in the universe `u`. At the very bottom of the hierarchy there is `Prop` where, as the name implies, all the propositions are. In Lean, types themselves are also made out of types. For example, types like `nat` or `bool` (which describe natural numbers and booleans, respectively) are objects of type `Type`.

2.2 Interacting with Lean

Chapter 3

AVL Trees

An AVL tree consists of two basic parts: a *binary search tree* (BST), and rotation algorithms that re-balance the tree when necessary. This chapter will give definitions of binary search trees and the definitions of rotation algorithms to re-balance the tree. say which textbooks were used extensively and why.

3.1 Binary Search Trees

Basics

I begin by defining a binary stree. 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. Each node in a binary tree has a key, and if needed a node value. Figure 3.1 shows the inductive type defined. The core Lean code already has a definition for a binary tree, `bin_tree`, but it does not have a definition for a tree being empty, and node values and keys are only accessible in the leaf nodes. say why this is inconvenient.

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

Figure 3.1

Ordering

In a 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 3.1 (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.

In order to formalize this definition, there first needs to be a definition of what it means for a key to be smaller than or larger than the rest of the keys in a child subtree. This is

done with a definition `forall_keys`, shown in Figure 3.2. This function takes a numerical relation `p` which is either $>$ or $<$, a binary tree `btree` and a key, and returns a proposition. In order for this proposition to be true, the relation has to hold in the left child subtree and the right child subtree, and between the two keys. This is not very good explanation, change it.

```
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 definition `ordered` uses `forall_keys` to finish the definition of a BST as per Definiton 3.1. A BST is only "ordered" - in other words, the binary search property holds for that tree - if all the keys in the left child subtree are smaller than the root, and all keys in the right child subtree are smaller than the root, and both of the child subtrees are ordered themselves. This definition can be seen in Figure 3.3.

```
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

Basic Operations

The binary search property (and consequently, and ordered binary tree) allows for 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 due to the relation between the root key and the input key.

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, where the root key becomes the right child node. The function definition of the insertion operation in Lean can be seen on Figure 3.4. The retrieval operation, `lookup`, is very similar, except the return value of the definition is the value of the node that is found.

Tree search is a bit different. On the one hand, it could be defined the same as `lookup` and `insert`, but this could present a problem during re-balancing. When a tree is re-balanced after an insert operation, the position of a particular node can change with respect to its immediate parent (and its parent could change as well). This could mean that the node, for example, wouldn't be in some right subtree but would now be in some left subtree. This would make proofs about key preservation after re-balance difficult. Therefore, I defined whether a key is `bound` in a tree in the simplest of terms: if it exists

```
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

in the right or in the left subtree. In this particular definition, I am not focused on whether the tree is ordered, or if the key is in the right place - I am focused on if they key exists at all. The definition can be seen in Figure 3.5.

```
def bound (x : nat) : btree α → bool
| btree.empty := ff
| (btree.node l k a r) :=
  x = k ∨ bound l ∨ bound r
```

Figure 3.5

3.2 Balance and Rotation

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 3.2 (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 [2]. 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 in $O(\log n)$ time in the worst case, with n being the amount of nodes in a tree [3]. A balancing factor can help define whether a tree is balanced or not.

Definition 3.3 (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. Both of these rotations can be formed from up to two compound rotations. For example, a right rotation can consists of a simple right rotation, or a simple left rotation followed by a right one, depending on the balance factor of the child tree and the child's subtrees.

Chapter 4

Conclusion

Bibliography

[1] J. Avigad, L. de Moura, and S. Kong, "Theorem proving in lean," 2021.

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

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