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 1, 2021

Contents

Chapter 1

Introduction

Chapter 2

Lean Theorem Prover

Chapter 3

AVL Trees

insert small description of the chapter

3.1 Binary Search 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 3.1.1 (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.

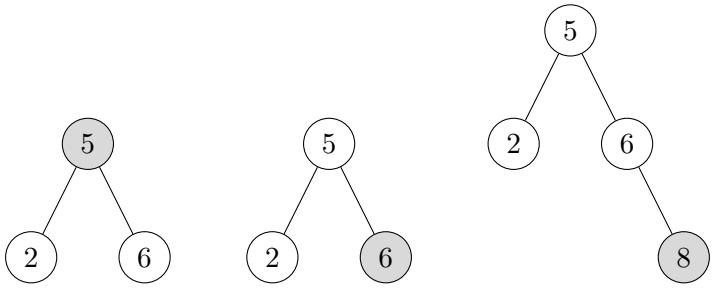

Figure 3.1: Example of an insertion operation with key 7

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 3.2.1 (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 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 3.2.2 (Balancing factor)**.** finish definition

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

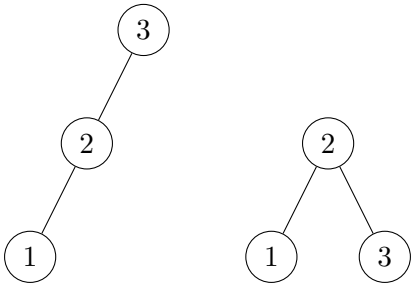

Figure 3.2: Example of a single right rotation

Chapter 4

Formalization

This chapter goes through the process of formalization of AVL search trees in Lean, with a description of the inductive types and functions that formalize the data structure. The full definitions can be found in Appendix A. The textbook *Discrete Mathematics with a Computer* was used extensively to help with a functional-programming definition of AVL constructs [2].

4.1 Definitions

Binary Search Trees

The first thing to defining an AVL tree is to define a binary search tree. A binary tree definition exists in Lean3 under the inductive type `bin_tree`, however the definition is incomplete for the purpose of binary search. The `bin_tree` constructor `leaf` is the only one that includes a definition for a key, and not the constructor dor `node`. We want access to subtrees and the key at any given node to define the binary search property, so we create our own definition for a binary tree. This definition can be seen in Figure 4.1. The type for the structure is inductive, which is useful for proofs as induction can be performed on any `btree`.

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

Figure 4.1

I also define the basic operations for a binary search tree: `insert`, `bound`, and `lookup` , shown in Figure 4.2. The definition of `bound` and `lookup` are similar, with bound not serving much reason except for using it in later proofs to show that node keys are preserved after other operations.

I also define the binary search property, as shown in Definition 3.1.1. For this I created a helper definition, `forall_keys`, which defines what exactly it means for all keys in a left or right subtree to be smaller or larger than the root key. This definition is made as abstract as possible, with the greater-than or less-than relation defined as just the

```
def lookup (x : nat) : btree α → option α
| btree.empty := none
| (btree.node l k a r) :=
  if x < k then lookup l
  else if x > k then lookup r
  else a

def bound (x : nat) : btree α → bool
| btree.empty := ff
| (btree.node l k a r) :=
  if x < k then bound l
  else if x > k then bound r
  else tt

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 4.2

parameter? variable? `p`, which is given a type of `nat → nat → Prop`, which is the type definition of the $>$ and $<$ relations in Lean. If the predicate is satisfied in the current node, as well as the left and right subtrees, then the predicate is satisfied in the whole tree. With this definition, the formalization of the binary search property is halfway done.

We call the definition for the binary search property `ordered`, as it is another name for trees where the property holds. It is also made recursive, as in our inductive `btree` definition, we only have direct definition of the left child and the right child, and not the children of those children. There is no guarantee that the grandchildren are ordered, so the definition takes this into account with the recursion: a binary tree can only be ordered if every single subtree is ordered. The complete definition can be seen in Figure 4.4.

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

Figure 4.3

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

Figure 4.4

Balancing

I define the height of a tree as per Definition 3.2.1 in Figure 4.5. Why is there a +1 in the definition?

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

Figure 4.5

Bibliography

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

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

Appendix A

Definitions

```
universe u

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

namespace btree
variables {α : Type u}

def empty_tree : btree α := btree.empty

def lookup (x : nat) : btree α → option α
| btree.empty := none
| (btree.node l k a r) :=
  if x < k then lookup l
  else if x > k then lookup r
  else a

def bound (x : nat) : btree α → bool
| btree.empty := ff
| (btree.node l k a r) :=
  if x < k then bound l
  else if x > k then bound r
  else tt

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
```

```
section ordering

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

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

end ordering

section balancing

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

def outLeft : btree α → bool
| btree.empty := ff
| (btree.node l k a r) :=
  match l with
  | btree.empty := ff
  | (btree.node ll lk la lr) :=
    (height ll ≥ height lr) ∧ (height ll ≤ height lr + 1) ∧
    (height lr ≥ height r) ∧ (height ll = height r + 1)
  end

def outRight : btree α → bool
| btree.empty := ff
| (btree.node l k a r) :=
  match r with
  | btree.empty := ff
  | (btree.node rl rk ra rr) :=
    (height rl ≤ height rr) ∧ (height rl ≤ height rr + 1) ∧
    (height rr ≥ height l) ∧ (height l + 1 = height rr)
  end

def easyR : btree α → btree α
| btree.empty := btree.empty
| (btree.node l k a r) :=
  match l with
    | btree.empty := (btree.node l k a r)
```

```
    | (btree.node ll lk la lr) := (btree.node ll lk la (btree.node lr k a
    r))
  end

def easyL : btree α → btree α
| btree.empty := btree.empty
| (btree.node l k a r) :=
  match r with
  | btree.empty := (btree.node l k a r)
  | (btree.node rl rk ra rr) := (btree.node (btree.node l k a rl) rk ra rr
    )
  end

def rotR : btree α → btree α
| btree.empty := btree.empty
| (btree.node l k a r) :=
  match l with
  | btree.empty := (btree.node l k a r)
  | (btree.node ll _ _ lr) :=
    if (height ll < height lr) then easyR (btree.node (easyL l) k a r)
    else easyR (btree.node l k a r)
  end

def rotL : btree α → btree α
| btree.empty := btree.empty
| (btree.node l k a r) :=
  match r with
  | btree.empty := btree.empty
  | (btree.node rl _ _ rr) :=
    if (height rr < height rl) then easyL (btree.node l k a (easyR r))
    else easyL (btree.node l k a r)
  end

end balancing

end btree
```