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

Chapter 1

Introduction

Chapter 2

Lean Theorem Prover

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.

The online textbook *Software Foundations Volume 3: Verified Functional Algorithms* [1] from the University of Pennsylvania was used for some of the basic definitions of the tree and operations, and the textbook *Discrete Mathematics with a Computer* [2] was used for definitions and basic proofs for re-balancing algorithms and deletion.

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 ref 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.

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 ref. 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.

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 ref.

Basic Operations

The binary search property (and consequently, and ordered binary tree) allows for for lookup and insertion to be done in $O(n)$ time in the worst case and $O\log(n)$ 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 ref. 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 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 ref.

3.2 Balance and Rotation

Height and Balance

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.

The definition for the height of a `btree` α can be seen in Figure ref. The definition is recursive and takes only the largest of the heights of the left child or the right child trees.

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 [3]. 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 3.3 (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.

In Lean, whether or not a tree is balanced is defined as a function with a tree as input and a boolean as output. The definition in Lean, shown on Figure ref, is slightly different that Definition ref, but it conveys the same theory. If the height of the left tree is larger than that of the right, then for the tree to be balanced, the height of the left tree should be less than the height of the right tree plus 1, and vice versa if the right tree is larger in height. If adding 1 to the height of the opposite child causes imbalance, that means the absolute value of the balance factor is more than 1 and therefore the tree is inbalanced.

Rotations

Before going through algorithms for tree rotation, the two cases for re-balancing are defined. A tree can either be left-heavy or right-heavy. A left-heavy is a tree whose left side is too tall - that is, the difference between the right and left child subtrees is more than 1. The vice versa is said for a right-heavy tree. There are two simple rotations that can be done to mitigate this imbalance - a left rotation (done on a right-heavy tree) and a right rotation (done on a left-heavy tree). terrible explanation, do better.

The definition of an left-heavy tree can be seen in Figure ref. The definition for a tree being right-heavy is mirrored. Explain definition a little bit.

There are two main rotations: left and right (performed on right-heavy and left-heavy trees, respectively). If we look at a right rotation, all that it accomplishes is that the rotated tree's root is the left subtree's data and the the right subtree is the subtree of the root node [2, p.333].

3.3 Deletion

Chapter 4

Comparison to Implementation in Coq

Chapter 5

Conclusion

Bibliography

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

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

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

Appendices

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}

inductive nonempty : btree α → Type u
| node : ∀ l k a r, nonempty (node l k a r)

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) :=
  x = k ∨ bound l ∨ bound r

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
```

```
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

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)

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

def balanced : btree α → bool
| btree.empty := tt
| (btree.node l k a r) :=
  if height l ≥ height r then height l ≤ height r + 1
  else height r ≤ height l + 1

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 r + 1 = height ll)
  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 rr ≥ height rl) ∧ (height rr ≤ height rl + 1) ∧
    (height rl ≤ height l) ∧ (height l + 1 = height rr)
  end

def easyR : btree α → btree α
| btree.empty := btree.empty
| (btree.node (btree.node ll lk la lr) k a r) := btree.node ll lk la (
   btree.node lr k a r)
```

```
| (btree.node l k a r) := btree.node l k a r

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

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 lk la 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.node l k a r)
  | (btree.node rl rk ra rr) :=
    if height rr < height rl then easyL (btree.node l k a (easyR r))
    else easyL (btree.node l k a r)
  end

def balance : btree α → btree α
| btree.empty := btree.empty
| (btree.node l k a r) :=
  if outLeft (btree.node l k a r) then rotR (btree.node (balance l) k a r)
  else if outRight (btree.node l k a r) then rotL (btree.node l k a (
   balance r))
  else btree.node l k a r

def insert_balanced (x : nat) (v : α) : btree α → btree α
| btree.empty := btree.node btree.empty x v btree.empty
| (btree.node l k a r) :=
  let nl := insert_balanced l in
  let nr := insert_balanced r in
  if x < k then
    if height nl > height r + 1 then rotR (btree.node nl k a r)
    else btree.node nl k a r
  else if x > k then
    if height nr > height l + 1 then rotL (btree.node l k a nr)
```

```
    else btree.node l k a nr
  else btree.node l x v r

-- def shrink : ∀ (t : btree α), nonempty t → (nat × α × btree α)
-- | _ (nonempty.node l k a btree.empty) := (k, a, l)
-- | _ (nonempty.node l k a (btree.node rl rk ra rr)) :=
--   let s := shrink (btree.node rl rk ra rr) (nonempty.node _ _ _ _) in
--   if height l > height s.2.2 + 1 then (s.1, s.2.1, rotR (btree.node l k
    a s.2.2))
--   else (s.1, s.2.1, (btree.node l k a s.2.2))

def shrink : btree α → option (nat × α × btree α)
| btree.empty := none
| (btree.node l k v r) :=
  match shrink r with
  | none := (k, v, l)
  | some (x, a, sh) :=
    if height l > height sh + 1 then (x, a, rotR (btree.node sh k v l))
    else (x, a, (btree.node sh k v l))
  end

def delRoot : btree α → btree α
| btree.empty := btree.empty
| (btree.node l k v r) :=
  match shrink l with
  | none := r
  | some (x, a, sh) :=
    if height r > height sh + 1 then rotL (btree.node sh x a r)
    else btree.node sh x a r
  end

def delete (x : nat) : btree α → btree α
| btree.empty := btree.empty
| (btree.node l k a r) :=
  let dl := delete l in
  let dr := delete r in
  if x = k then delRoot (btree.node l k a r)
  else if x < k then
    if height r > height dl + 1 then rotL (btree.node dl k a r)
    else btree.node dl k a r
  else if height l > height dr + 1 then rotR (btree.node l k a dr)
  else (btree.node l k a dr)

end btree
```