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

April 27, 2021

Contents

1 Introduction

2 Lean Theorem Prover

3 AVL Trees

give a small introduction to the section

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, 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.

This allows for lookup and insertion to be done in add complexity here time in the worst case, as at any given node half of the tree is skipped.

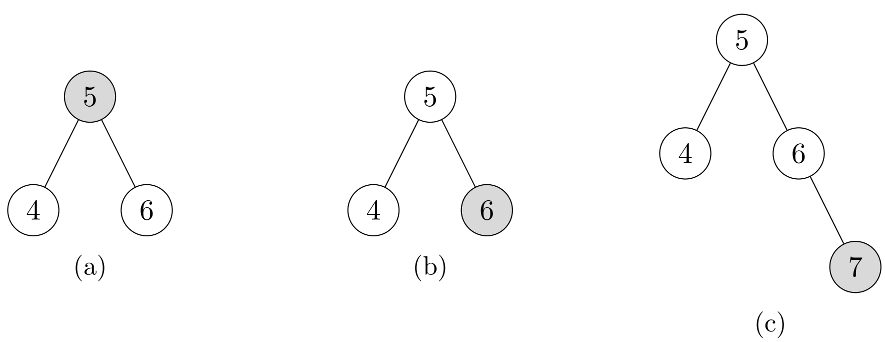

Figure 1: Insertion operation in a binary search tree.

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 1 shows a node with key 7 being inserted into a binary search tree. At 1(a), the new node is compared to the root. As $7 > 5$, the operation continues at the right subtree. At 1(b) the comparison is done again. As $7 > 6$, the operation continues at the right subtree. Because the node 6 doesn't have any children and $7 > 6$, a new right child node is created.

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)**.** get concrete definition of tree height.

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].

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

4 Formalization

I begin by formalizing a binary search tree. Lean already has a definition for a binary tree, `bin_tree`, but it does not fulfill the requirements for a search tree. I want to be able to have a key and a value for each tree node, and left and right child nodes (which may be empty). Figure 2 shows the new implementation of a binary tree.

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

Figure 2

Why using an inductive definition? Add lookup and bound definitions

I am also formalizing the binary search property as described in Definition 3.1, as the inductive definition above does not guarantee an ordered tree. For a search tree to be ordered means that all the keys in its left subtree are smaller than the root, and all the keys in its right child subtree are larger than the root. This also has to hold for all the subtrees in the structure.

```
  else btree.node l x a r

section ordering
```

Figure 3

explain what forall_keys is

```
| x btree.empty := tt
| x (btree.node l k a r) :=
  forall_keys x l ∧ (p x k) ∧ forall_keys x r
```

Figure 4

References

[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.

Appendices

A Definitions

```
universe u

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

-- def tostring : btree string → string
-- | btree.empty := "_"
-- | (btree.node l k a r) := "[" ++ " " ++ to_string k ++ " " ++ tostring
   l ++ " " ++ tostring r ++ " " ++ "]"

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

-- def bf : btree α → nat
-- | btree.empty := 0
-- | (btree.node l k a r) :=
--   height l - height r

-- def ex_tree1 : btree string := btree.node btree.empty 1 "a" (btree.node
    btree.empty 2 "b" (btree.node btree.empty 3 "c" btree.empty))

-- def ex_tree2 : btree string := btree.node (btree.node (btree.node btree
   .empty 1 "a" btree.empty) 2 "b" btree.empty) 3 "c" btree.empty

end balancing

end btree

-- inductive outLeft {α : Type u} : btree α → Prop
-- | empty {} : outLeft (btree.empty)
-- | node (xL xR zR : btree α) (x z : nat) (a d : α) :
--     (height xL ≥ height xR) →
--     (height xL ≤ height xR + 1) →
--     (height xR ≥ height zR) →
--     (height xL = height zR + 1) →
--     outLeft (btree.node (btree.node xL x a xR) z d zR)
```

```
-- def outLeft : btree α → bool
-- | btree.empty := ff
-- | (btree.node (btree.node xL x a xR) z d zR) :=
--   (height xL ≥ height xR) ∧ (height xL ≤ height xR + 1) ∧
--   (height xR ≥ height zR) ∧ (height xL = height zR + 1)
-- | (btree.node l k a r) := outLeft l

-- def outRight : btree α → bool
-- | btree.empty := ff
-- | (btree.node zL z d (btree.node yL y b yR)) :=
--   (height yL ≤ height yR) ∧ (height yL ≤ height yR + 1) ∧
--   (height yR ≥ height zL) ∧ (height zL + 1 = height yR)
-- | (btree.node l k a r) := ff

-- def easyR : btree α → btree α
-- | btree.empty := btree.empty
-- | (btree.node (btree.node xL x a xR) z d zR) :=
--   (btree.node xL x a (btree.node xR z d zR))
-- | (btree.node l k a r) := btree.node l k a r

-- def easyL : btree α → btree α
-- | btree.empty := btree.empty
-- | (btree.node zL z d (btree.node yL y b yR)) :=
--   (btree.node (btree.node zL z d yL) y b yR)
-- | (btree.node l k a r) := btree.node l k a r

-- def rotR : btree α → btree α
-- | btree.empty := btree.empty
-- | (btree.node (btree.node xL x a xR) z d zR) :=
--   if (height xL < height xR) then easyR (btree.node (easyL (btree.node
   xL x a xR)) z d zR)
--   else easyR (btree.node (btree.node xL x a xR) z d zR)
-- | (btree.node l k a r) := btree.node (rotR l) k a r

-- def rotL : btree α → btree α
-- | btree.empty := btree.empty
-- | (btree.node zL z d (btree.node yL y b yR)) :=
--   if (height yR < height yL) then easyL (btree.node zL z d (easyR (
   btree.node yL y b yR)))
--   else easyL (btree.node zL z d (btree.node yL y b yR))
-- | (btree.node l k a r) := btree.node l k a (rotL r)
```