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 21, 2021

Contents

1 AVL Trees

1.1 Binary Search Trees

First we begin by defining what exactly a binary tree is. A binary tree is a tree data structure in which a node has at most two children. These children are called the left child and the right child. We can define binary trees recursively using set notation: a non-empty binary tree is a tuple (L, S, R) where L and R are the left and right subtrees respectively, and S is a singleton set containing only the root node [1]. A binary search tree is a special case of a binary tree. In a binary search tree, all keys in the left subtree are lesser than of a node, and all keys in the right subtree are greater than of a node. This is sometimes called the *binary search property*. This allows for faster lookup and insertion, as the ordering means that half of the tree can be skipped, so lookup and insertion takes in the worst case $\log_2 n$ time, with n being the number of nodes in the tree.

Insertion and lookup can be done recursively or iteratively, where at each node a comparison is made between the new key and the key of the current node. If the new key is smaller, then we compare the left node. Otherwise, we compare with the right node.

AVL trees are built on a binary search trees. However, as AVL trees are *self-balancing*, rotation algorithms need to be defined to be used during certain cases of insertion.

1.2 AVL tree construction

Since AVL trees are based on binary search trees, there is not much that needs to be done in terms of construction. However, as mentioned previously, AVL trees are *self-balancing*. We begin by defining the *balancing factor*, which defines what it means for a tree to be balanced in the first place.

Definition 1.1 (Balancing factor)**.** The balancing factor $BF(N)$ of a node N is defined as the height difference between its two subtrees. A tree is an AVL tree if the invariant $BF(N) \in \{-1, 0, 1\}$ for all nodes in the tree.

In plain language, Definition 1.1 means that the difference in heights of two child subtrees cannot differ by more than one.

In general, read-only operations on AVL trees function the same as in binary search trees. Modifications are made in the insertion operations, where a change in the balancing factor needs to be observed and taken care of with rotation algorithms. These modifications are what make AVL trees *self-balancing*.

2 Formalization

2.1 Simple BST

The first step to formalizing an AVL tree was to formalize a simple binary search tree. Even though Lean has a definition of a basic binary tree `bin_tree`, that definition does not include a node key. For a binary search tree, this is important, so the inductive type for this formalization includes both a key and a value for each node.

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

Figure 1

Figure 1 shows the inductive type for the binary tree. This simple simple tree has two constructors: an empty binary tree, and a node with a left and right subtree, a key and a node value.

Figure 2 shows a simple lookup operation. As mentioned before, nodes in a binary search tree are ordered; all keys in the left child subtree are smaller than the root, and all keys in the right child subtree are larger than the root. Therefore when looking up a key in the tree, we need to do a comparison. If the key that is being looked up is smaller, then do a recursive lookup on the left subtree; if it is larger, then do a recursive lookup on the right subtree. If the key is neither larger nor smaller, than we have found the key.

The `bound` is very similar to `lookup`. The definition checks if a key exists in the tree.

```
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
```

Figure 2

```
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
```

Figure 3

The insertion operation also does key comparisons in order to pick which child subtree to enter the new node into. If the new key is smaller than the key of the current tree node, the insertion is done recursively on the left subtree; if the new key is larger, then insertion is done recursively on the right subtree.

References

[1] Garnier, R., and Taylor, J. *Discrete Mathematics:Proofs, Structures and Applications, Third Edition.* CRC Press, 2009.

Appendices

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

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) := ff

-- inductive outLeft' {α : Type} : btree α → Prop
-- | intro (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 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)

end balancing

end btree
```