# AVL Trees

D Oisín Kidney

July 28, 2018

**Abstract**

This is a verified implementation of AVL trees in Agda, taking ideas primarily from Conor McBride's paper "How to Keep Your Neighbours in Order" [2] and the Agda standard library [1].

## Contents



## 1 Introduction

First, some imports.

```
open import Relation.Binary
open import Relation.Binary.PropositionalEquality
open import Level using (Lift; lift; _⊔_; lower)
open import Data.Nat as ℕ using (ℕ; suc; zero; pred)
open import Data.Product
open import Data.Unit renaming (⊤ to 1)
open import Data.Maybe
open import Function
open import Data.Bool
open import Data.Empty renaming (⊥ to 0)
```

Next, We declare a module: the entirety of the following code is parameterized over the *key* type, and a strict total order on that key.

```
module AVL
  {k r} (Key : Set k)
  {_<_ : Rel Key r}
  (isStrictTotalOrder : IsStrictTotalOrder _≡_ _<_)
  where

  open IsStrictTotalOrder isStrictTotalOrder
```

## 2 Bounded

The basic idea of the verified implementation is to store in each leaf a proof that the upper and lower bounds of the trees to its left and right are ordered appropriately.

Accordingly, the tree type itself will have to have the upper and lower bounds in its indices. But what are the upper and lower bounds of a tree with no neighbours? To describe this case, we add lower and upper bounds to our key type.

```
module Bounded where

  infix 5 [_]

  data ⊤⊥ : Set k where
    ⊥ ⊤ : ⊤⊥
    [_] : (k : Key) → ⊤⊥
```

This type itself admits an ordering relation.

```
infix 4 _⊤⊥<_

_⊤⊥<_ : ⊤⊥ → ⊤⊥ → Set r
⊥     ⊤⊥< ⊥     = Lift r 0
⊥     ⊤⊥< ⊤     = Lift r 1
⊥     ⊤⊥< [ _ ] = Lift r 1
⊤     ⊤⊥< _     = Lift r 0
[ _ ] ⊤⊥< ⊥     = Lift r 0
```

```
[ _ ] ⊤⊥< ⊤     = Lift r 1
[ x ] ⊤⊥< [ y ] = x < y
```

Finally, we can describe a value as being “in bounds” like so.

```
infix 4 _<_<_

_<_<_ : ⊤⊥ → Key → ⊤⊥ → Set r
l < x < u = l ⊤⊥< [ x ] × [ x ] ⊤⊥< u
```

# 3 Balance

To describe the balance of the tree, we use the following type:

```
data ⟨_⊔_⟩≡_ : ℕ → ℕ → ℕ → Set where
  ⨬ : ∀ {n} → ⟨ suc n ⊔     n ⟩≡ suc n
  ⨪ : ∀ {n} → ⟨     n ⊔     n ⟩≡     n
  ⨫ : ∀ {n} → ⟨     n ⊔ suc n ⟩≡ suc n
```

The tree can be either left- or right-heavy (by one), or even. The indices of the type are phrased as a proof:

$$max(x, y) = z \tag{1}$$

The height of a tree is the maximum height of its two subtrees, plus one. Storing a proof of the maximum in this way will prove useful later.

We will also need some combinators for balance:

```
⨫⇒⨬ : ∀ {x y z} → ⟨ x ⊔ y ⟩≡ z → ⟨ z ⊔ x ⟩≡ z
⨫⇒⨬ ⨬ = ⨪
⨫⇒⨬ ⨪ = ⨪
⨫⇒⨬ ⨫ = ⨬

⨬⇒⨫ : ∀ {x y z} → ⟨ x ⊔ y ⟩≡ z → ⟨ y ⊔ z ⟩≡ z
⨬⇒⨫ ⨬ = ⨫
⨬⇒⨫ ⨪ = ⨪
⨬⇒⨫ ⨫ = ⨪
```

# 4 The Tree Type

The type itself is indexed by the lower and upper bounds, some value to store with the keys, and a height. In using the balance type defined earlier, we ensure that the children of a node cannot differ in height by more than 1. The bounds proofs also ensure that the tree must be ordered correctly.

```
data Tree {v} (V : Key → Set v) (l u : ⊤⊥) : ℕ → Set (k ⊔ v ⊔ r) where
  leaf  : (l<u : l ⊤⊥< u) → Tree V l u 0
  node : ∀ {h lh rh}
```

```
(k : Key)
(v : V k)
(bl : ⟨ lh ⊔ rh ⟩≡ h)
(lk : Tree V l [ k ] lh)
(ku : Tree V [ k ] u rh) →
Tree V l u (suc h)
```

## 5 Rotations

AVL trees are rebalanced by rotations: if, after an insert or deletion, the balance invariant has been violated, one of these rotations is performed as correction.

Before we implement the rotations, we need a type to describe a tree whose height may have changed:

```
Altered : ∀ {v} (V : Key → Set v) (l u : ⊤⊥) (n : ℕ) → Set (k ⊔ v ⊔ r)
Altered V l u n = ∃[ inc ] (Tree V l u (if inc then suc n else n))

pattern 0+ tr = false , tr
pattern 1+ tr = true , tr
```

### 5.1 Right Rotation

When the left subtree becomes too heavy, we rotate the tree to the right.

```
rotʳ : ∀ {lb ub rh v} {V : Key → Set v}
     → (k : Key)
     → V k
     → Tree V lb [ k ] (suc (suc rh))
     → Tree V [ k ] ub rh
     → Altered V lb ub (suc (suc rh))
```

This rotation comes in two varieties: single and double. Single rotation can be seen in figure 1.

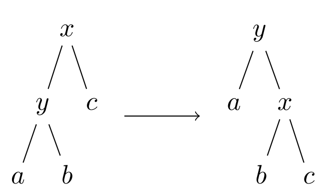

Figure 1: Single right-rotation

```
rotʳ x xv (node y yv ⟋ a b) c = 0+ (node y yv ⨪ a (node x xv ⨪ b c))
rotʳ x xv (node y yv ⨪ a b) c = 1+ (node y yv ⟍ a (node x xv ⟋ b c))
```

And double rotation in figure 2.

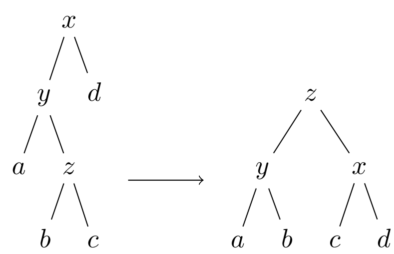

Figure 2: Double right-rotation

```
rotʳ x xv (node y yv ⋋ a (node z zv bl b c)) d =
  0+ (node z zv ⨪ (node y yv (⋋⇒⋌ bl) a b) (node x xv (⋌⇒⋋ bl) c d))
```

## 5.2 Left Rotation

Left-rotation is essentially the inverse of right.

```
rotˡ : ∀ {lb ub lh v} {V : Key → Set v}
     → (k : Key)
     → V k
     → Tree V lb [ k ] lh
     → Tree V [ k ] ub (suc (suc lh))
     → Altered V lb ub (suc (suc lh))
```

Single:

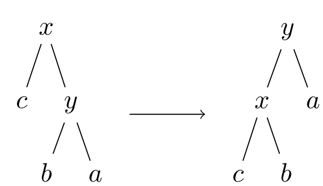

Figure 3: Single left-rotation

```
rotˡ x xv c (node y yv ⋋ b a) = 0+ (node y yv ⨪ (node x xv ⨪ c b) a)
rotˡ x xv c (node y yv ⨪ b a) = 1+ (node y yv ⋌ (node x xv ⋋ c b) a)
```

and double:

```
rotˡ x xv d (node y yv ⋌ (node z zv bl c b) a) =
  0+ (node z zv ⨪ (node x xv (⋋⇒⋌ bl) d c) (node y yv (⋌⇒⋋ bl) b a))
```

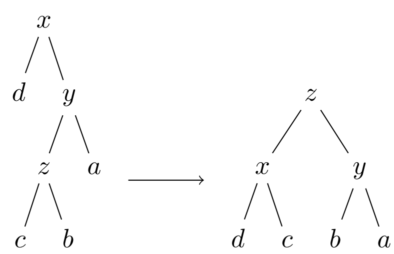


Figure 4: Double left-rotation

# 6 Insertion

After the rotations, insertion is relatively easy. We allow the caller to supply a combining function.

```
insert : ∀ {l u h v} {V : Key → Set v} (k : Key)
       → V k
       → (V k → V k → V k)
       → Tree V l u h
       → l < k < u
       → Altered V l u h
insert v vc f (leaf l<u) (l , u) = 1+ (node v vc ÷ (leaf l) (leaf u))
insert v vc f (node k kc bl tl tr) prf with compare v k
insert v vc f (node k kc bl tl tr) (l , _)
  | tri< a _ _ with insert v vc f tl (l , a)
... | 0+ tl′ = 0+ (node k kc bl tl′ tr)
... | 1+ tl′ with bl
...     | ⟋ = rot^r k kc tl′ tr
...     | ÷ = 1+ (node k kc ⟋ tl′ tr)
...     | ⟍ = 0+ (node k kc ÷ tl′ tr)
insert v vc f (node k kc bl tl tr) _
  | tri≈ _ refl _ = 0+ (node k (f vc kc) bl tl tr)
insert v vc f (node k kc bl tl tr) (_ , u)
  | tri> _ _ c with insert v vc f tr (c , u)
... | 0+ tr′ = 0+ (node k kc bl tl tr′)
... | 1+ tr′ with bl
...     | ⟋ = 0+ (node k kc ÷ tl tr′)
...     | ÷ = 1+ (node k kc ⟍ tl tr′)
...     | ⟍ = rot^l k kc tl tr′
```

# 7 Lookup

Lookup is also very simple. No invariants are needed here.

```
lookup : (k : Key)
       → ∀ {l u s v} {V : Key → Set v}
       → Tree V l u s
```

```
        → Maybe (V k)
lookup k (leaf l<u) = nothing
lookup k (node v vc _ tl tr) with compare k v
... | tri< _ _ _  = lookup k tl
... | tri≈ _ refl _ = just vc
... | tri> _ _ _  = lookup k tr
```

# 8 Deletion

Deletion is by far the most complex operation out of the three provided here. For deletion from a normal BST, you go to the node where the desired value is, perform an "uncons" operation on the right subtree, and use that to rebuild and rebalance the tree.

## 8.1 Uncons

First then, we need to define "uncons". We'll use a custom type as the return type from our uncons function, which stores the minimum element from the tree, and the rest of the tree:

```
record Cons {v}
            (V : Key → Set v)
            (lb ub : ⊤⊥)
            (h : ℕ) : Set (k ⊔ v ⊔ r) where
  constructor cons
  field
    head : Key
    val  : V head
    l<u  : lb ⊤⊥< [ head ]
    tail : Altered V [ head ] ub h
```

You'll notice it also stores a proof that the extracted element preserves the lower bound.

The uncons function itself is written in a continuation-passing style.

```
uncons : ∀ {lb ub h lh rh v} {V : Key → Set v}
       → (k : Key)
       → V k
       → ⟨ lh ⊔ rh ⟩≡ h
       → Tree V lb [ k ] lh
       → Tree V [ k ] ub rh
       → Cons V lb ub h
uncons k v bl tl tr = go k v bl tl tr id
  where
  go : ∀ {lb ub h lh rh v ub′ h′} {V : Key → Set v}
     → (k : Key)
     → V k
     → ⟨ lh ⊔ rh ⟩≡ h
     → Tree V lb [ k ] lh
```

```
    → Tree V [ k ] ub rh
    → (∀ {lb′} → Altered V [ lb′ ] ub h → Altered V [ lb′ ] ub′ h′)
    → Cons V lb ub′ h′
go k v ⨪ (leaf l<u) tr c = cons k v l<u (c (0+ tr))
go k v ⨪ (node kₗ vₗ blₗ tlₗ trₗ) tr c = go kₗ vₗ blₗ tlₗ trₗ
  λ { (0+ tl′) → c (1+ (node k v ⟍ tl′ tr))
    ; (1+ tl′) → c (1+ (node k v ⨪ tl′ tr)) }
go k v ⟍ (leaf l<u) tr c = cons k v l<u (c (0+ tr))
go k v ⟍ (node kₗ vₗ blₗ tlₗ trₗ) tr c = go kₗ vₗ blₗ tlₗ trₗ
  λ { (0+ tl′) → c (rotˡ k v tl′ tr)
    ; (1+ tl′) → c (1+ (node k v ⟍ tl′ tr)) }
go k v ⟋ (node kₗ vₗ blₗ tlₗ trₗ) tr c = go kₗ vₗ blₗ tlₗ trₗ
  λ { (0+ tl′) → c (0+ (node k v ⨪ tl′ tr))
    ; (1+ tl′) → c (1+ (node k v ⟋ tl′ tr))}
```

## 8.2 Widening

To join the two subtrees together after a deletion operation, we need to weaken (or widen) the bounds of the left tree. This is an $\mathcal{O}(\log n)$ operation.

For the widening, we'll need some properties on orderings:

```
x≮⊥ : ∀ {x} → x ⊤⊥< ⊥ → Lift r 0
x≮⊥ {⊥}     = lift ∘ lower
x≮⊥ {⊤}     = lift ∘ lower
x≮⊥ {[ _ ]} = lift ∘ lower

⊤⊥<-trans : ∀ {x y z} → x ⊤⊥< y → y ⊤⊥< z → x ⊤⊥< z
⊤⊥<-trans {⊥}     {y}     {⊥}     _        y<z = x≮⊥ {x = y} y<z
⊤⊥<-trans {⊥}     {_}     {⊤}     _        _   = _
⊤⊥<-trans {⊥}     {_}     {[ _ ]} _        _   = _
⊤⊥<-trans {⊤}     {_}     {_}     (lift ()) _
⊤⊥<-trans {[ _ ]} {y}     {⊥}     _        y<z = x≮⊥ {x = y} y<z
⊤⊥<-trans {[ _ ]} {_}     {⊤}     _        _   = _
⊤⊥<-trans {[ _ ]} {⊥}     {[ _ ]} (lift ()) _
⊤⊥<-trans {[ _ ]} {⊤}     {[ _ ]} _ (lift ())
⊤⊥<-trans {[ x ]} {[ y ]} {[ z ]} x<y y<z =
  IsStrictTotalOrder.trans isStrictTotalOrder x<y y<z
```

Finally, the widen function itself simply walks down the right branch of the tree until it hits a leaf.

```
widen : ∀ {lb ub ub′ h v} {V : Key → Set v}
  → ub ⊤⊥< ub′
  → Tree V lb ub h
  → Tree V lb ub′ h
widen {lb} ub<ub′ (leaf l<u) = leaf (⊤⊥<-trans {lb} l<u ub<ub′)
widen ub<ub′ (node k v bl tl tr) = node k v bl tl (widen ub<ub′ tr)
```

### 8.3 Full Deletion

The deletion function is by no means simple, but it does maintain the correct complexity bounds.

```
delete : ∀ {lb ub h v} {V : Key → Set v}
       → (k : Key)
       → Tree V lb ub (suc h)
       → Altered V lb ub h
delete k (node k₁ v bl tl tr) with compare k k₁
delete k (node {lh = zero}   k₁ v bl tl tr) | tri< _ _ _ = 1+ (node k₁ v bl tl tr)
delete k (node {lh = suc lh} k₁ v bl tl tr) | tri< _ _ _ with delete k tl | bl
... | 0+ tl′ | ◿ = 0+ (node k₁ v ▽ tl′ tr)
... | 0+ tl′ | ▽ = 1+ (node k₁ v ◺ tl′ tr)
... | 0+ tl′ | ◺ = rotˡ k₁ v tl′ tr
... | 1+ tl′ | _ = 1+ (node k₁ v bl tl′ tr)
delete {lb} k (node {rh = zero} k v bl tl (leaf k<ub)) | tri≈ _ refl _ with bl | tl
... | ◿ | _ = 0+ (widen k<ub tl)
... | ▽ | leaf lb<k = 0+ (leaf (⊤⊥<-trans {lb} lb<k k<ub))
delete k (node {rh = suc rh} k v bl tl (node kᵣ vᵣ blᵣ tlᵣ trᵣ)) | tri≈ _ refl _
  with bl | uncons kᵣ vᵣ blᵣ tlᵣ trᵣ
... | ◿ | cons k′ v′ l<u (0+ tr′) = rotʳ k′ v′ (widen l<u tl) tr′
... | ◿ | cons k′ v′ l<u (1+ tr′) = 1+ (node k′ v′ ◿ (widen l<u tl) tr′)
... | ▽ | cons k′ v′ l<u (0+ tr′) = 1+ (node k′ v′ ◿ (widen l<u tl) tr′)
... | ▽ | cons k′ v′ l<u (1+ tr′) = 1+ (node k′ v′ ▽ (widen l<u tl) tr′)
... | ◺ | cons k′ v′ l<u (0+ tr′) = 0+ (node k′ v′ ▽ (widen l<u tl) tr′)
... | ◺ | cons k′ v′ l<u (1+ tr′) = 1+ (node k′ v′ ◺ (widen l<u tl) tr′)
delete k (node {rh = zero} k₁ v bl tl tr) | tri> _ ¬b c = 1+ (node k₁ v bl tl tr)
delete k (node {rh = suc rh} k₁ v bl tl tr) | tri> _ ¬b c with delete k tr | bl
... | 0+ tr′ | ◿ = rotʳ k₁ v tl tr′
... | 0+ tr′ | ▽ = 1+ (node k₁ v ◿ tl tr′)
... | 0+ tr′ | ◺ = 0+ (node k₁ v ▽ tl tr′)
... | 1+ tr′ | _ = 1+ (node k₁ v bl tl tr′)
```

# 9 Packaging

Users don't need to be exposed to the indices on the full tree type: here, we package it in thee forms.

### 9.1 Dependent Map

```
module DependantMap where
  data Map {v} (V : Key → Set v) : Set (k ⊔ v ⊔ r) where
    tree : ∀ {h} → Bounded.Tree V Bounded.⊥ Bounded.⊤ h → Map V

  insertWith : ∀ {v} {V : Key → Set v} (k : Key)
             → V k
             → (V k → V k → V k)
```

```
          → Map V
          → Map V
insertWith k v f (tree tr) =
  tree (proj₂ (Bounded.insert k v f tr (lift tt , lift tt)))

insert : ∀ {v} {V : Key → Set v} (k : Key) → V k → Map V → Map V
insert k v = insertWith k v const

lookup : (k : Key) → ∀ {v} {V : Key → Set v} → Map V → Maybe (V k)
lookup k (tree tr) = Bounded.lookup k tr

delete : (k : Key) → ∀ {v} {V : Key → Set v} → Map V → Map V
delete k (tree {zero} tr) = tree tr
delete k (tree {suc h} tr) with (Bounded.delete k tr)
... | Bounded.0+ tr′ = tree tr′
... | Bounded.1+ tr′ = tree tr′
```

## 9.2 Non-Dependent (Simple) Map

```
module Map where
  data Map {v} (V : Set v) : Set (k ⊔ v ⊔ r) where
    tree : ∀ {h} → Bounded.Tree (const V) Bounded.⊥ Bounded.⊤ h → Map V

  insertWith : ∀ {v} {V : Set v} (k : Key) → V → (V → V → V) → Map V → Map V
  insertWith k v f (tree tr) =
    tree (proj₂ (Bounded.insert k v f tr (lift tt , lift tt)))

  insert : ∀ {v} {V : Set v} (k : Key) → V → Map V → Map V
  insert k v = insertWith k v const

  lookup : (k : Key) → ∀ {v} {V : Set v} → Map V → Maybe V
  lookup k (tree tr) = Bounded.lookup k tr

  delete : (k : Key) → ∀ {v} {V : Set v} → Map V → Map V
  delete k (tree {zero} tr) = tree tr
  delete k (tree {suc h} tr) with (Bounded.delete k tr)
  ... | Bounded.0+ tr′ = tree tr′
  ... | Bounded.1+ tr′ = tree tr′
```

## 9.3 Set

Note that we can't call the type itself "Set", as that's a reserved word in Agda.

```
module Sets where
  data ⟨Set⟩ : Set (k ⊔ r) where
    tree : ∀ {h} → Bounded.Tree (const 1) Bounded.⊥ Bounded.⊤ h → ⟨Set⟩

  insert : Key → ⟨Set⟩ → ⟨Set⟩
```

```
insert k (tree tr) =
  tree (proj₂ (Bounded.insert k tt const tr (lift tt , lift tt)))

member : Key → ⟨Set⟩ → Bool
member k (tree tr) = is-just (Bounded.lookup k tr)

delete : (k : Key) → ⟨Set⟩ → ⟨Set⟩
delete k (tree {zero} tr) = tree tr
delete k (tree {suc h} tr) with (Bounded.delete k tr)
... | Bounded.0+ tr′ = tree tr′
... | Bounded.1+ tr′ = tree tr′
```

# References


[1] N. A. Danielsson, "The Agda standard library." [Online]. Available: https://agda.github.io/agda-stdlib/README.html

[2] C. T. McBride, "How to Keep Your Neighbours in Order," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14. ACM, pp. 297–309. [Online]. Available: https://personal.cis.strath.ac.uk/conor.mcbride/pub/Pivotal.pdf