Trees in Ruby

Mark Armstrong

October 28, 2020

Contents

1 Introduction 1

2 Motivation 1

3 Wednesday: Leaf trees using subclasses 1

4 Wednesday: Leaf trees using a single class 3

5 Monday: Leaf trees using subclasses 4

6 Monday: Leaf trees using a single class 5

1 Introduction

These notes were created during the tutorials on October 26th and 28th.

2 Motivation

3 Wednesday: Leaf trees using subclasses

```ruby
class LeafTree

  # We shouldn't be able to initialize
  # a LeafTree directly, only one of its subclasses
  # (there might be a better way to implement
  # an "abstract" class like this in Ruby;
  # I am not certain.)
  def initialize; throw "Cannot construct a LeafTree directly"
      end

  # All LeafTree's should have a flatten
  # But we can't really write it until we know
  # what kind of LeafTree we have.
  # Just throw an exception if it's called
  # and we forgot to implement it in the subclass
  def flatten; throw "flatten not implemented!" end
end

class Branch < LeafTree
  attr_reader :left
  attr_reader :right

  def initialize(l,r)
    # First, check immediately if these are LeafTrees
    if l.is_a?(LeafTree) && r.is_a?(LeafTree)
      @left = l
      @right = r
    else
      throw "Constructing a branch with a non-leaftree"
    end
  end

  def flatten
    @left.flatten + @right.flatten
    # flatten(@left) will cause a complaint of "too many
        arguments"
    # flatten(@left) is implicitely called on "this" object,
    # (called `self` in Ruby) so we'd be giving an extra
        argument.
  end
end

class Leaf < LeafTree
  attr_reader :value

  def initialize(v)
    @value = v
  end

  def flatten; [@value] end
end

# Test it out.
x = Branch.new(Leaf.new(1),Leaf.new(2))
print x.flatten
```

4 Wednesday: Leaf trees using a single class

```ruby
class LeafTree
  attr_reader :kind
  attr_reader :value
  attr_reader :left
  attr_reader :right

  # In the constructor, the second parameter
  # defaults to nil. So if we are constructing a leaf,
  # we just don't give the second parameter.
  def initialize(x, y=nil)
    if y == nil
      @kind = :leaf
      @value = x
    else
      @kind = :branch
      @left = x
      @right = y
    end
  end

  def flatten
    if @kind == :leaf
      [@value]
    elsif @kind == :branch
      @left.flatten + @right.flatten
    else
      throw "Unknown kind of LeafTree"
    end
  end
end

x = LeafTree.new(LeafTree.new(1),LeafTree.new(2))
print x.flatten
```

5 Monday: Leaf trees using subclasses

```
class LeafTree
end

class Branch < LeafTree
  # Getters and setters for the left and right subtrees
  attr_reader :left
  attr_reader :right

  # Constructor; require two trees as parameters
  def initialize(l,r)
    if l.is_a?(LeafTree) && r.is_a?(LeafTree)
      @left = l
      @right = r
    else
      raise "Branch constructed with non-leaftrees"
    end
  end

  def flatten; @left.flatten + @right.flatten end
end

class Leaf < LeafTree
  attr_accessor :val

  def initialize(v)
    @val = v
  end

  def flatten; [@val] end
end
```

```
z = Branch.new(2,3)
y = Branch.new(Leaf.new(1), Leaf.new(2))
x = Branch.new(y,y)
print x.flatten
```

6 Monday: Leaf trees using a single class

```
class LeafTree
  # Type is either :leaf or :branch.
  attr_reader :type

  # If type is :leaf, val will be set to a value.
  attr_reader :val

  # If type is :branch, left and right will be the subtrees.
  attr_reader :left
  attr_reader :right

  def initialize(x, y=nil)
    if y == nil
      @type = :leaf
      @val = x
    else
      @type = :branch
      # Could check here that x and y are LeafTrees
      @left = x
      @right = y
    end
  end

  def flatten
    if @type == :leaf
      [@val]
    elsif @type == :branch
      @left.flatten + @right.flatten
    else
      raise "Unknown type of tree."
    end
  end
```

```
end

print LeafTree.new(LeafTree.new(1),LeafTree.new(2)).flatten
```