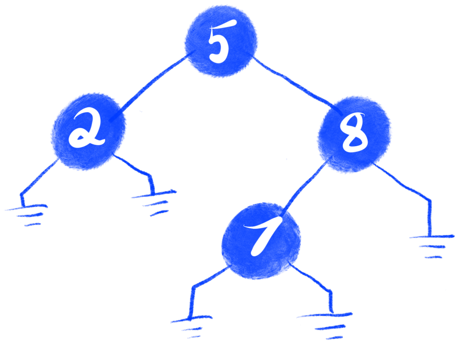
Effective Functional Programming
Correctness
Assignment 3
`Red-Black Trees`

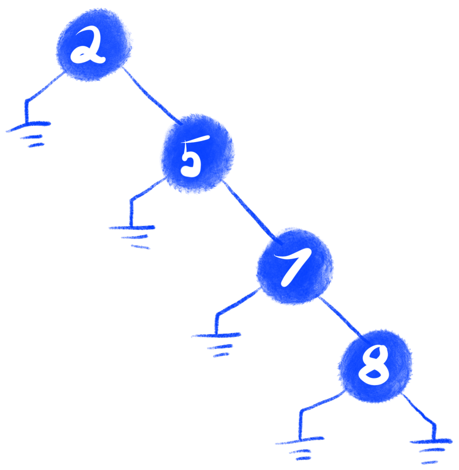
Paul Downen

Ordered trees are an important data structure, since they let us represent collections that can search for elements in *sub-linear* time; that is, without checking every single element exhaustively. For example, we can quickly confirm that the following binary tree



does not contain 9 by just checking the right-most path. If there was a 9 in the tree, it would be to the right of the 8. The nodes containing 2 and 7 do not even need to be checked at all, because we know they must be smaller than 5 and 8, respectively, which means they could not possibly contain the value 9.

However, *balancing* is an important property to make sure that ordered trees do actually provide sub-linear search. For example, the following is also a valid ordered tree,



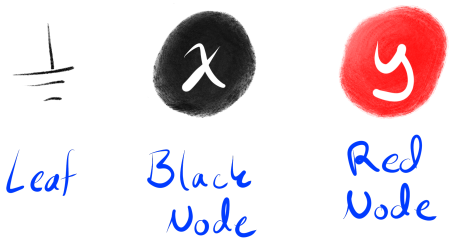
but it is no better than a linear list. The search for 9 in this tree is forced to visit every single node because it is *unbalanced*: some paths are very short (like the left-most path from 2, which immediately stops after one step) whereas some paths are very long (like the right-most path from 2, which has four steps).

One way of maintaining balance is to use a *red-black tree*, which is an ordinary binary tree but where every node is colored either red or black. Red-black trees can be represented in Haskell with the following data types:

```haskell
data Color = R | B
  deriving (Eq, Show)

data RBTree a = L | N Color (RBTree a) a (RBTree a)
  deriving (Show)
```

A `RBTree a` comes in effectively three different forms: a leaf `L`, a black node `N B` or a red node `N R`. Written graphically, these are:

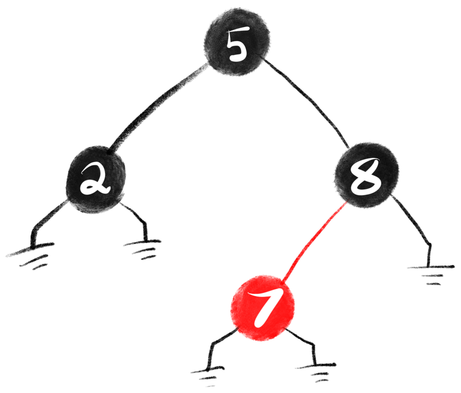


The color of a red-black tree is the color of its root node. In addition to marking nodes with a color, a proper red-black tree also meets the following properties:

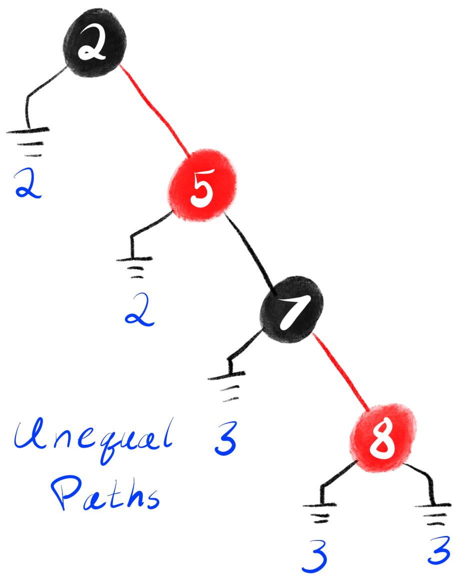
0. Ordered: The values of the nodes are in strict ascending order with respect to a left-to-right depth-first search. In other words, everything in the left sub-tree of a node is strictly less than the node value, and everything in the right sub-tree of a node is strictly greater than the node value.

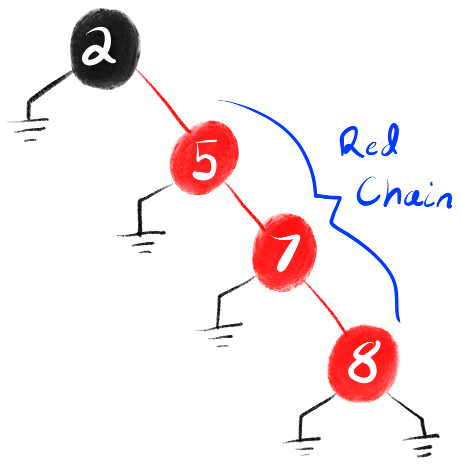
1. Black Roots: The root of every complete tree must be black. A leaf is considered black. Only sub-trees can be red.
2. No Red Chains: The left and right sub-trees of a red node must be black.
3. Black Paths: The number of black nodes contained in every path from the root to a leaf (including the root and leaf) must be equal.

Properties 1, 2, and 3 together force proper red-black trees to be balanced, since only balanced trees can follow these coloring criteria. For example, the balanced tree above has the following proper coloring (among others):



But the unbalanced tree cannot be colored properly. For example, here are two attempts that violate criteria 3 (black paths) and 2 (red chains) respectively:





Red-black trees are also a great candidate for a good testing suite, since they must follow several interesting (and somewhat complex) properties, and the fact that the implementation of tree operations preserve these properties is not obvious.

1 Red-Black Trees as Sets (35 points)

Exercise 1.1 (10 points)**.** Implement the function

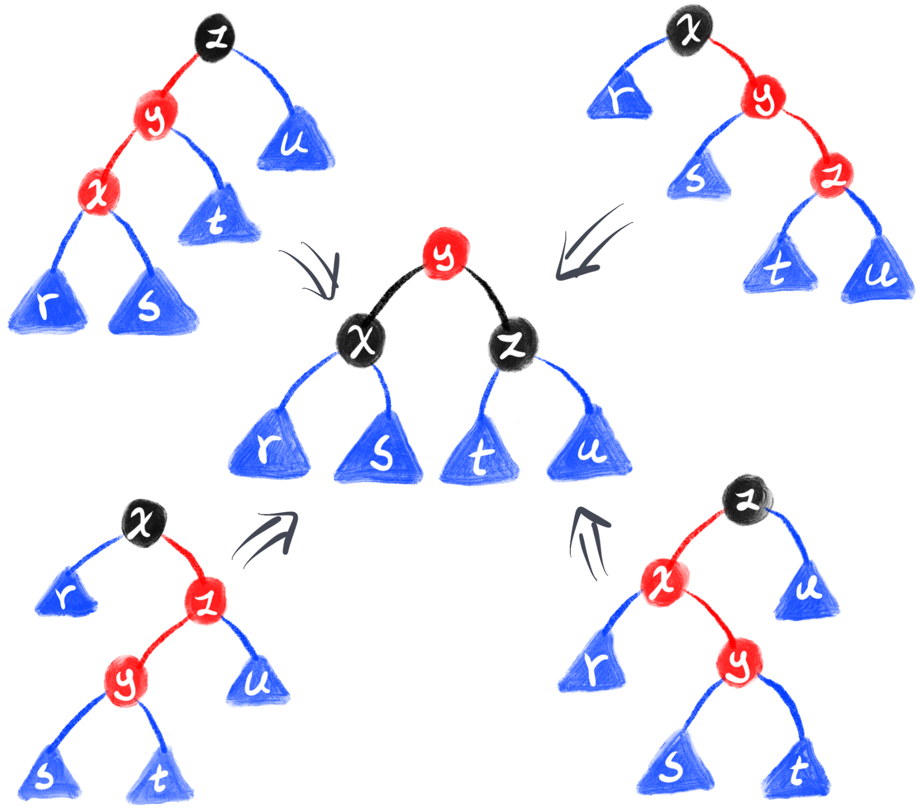
```
find :: Ord a => a -> RBTree a -> Maybe a
```

that searches for an element in the tree assuming that the given red-black tree is ordered. `find x t` should return `Just y` if an element `y` is found in `t` such that `x == y`, and return `Nothing` if no such element is found in `t`. When searching the `RBTree a`, at most *one* of the two sub-trees of a node should be searched; the other must be ignored.

Exercise 1.2 (10 points)**.** Implement the (non-recursive) functions

```
blackenRoot :: RBTree a -> RBTree a
balance     :: RBTree a -> RBTree a
```

`blackenRoot t` returns the same `RBTree a` as `t`, except that the root node is forced to be black. `balance` eliminates violations of the No Red Chain property by rearranging and recoloring the top part of the tree while preserving the order of the elements. The four interesting cases of `balance` are:



Each of the four cases start with a black root, and a chain of two red nodes descending in some direction from the root. Balancing each of these cases leads to a well-formed tree with a red root node. In all other cases, `balance` should just return the exact same `RBTree a` it was given.

Exercise 1.3 (15 points)**.** Implement the functions

```
ins    :: Ord a => a -> RBTree a -> RBTree a
insert :: Ord a => a -> RBTree a -> RBTree a
```

`ins x t` should insert the element `x` into the tree `t`. If `t` is a leaf, then `ins x t` should result in a tree with one red node and two leaves. If `t` is a node containing an element `y` that is equal to `x`, then `ins` should replace `y` with `x`. Otherwise, `ins` should recursively insert the element into one of the two sub-trees, picking the correct sub-tree to preserve ordering. In the recursive case where `ins` descends into one of the two sub-trees, then `ins` must `balance` the root of the tree being returned.

`insert` is exactly the same as `ins`, except that once the insertion operation is completely finished, the root node of the final tree is blackened, to satisfy property 1 (black roots). Note, that when calling `insert`, `balance` may be called many times (once each time a sub-tree is inserted into), but `blackRoot` should be called exactly once at the root of the tree.

2 Constructing and Inspecting Red-Black Trees (25 points)

Exercise 2.1 (10 points)**.** Implement the conversion functions

```
toList   ::          RBTree a -> [a]
fromList :: Ord a => [a]      -> RBTree a
```

The `fromList` function should produce an ordered, well-formed red-black tree, which requires the `Ord a` constraint.

Two red-black trees should be considered equal when they consist of the same elements, regardless of their internal tree structure. Implement an `Eq (RBTree a)` instance that compares the lists generated from two red-black trees.

Exercise 2.2 (5 points)**.** Implement the two query functions

```
rootColor  :: RBTree a -> Color
colorValue :: Color    -> Int
```

`rootColor` should return the color of the root node of the given tree (remember, leaves are considered black!). `colorValue` should return an `Int` corresponding to the value of a color, which can be used to count the weight of a path for checking the third red-black tree property (Black Paths). Since property 3 only counts black nodes and ignores red ones, the color black should be given a value 1 and red should be given a value of 0.

Exercise 2.3 (10 points)**.** A single path from the root of a red-black tree to one of the leaves can be represented by the type

```
type Path a = [(Color, a)]
```

Each node in the path (including the root) corresponds to a pair `(Color, a)` recording that node's color and element, and the leaf at the end of the path corresponds to the empty list.

Implement the function

```
paths :: RBTree a -> [Path a]
```

that converts a red-black tree into a list of *all* paths starting from the root and ending at one of the leaves.

3 Testing Red-Black Properties (30 points)

Exercise 3.1 (10 points)**.** Implement an `Arbitrary (RBTree a)` instance, assuming that the `Arbitrary a` and `Ord a` constraints hold, by defining the value

```
arbitrary :: (Arbitrary a, Ord a) => Gen (RBTree a)
```

of the `Arbitrary` type class.

Hint 3.1. The `fromList` function in section 2 might be helpful for defining `arbitrary`. Remember that you can get the value `b` out of a `Gen b` action like `arbitrary` by using a `do` expression to bind (with a statement using the left arrow `<-`) the returned result of that action.

Exercise 3.2 (10 points)**.** Implement the following tests of the red-black tree properties

```
inOrder        :: RBTree Int -> Bool
isBlackRoot    :: RBTree Int -> Bool
noRedChain     :: RBTree Int -> Bool
samePathValues :: RBTree Int -> Bool
```

that encode properties 0–3 of red-black trees as Haskell functions returning a boolean value: a `True` is returned if the given tree satisfies that property and a `False` is returned if the tree violates that property. Add each of these properties to a `main :: IO ()` test suit that tests them using `QuickCheck` or `Hspec`.

Exercise 3.3 (5 points)**.** Test that the properties defined in exercise 3.2 still hold after an `Int` is inserted into the tree and add them to your `main` test suit. For example, test `\x -> inOrder . insert x`, and so on for the other operations.

Exercise 3.4 (5 points)**.** Implement the following additional tests of red-black tree operations and add them to your test suit

```
roundTripSort   :: [Int] -> Bool
findAfterInsert :: Int -> RBTree Int -> Bool
```

`roundTripSort` should test that `toList (fromList xs)` is the same thing as sorting `xs` and removing duplicate elements for any `Int` list `xs`.

`findAfterInsert` should test that `find x (insert x t)` is `Just x` for any tree `t` and `Int x`.

Hint 3.2. The `nub :: Eq a => [a] -> [a]` function from the `Data.List` module removes duplicate elements from a list.

4 Proving Correctness With a Specification (45 points)

The purpose behind a red-black tree is to more efficiently implement a search (via the above `find` function) that scales logarithmically rather than linearly with the number of elements to search through. For example, doubling the number of elements in the red-black tree only adds a constant (some fixed number) of steps to `find`, since only a single path from the root of the tree to a leaf is searched, and the tree only grows (approximately) one more level deep after doubling.

By analogy, the red-black tree `find` function *should* be equivalent to a linear `search` through a sequential list, just faster. We can define the linear `search` function as

```
search :: Eq a => a -> [a] -> Maybe a
search x []    = Nothing
search x (y:ys)
  | x == y     = Just y
  | otherwise = search x ys
```

It is relatively easier to see that the linear `search` function is correct because it exhaustively checks every element: if the given list contains an element equal to the given value, then `search` will return that element, and otherwise `search` will return nothing. In contrast, it is harder to see that the binary `find` function is harder to see that it is correct, because it skips over many elements without even checking them. But, you can prove that the efficient `find` function is correct by proving that it is equal to the simpler `search` specification function.

Exercise 4.1 (10 points)**.** Using equational reasoning, show that, for all values `x :: a` and `ys :: [a]`, if `x /= y` for each `y` in the list `ys`, then

```
  search x ys = Nothing
```

Exercise 4.2 (10 points)**.** Suppose that `x :: a`, `ys :: [a]`, and `zs :: [a]` are arbitrary values. Use equational reasoning to:

1. Show that, if `x /= y` for each `y` in the list `ys`, then

   ```
   search x (ys ++ zs) = search x zs
   ```

2. Show that, if `x /= z` for each `z` in the list `zs`, then

   ```
   search x (ys ++ zs) = search x ys
   ```

Hint 4.1. The built-in append function `(++)` following the recursive definition

```
(++) :: [a] -> [a] -> [a]
[]     ++ zs = zs
(y:ys) ++ zs = y : (ys ++ zs)
```

Since `(++)` recursively takes apart its left (first) argument, it may be helpful to start the proof of exercise 4.2 by doing an induction on the structure of the list `ys` (the left argument to `(++)` in both equations). For the purposes of this exercise, you may assume that `(++)` is associative, meaning that for all list values `xs, ys, zs :: [a]`, you may assume that

```
  (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

This means that it doesn't matter how you parenthesize a chain of `(++)` operations, as all groupings all equal. By default, an unparethesized chain of `(++)` is grouped to the right, so that

```
ws ++ xs ++ ys ++ zs = ws ++ (xs ++ (ys ++ zs))
```

Exercise 4.3 (10 points)**.** Show that for all values `x :: a`, `y :: a`, `ys :: [a]`, and `zs :: [a]`, if `x == y` and `x /= z` for each `z` in `zs`, then

```
search x (zs ++ ([y] ++ ys)) = Just y
```

Exercise 4.4 (15 points)**.** Show that, for all red-black trees `t :: RBTree a` and element values `x :: a`, if `t` is a well-formed red-black tree (meaning it satisfies properties 0–3 described in the introduction to red-black trees), then

```
search x (toList t) = find x t
```

Hint 4.2*.* The ordered property of red-black trees (property 0) will be important for proving exercise 4.4. You may also find some of properties proved above in exercises 4.1 to 4.3 useful when doing equational reasoning in exercise 4.4.

5 *Bonus:* Red-Black Trees as Maps (30 extra credit)

We can use the `Indexed i a` type, and its associated `Eq` and `Ord` instances, from Assignment 1 to model maps from keys (indexes) to values (items) using red-black trees:

```
type RBMap i a = RBTree (Indexed i (Maybe a))
```

Bonus Exercise 5.1 (10 extra credit)**.** Implement the function

```
deleteAt :: Ord i => i -> RBMap i a -> RBMap i a
```

which removes an element stored at an index `i` from the given tree by setting the element at that index to `Noting`. If it turns out there is no element in the tree stored at the given index, then `deleteAt` should do return the same tree it was given. Like `find` and `insert` previously defined in section 1, `deleteAt` should not search the entire tree, but only check the single relevant path of the tree based on the order of the indexes.

Bonus Exercise 5.2 (10 extra credit)**.** Implement the following wrapper functions

```
findAt   :: Ord i => i -> RBMap i a -> Maybe a
insertAt :: Ord i => i -> a -> RBMap i a -> RBMap i a
```

using `find` and `insert`. Find should return the element stored at the given index (or `Nothing` if there is no element stored at the index) and `insertAt` should insert a new index-value mapping into the given `RBMap i a`, overriding the existing mapping if one was already present.

Bonus Exercise 5.3 (10 extra credit)**.** Implement the functions

```
toAssoc   :: RBMap i a -> [Indexed i a]
fromAssoc :: Ord i => [Indexed i a] -> RBMap i a
```

that convert between an `RBMap i a` and an association list `[Indexed i a]`. These two functions are similar to `toList` and `fromList`, except that `toAssoc` should ignore any index mapped to `Nothing` in `RBMap i a`.

6 *Bonus:* Testing Map Properties (15 extra credit)

Bonus Exercise 6.1 (5 extra credit)**.** Modify the properties defined in section 3 to operate over `RBMap Int Int` instead of `RBTree Int` by changing their type signature.

Bonus Exercise 6.2 (5 extra credit)**.** Similar to exercise 3.3, write properties testing the red-black properties are maintained after `delete`ing an element from a `RBMap`, and add them to your `main` test suit.

Bonus Exercise 6.3 (5 extra credit)**.** Implement the additional properties for the map operations and add them to your `main` test suit:

```
findAfterInsertAt :: Int -> Int -> RBMap Int Int -> Bool
findAfterDeleteAt :: Int -> RBMap Int Int -> Bool
deleteAfterInsert :: Int -> Int -> RBMap Int Int -> Bool
```

`findAfterInsertAt` should check that `findAt i (insertAt i x t)` is `Just x` for any index `i`, element `x`, and tree `t`.

`findAfterDeleteAt` should check that `findAt i (deleteAt i t)` is `Nothing` for any index `i` and tree `t`.

`deleteAfterInsert` should test that

```
toAssoc (deleteAt i (insertAt i x t))
```

is the same as

```
toAssoc (deleteAt i t)
```

for any index `i` and tree `t`. The reason why the call to `toAssoc` is necessary is to ignore the internal differences in two `RBMap`s that do not matter (like the remnants of a deleted index).