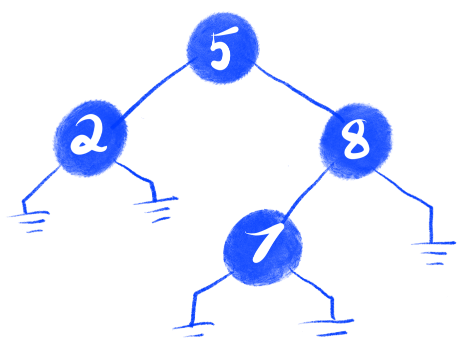
Effective Functional Programming
Correctness
Assignment 3
`Red-Black Trees`

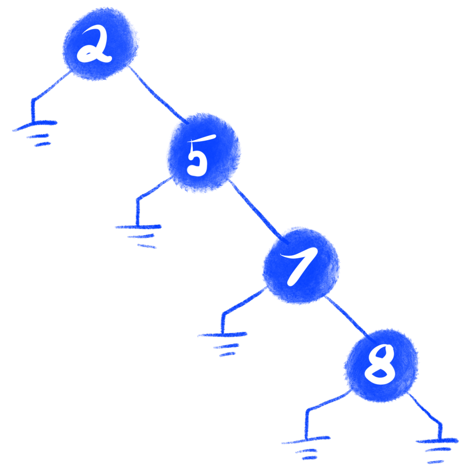
Paul Downen

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, consider the following ordered binary tree where numbers larger than the one in the current node are always stored to the right, and numbers smaller than the current node are always stored to the left:



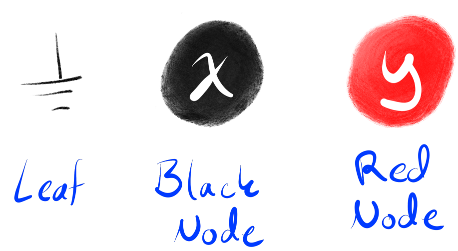
We can quickly confirm that 9 is not in this ordered tree by only checking the right-most path: starting from the top node containing 5, move to 8 on the right (because $9 > 5$), then move again to the right (because $9 > 8$) and end the search at the right-most terminal leaf. If there was a 9 in the tree, it would be to the right of the 8 because the binary tree is ordered. 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 lead to a node containing 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 to 8, which has four steps).

One way of maintaining balance is to use a *red-black tree*: an ordinary binary tree but where every node is colored either red or black. Written graphically, a red-black tree comes in three different forms: a leaf, a black node, or a red node.



In addition to marking nodes with a color, a proper red-black tree also meets the following properties:

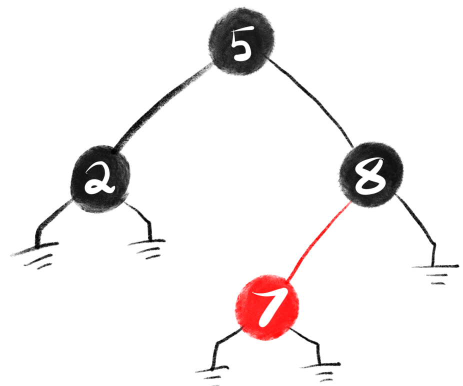
0. Ordered: The values of the nodes are in strict ascending order with respect to a left-to-right depth-first search. In other words, everything in the left sub-tree of a node is strictly less than the node value, and everything in the right sub-tree of a node is strictly greater than the node value.

1. Black Root: The root of every complete tree must be black. A leaf is considered black. Only sub-trees can be red.

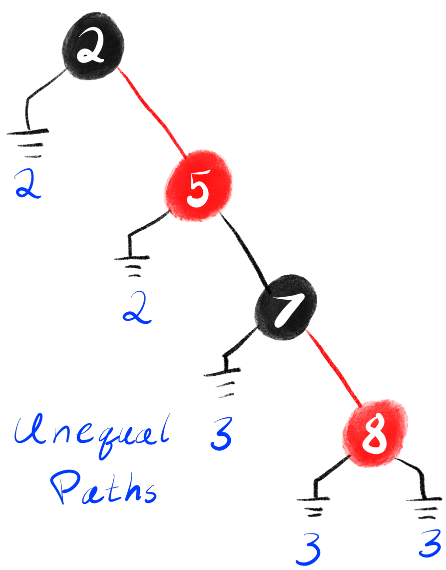
2. No Red Chains: The left and right sub-trees of a red node must be black.

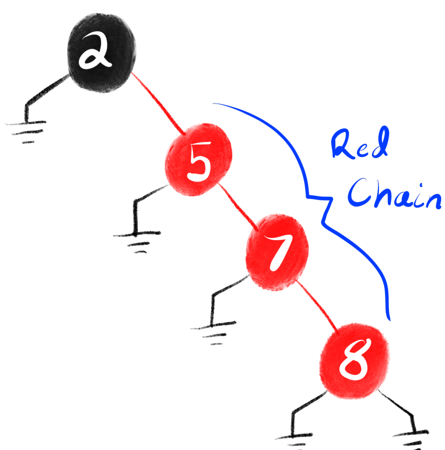
3. Equal Paths: The number of black nodes contained in every path from the root to a leaf (including the root and leaf) must be equal.

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 (Equal Paths) and 2 (red chains) respectively:





Red-black trees can be represented in Haskell with the following data types:

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

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

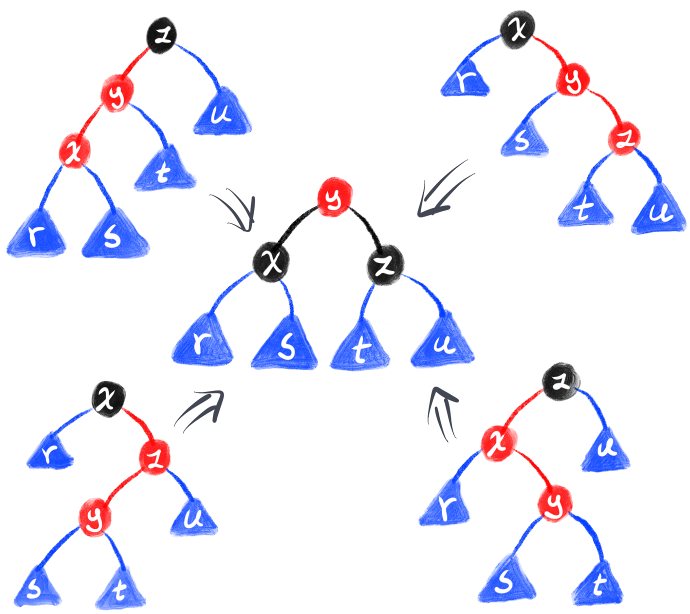
The constructors of `Color` and `RBTree a` effectively correspond to the three forms of red-black trees: leaves are built by `L`, black nodes by `N B`, and red nodes by `N R`. The color of a red-black tree is the color of its root node (which is built by either `L` or `N`).

The main operations of red-black trees are the functions

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

`find` looks to see if a given element is in a red-black tree, returning `Nothing` if it is not there, and `insert` adds a new element to a red-black tree, returning the updated tree with the element in it. Because red-black trees are ordered, both `find` and `insert` only need to trace a single path from the root of the tree to a leaf to do their job. And because the other red-black tree properties (namely (2) No Red Chains and (3) Equal Paths) force trees to be balanced, any path through the tree has only $\log(n)$ steps. As a result, both `find` and `insert` cost $O(\log(n))$ time, where n is the number of elements in the tree they operate on.

The challenge of maintaining balanced trees is that `insert` might make a tree out of balance by adding one too many nodes along a path. Assuming, `insert` always adds a new **R**ed **N**ode, this can be seen in red-black trees as a violation of the No Red Chain property. Because of this, `insert` needs to rebalance the nodes along the path it takes, according to this balancing diagram that transforms bad red-black trees into good ones:



The subtle properties and balancing act make red-black trees a difficult data structure to implement correctly, since it is easy to accidentally break one of properties needed to ensure that `find` and `insert` are efficient. Because it is not easy to see if the code implementing red-black trees is correct, they are a great candidate for a good testing suite. The red-black tree properties form hard invariants that must be maintained by every `insert`, and we need to know if `find`ing an element in a tree never accidentally misses the answer by taking the wrong path.

A complete implementation of red-black trees, along with the `find` and `insert` functions, has been given to you in the template for this assignment. Your job is to ensure that it is correct. You will create an automated test suite for checking that `insert` builds good red-black trees, and to work towards a proof that `find` does the same thing as a complete, thorough search.

1 Testing Red-Black Properties (50 points)

In order to confirm that the implementation of red-black trees you were given is correct, you will need to generate many test cases to check that all the red-black tree properties are followed. This can be done automatically using the `QuickCheck`[1] library, by showing how to generate `arbitrary` red-black trees. A fully-automated test suit for checking correctness of the provided implementation of red-black trees using the `hspec`[2] framework.

The code for generating `arbitrary` trees as well as the `main` testing script is already provided for you in the template of this assignment in `test/Spec.hs`. You are responsible for writing definitions of the properties that will be checked, and can give in your answers to the following Exercises in this section by filling in the blanks in `test/Spec.hs`. To check your test suite, you can run the command

```
> stack test
```

in the project directory, which will run the `main` testing script found in `test/Spec.hs` and print the results.

Exercise 1.1 (5 points)**.** Implement a test with the type signature

```haskell
findAfterInsert :: Int -> RBTree Int -> Bool
```

`findAfterInsert` is a function which takes any `Int x`and valid red-black tree `t` (of type `RBTree Int`) containing `Int`s, and checks that `x` can be found in the tree made by `insert`ing `x` into `t`. In other words, the output of `findAfterInsert x t` should be:

- `True` if `find x (insert x t)` is equal to `Just x`, and
- `False` otherwise. **End Exercise 1.1**

Exercise 1.2 (10 points)**.** The provided `RedBlackTree` module defines functions

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

[1] https://hackage.haskell.org/package/QuickCheck
[2] https://hackage.haskell.org/package/hspec

for converting between trees and lists. Implement a test with the type signature

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

`roundTripSort` is a function which takes any list of `Int`s, and checks that a round trip from `[Int]` to `RBTree a` (by calling `fromList`) and back to `[Int]` contains all the unique elements as the original list in ascending order. In other words, the output of `roundTripSort xs` should be:

- `True` if `toList (fromList xs)` is equal to the list obtained by sorting and removing duplicates from `xs`, and
- `False` otherwise. **End Exercise 1.2**

Hint 1.1. The `Data.List` module provides the two functions

```
sort :: Ord a => [a] -> [a]
nub  :: Eq a  => [a] -> [a]
```

`sort` sorts a list and `nub` removes duplicate elements from a list. *End Hint 1.1*

Exercise 1.3 (15 points)**.** Implement tests with these type signatures

```
ordered     :: RBTree Int -> Bool
blackRoot   :: RBTree Int -> Bool
noRedChains :: RBTree Int -> Bool
```

that encode properties 0–2 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.

0. `ordered t` returns `True` only when the list of elements in `t`, as given by `toList t`, are in order.

   Hint 1.2. Remember that `sort` from `Data.List` sorts a list. You can check if a list `xs` is in order by checking that `xs` is equal to `sort xs`. *End Hint 1.2*

1. `blackRoot t` returns `True` only when the root node of is black.

   Hint 1.3. Remember that a leaf `L` counts as black, and a node built by `N` has the color contained in the first parameter of `N`. Since you only need to check the color of the root, the `blackRoot` function does not need to recurse or check any sub-trees. *End Hint 1.3*

2. `noRedChains t` returns `True` only when there is never any two red nodes in a row. In other words, if `t` contains a node of the shape `N R left x right` *anywhere*, then it must be the case that both its `left` and `right` sub-trees have a `blackRoot`.

   Hint 1.4. `noRedChains t` needs to check the property for every single node within `t`. So `noRedChains t` needs to recursively check the sub-trees of `t`, unlike `blackRoot t` that only checks a property of the top-most node of `t`. *End Hint 1.4*

End Exercise 1.3

Exercise 1.4 (20 points)**.** The last property of red-black trees is the most complex, and involves comparing a particular count over all possible paths you can take from the root to the leaf of a tree. Generating this list of paths has already been defined for you in the `RedBlackTree` module in the template, which includes the function

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

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

The result of `paths t` contains a list of paths. Each path is itself a list containing the `Color` and value contained within each non-leaf node visited along that path. The only path possible starting from an empty leaf `L` is the empty path `[]`. The two possible paths starting from the single-node tree `(N B L 1 L)` are `[[(B,1)],[(B,1)]]`: both start at the root `N`ode with color `B` and value `1`, and can continue down to the left or right `L`eaf. The two-node tree `(N B L 1 (N R L 2 L))` has three paths `[[(B,1)],[(B,1),(R,2)],[(B,1),(R,2)]]`, and so on.

To finish implementing a test for the Equal Paths property, break it down into smaller parts. First, implement helper functions with the type signatures

```haskell
countBlackNodes :: Path a      -> Int
pathCounts      :: RBTree Int -> [Int]
```

`countBlackNodes` takes a single path, and should count only the black nodes visited along that path. So `countBlackNodes [] = 1`, because an empty path always ends at a leaf, which counts as a black node. Another example is `countBlackNodes [(B,1)] = 2`, which counts 1 for the first `B`lack non-leaf `N`ode, and another 1 for the final `L`eaf. In addition, `countBlackNodes [(B,1),(R,2)] = 2` as well, because the `R`ed `N`ode visited in the second step isn't counted. In general, `countBlackNodes ((c, x) : path)` should add 1 to the count of `path` when `c` is `B`lack, and otherwise just be the same count as `path` when `c` is `R`ed.

`pathCounts` takes a red-black tree `t`, calculates all the `paths` starting from `t`, and applies `countBlackNodes` to each one of those paths, collecting the list of counts taken for each individual path. As examples,

```haskell
  pathCounts L                     = [1]
  pathCounts (N B L 1 L)           = [2,2]
  pathCounts (N B L 1 (N R L 2 L)) = [2,2,2]
```

Hint 1.5*.* Recall from the lectures that you can use the `map` function or a list comprehension to apply a function to every element of a list. *End Hint 1.5*

Using the above helper functions, implement a test with the type signature

```haskell
equalPaths  :: RBTree Int -> Bool
```

that encodes property 3 (Equal Paths) of red-black trees as a Haskell function returning `True` for a tree only when it satisfies property 3. More specifically, `equalPaths t` should:

1. Calculate the list of all `pathCounts t` for the given tree.

2. Check if every number in the list from step 1 is equal, returning `True` if they are all equal to the same number `n`, and `False` otherwise.

   Hint 1.6. Since even the empty tree `L` has one path in it, you know that `pathCounts t` will never be an empty list. So you can check that every element of `pathCounts t` is equal to the `head` of `pathCounts t`. The `all :: (a -> Bool) -> [a] -> Bool` function from the standard library checks if every element in a list satisfies some `Bool`ean test. *End Hint 1.6*

End Exercise 1.4

2 Proving Correctness (25 points, 45 extra credit)

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 correct, because it skips over many elements without even checking them. You can prove that the efficient `find` function is correct by proving that it is equal to the simpler `search` specification function. Proving, which considers every possible argument to a function, even if there are infinitely many options, is much more thorough than testing, which only checks a relatively small, finite number of possible arguments.

The following exercises in this section ask you to employ *equational reasoning* to prove that two expressions are equal to one another. Show your work by doing a “pen-and-paper” style calculation by hand, chaining equations together similar to solving an algebra problem, and include your work in a (plain text, pdf, word, or hand-written) document. The template for this assignment contains an outline for this section in `test/Proofs.md`, where you can fill in your answers to each of the following sections. Only Exercises 2.1 and 2.3 are mandatory for this assignment. You can optionally do the Bonus Exercises 2.2 and 2.4 for extra credit.

Exercise 2.1 (15 points)**.** Using equational reasoning, prove that, if `x == y` is `False` for each `y` in the list `ys`, then

```
search x ys = Nothing
```

End Exercise 2.1

Hint 2.1. Try to prove this property by induction on the structure of the list `ys`. To do so, answer these two questions:

1. What happens when `ys` is the empty list `[]`? In other words, manually calculate for yourself the result of the function call `search x []`, according to the definition of `search`, and show it is equal to `Nothing` regardless of the value of `x`.

2. Assuming that `search x ys' = Nothing`, what happens when `ys` is the non-empty list built by `y:ys'`? In other words, use equational reasoning to calculate the result of the function call `search x (y:ys')` according to the definition of `search` and the assumption that `x == y` is `False`, and show it is equal to `Nothing`. In one of the steps, you will need to use the assumption that `search y ys' = Nothing` (known as the *inductive hypothesis*) to finish the equation. *End Hint 2.1*

Bonus Exercise 2.2 (20 extra credit)**.** Use equational reasoning to prove the following two properties:

1. If `x == z` is `False` for each `z` in the list `zs`, then

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

2. If `x == y` is `False` for each `y` in the list `ys`, then

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

End Bonus 2.2

Hint 2.2. The built-in append function `(++)` has the recursive definition

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

Since `(++)` recursively takes apart its left (first) argument, it may be helpful to start the proof of Bonus Exercise 2.2 by doing an induction on the structure of the list `zs` (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 unparenthesized chain of (++) is grouped to the right, so that

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

End Hint 2.2

Exercise 2.3 (10 points)**.** Use equational reasoning to prove that, if `x == y` is `True` and `x == z` is `False` for each `z` in `zs`, then

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

End Exercise 2.3

Hint 2.3. You do not need to use induction to prove this property. Instead, you can apply one of the properties from Bonus Exercise 2.2 (whether or not you completed that optional exercise) to calculate the result directly. Which of the assumptions in the two properties of Bonus Exercise 2.2 matches the assumptions that you have here? *End Hint 2.3*

Bonus Exercise 2.4 (25 extra credit)**.** Use equational reasoning to prove that, 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
```

End Bonus 2.4

Hint 2.4. The ordered property of red-black trees (property 0) will be important for proving Bonus Exercise 2.4. You may also find some of properties proved above in Exercises 2.1 and 2.3 and Bonus Exercise 2.2 useful when doing equational reasoning in Bonus Exercise 2.4. *End Hint 2.4*

3 *Bonus:* Trees as Maps (40 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 3.1 (15 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 return the same tree it was given. Like `find` and `insert` found in the `RedBlackTree` module provided in the template for this assignment, , `deleteAt` should not search the entire tree, but only check the single relevant path of the tree based on the order of the indexes. **End Bonus 3.1**

Bonus Exercise 3.2 (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. **End Bonus 3.2**

Bonus Exercise 3.3 (15 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`. **End Bonus 3.3**

4 *Bonus:* Testing Map Properties (40 extra credit)

Bonus Exercise 4.1 (5 extra credit)**.** Modify the properties defined in section 1 to operate over `RBMap Int Int` instead of `RBTree Int` by changing their type signature. **End Bonus 4.1**

Bonus Exercise 4.2 (10 points)**.** Test that the properties defined in Exercises 1.3 and 1.4 still hold after an `Int` is inserted into the tree with `insertAt`, and add them to your `main` test suit. For example, test `\x -> inOrder . insertAt x`, and so on for all red-black properties 0–3. **End Bonus 4.2**

Bonus Exercise 4.3 (10 extra credit)**.** Similar to Bonus Exercise 4.2, 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. **End Bonus 4.3**

Bonus Exercise 4.4 (15 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). **End Bonus 4.4**