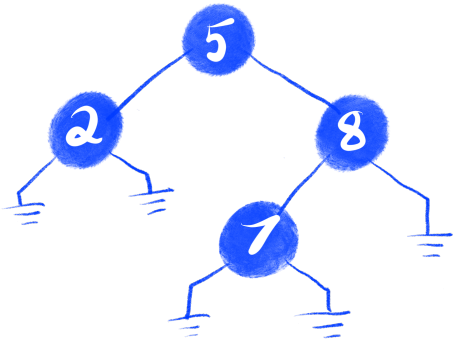
Effective Functional Programming
Correctness
Assignment 3
`Red-Black Trees`

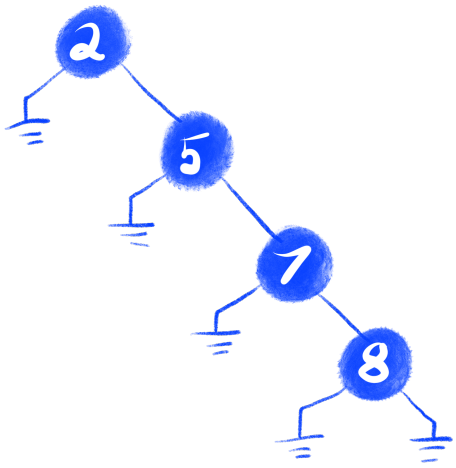
Paul Downen

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:



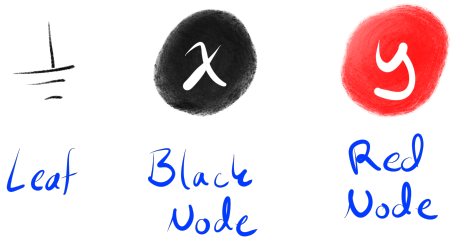
We can quickly confirm that 9 is not in this ordered tree by only checking the right-most path: starting from the top node containing 5, move to 8 on the right (because $9 > 5$), then move again to the right (because $9 > 8$) and end the search at the right-most terminal leaf. If there was a 9 in the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

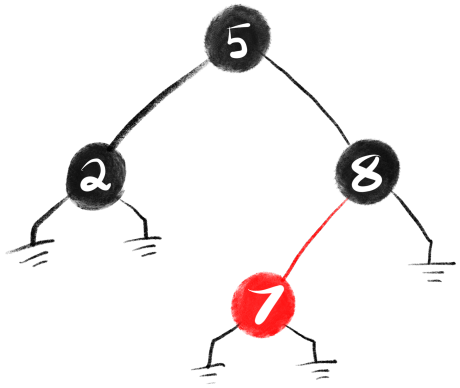
0. Ordered: The values of the nodes are in strict ascending order with respect to a left-to-right depth-first search. In other words, everything in the left sub-tree of a node is strictly less than the node value, and everything in the right sub-tree of a node is strictly greater than the node value.

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

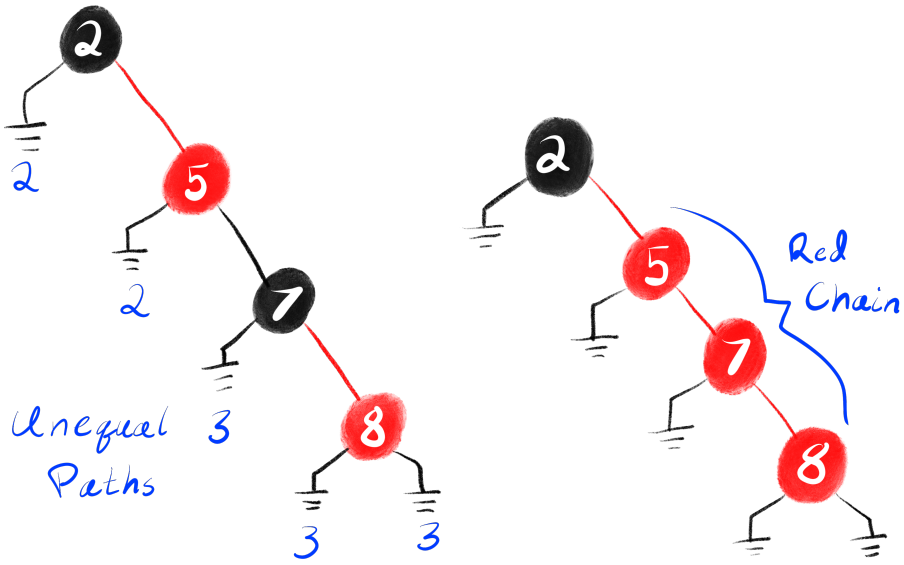
2. No Red Chains: The left and right sub-trees of a red node must be black.

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:

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

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

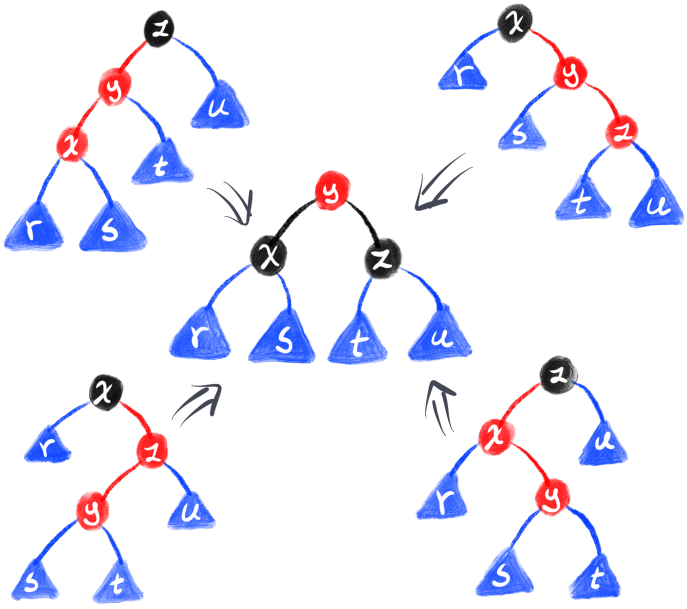
The constructors of `Color` and `RBTree a` effectively correspond to the three forms of red-black trees: leaves are built by `L`, black nodes by `N B`, and red nodes by `N R`. The color of a red-black tree is the color of its root node (which is built by either `L` or `N`).

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, and to separate it from other, subtly buggy, alternatives. 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 (45 points+ 30 extra credit)

In order to confirm that the implementation of red-black trees you were given is correct, you will need to generate several test cases to check that all the red-black tree properties are followed, and that the operations work as expected. 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 is organized 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 the `test` directory. 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 the library file `src/Properties.hs`. Your definitions `Properties` are imported and used in the red-black tree specification `test :: Spec` in `test/Spec.hs`, which is called by the `main` testing operation. To check your test suite, you can run the command

```
> stack test
```

or alternatively

```
> cabal test
```

in the project directory, which will run the `main` operation in `test/Main.hs` and print the results.

To keep things interesting, your tests will have to correctly *pass* or *fail* different alternative implementations of the main tree operations. Each of these possible implementations are all instances of the `OrdTree` type class in the `Data.RedBlack` module:

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

```haskell
class OrdTree t where
  -- An empty tree
  emptyT :: t a
  -- Find a given element somewhere in a tree
  find   :: Ord a => a -> t a -> Maybe a
  -- Insert a new element into a tree
  insert :: Ord a => a -> t a -> t a

  -- Convert a tree to a list
  toList   ::          t a -> [a]
  -- Convert a list into a tree (can be defined via insert)
  fromList :: Ord a => [a] -> t a
```

Your properties won't know which implementation they are being used to test; they will have to only rely on generically enforcing the invariants they are responsible for.

Exercise 1.1 (5 points)**.** Implement two tests with the type signatures

```haskell
findAfterInsert  :: (Ord a, OrdTree t) => a -> t a -> Bool
irrelevantInsert :: (Ord a, OrdTree t) => a -> a -> t a -> Property
```

Both of these functions must consider *any* possible type of orderable element and *any* possible implementation of the `OrdTree` interface. To do this, you will only be able to rely on the methods provided by `Ord` (equality and comparison) and `OrdTree` (such as `find` and `insert`).

`findAfterInsert` is a function which takes any `Ord`erable `a` `x` and any ordered tree `ys` containing `a`s, and checks that `x` can be found in the tree made by `insert`ing `x` into `ys`. In other words, the output of `findAfterInsert x ys` should be:

- `True` if `find x (insert x ys)` is equal to `Just x`, and
- `False` otherwise.

`irrelevantInsert` takes two *different* elements `x` and `y` and one such tree `zs`, and checks that `find x` returns the same result whether or not `y` is `insert`ed into `zs`. In other words, the property `irrelevantInsert x y zs` has the *precondition* `x /= y`, and if that pre-condition is true, should be:

- `True` if `find x (insert y zs)` returns the same result as `find x zs`, and
- `False` otherwise.

End Exercise 1.1

Hint 1.1. The operation `(==>) :: Bool -> Property` from `Test.QuickCheck` is the proper way to add a precondition to a property. Unlike a simple boolean implication (like an `if then else`), `(==>)` has special support for making sure the expected number of test cases actually run. If an example input fails the

precondition (i.e., the boolean to the left of `==>` is `False`), then that example is thrown out without being counted, and another is tried, until the property (i.e., to the right of `==>`) is actually run and proved successful for the total number of test cases. Note, that if `QuickCheck` has too hard of a time trying to come up with examples that satisfy the precondition (e.g., it goes through 1000 examples when trying to just find 100 that work), it will give up, and the test will fail due to insufficient examples. For example, requiring that `x` and `y` are not equal is not particularly restrictive, as it is unlikely to pick the same value for both. However, requiring that `x` and `y` are equal is unlikely to happen coincidentally by random chance, and will tend to exhaust `QuickCheck`'s patience when trying to come up with enough valid examples. *End Hint 1.1*

Bonus Exercise 1.2 (10 extra credit)**.** Balanced trees, like red-black trees, are useful because even in the worst case, their height grows (logarithmically, $O(\log(n))$) slower than the number (n) of elements they contain. Specifically in the case of proper red-black trees containing n elements, the upper bound on their maximum height is $2\log_2(n+1)$.

Calculate the maximum height of a red-black tree, and make sure it is within the expected bounds, by implementing the functions:

```
height     :: RBTree a -> Int
upperBound :: Int -> Int
```

The `height` of an empty tree (`L`) is 0, and the `height` of a node is 1 plus the maximum hight of its two sub-trees. `upperBound n` should return the result of the formula $2\log_2(n+1)$ (rounded up). **End Bonus 1.2**

Hint 1.2. The Haskell function for calculating the logarithm of a number with a given base is `logBase :: Floating a => a -> a -> a`. However, note that you will have to do some numeric coercions to turn an integer into a floating number, and then back again. *End Hint 1.2*

Bonus Exercise 1.3 (20 extra credit)**.** The `OrdTree` type class includes the functions

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

for converting between trees and lists. Since `RBTree`s implement a form of sets (the order between elements cannot be controlled, and duplicates are overwritten), a round-trip from a list to a tree and back effectively converts that list into an (ordered) set.

Implement the function

```
roundTrip :: (Ord a, OrdTree t) => t a -> [a] -> [a]
```

for sending a list through a red-black tree round-trip: `roundTrip _ xs` should generate an intermediate tree $ys :: ta$ using `fromList`, which is then converted back to a list using `toList`.

Note that since `fromList` and `toList` are generic functions that work for *any* instance of `OrdTree t`, Haskell will get confused and not know which implementation to use if you just compose these two functions directly as `toList (fromList xs)`. That's where the first parameter comes in: it's value does not change the output, but its *type* `t a` spells out which instance of `OrdTree t` to use. In order to avoid the "ambiguous" type errors when composing `fromList` and `toList`, you might find this helper function useful:

```
ofType :: a -> a -> a
x `ofType` _ = x
```

The result is exactly the value of the first argument (ignoring the value of the second argument), however the *types* of the two arguments have to be the same. If you want the value of `x` but Haskell can't figure out that it's type should be `a`, you can use another `y :: a` on hand to call `x `ofType` y :: a` which is just the value of `x` (now that we know it's an `a`).

To make sure that a round-trip through red-black trees correct, fill out the following properties on lists that the result of a correct implementation should satisfy:

```
sortedList :: (Ord a) => [a] -> Bool
uniqueList :: (Ord a) => [a] -> Bool
subList    :: (Ord a) => [a] -> [a] -> Bool
```

- `sortedList xs` returns `True` when `xs` is in order (that is, `xs` is already sorted).
- `uniqueList xs` returns `True` when `xs` contains no duplicates (that is, any element of `xs` never appears more than once).
- `subList xs ys` returns true if *every* element of `xs` appears somewhere (at least once) in `ys`, similar to a subset. **End Bonus 1.3**

Hint 1.3. 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 1.3*

Exercise 1.4 (15 points)**.** Implement tests with these type signatures

```
ordered     :: Ord a => RBTree a -> Bool
blackRoot   ::          RBTree a -> Bool
noRedChains ::          RBTree a -> 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 xs` returns `True` only when the list of elements in `xs`, as given by `toList xs`, are in order.

Hint 1.4. Remember that `sort` from `Data.List` sorts a list. You can check if a list `ys` is in order by checking that `ys` is equal to `sort ys`. *End Hint 1.4*

(1) `blackRoot xs` returns `True` only when the root node of is black.

Hint 1.5. 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 1.5*

(2) `noRedChains xs` returns `True` only when there is never any two red nodes in a row. In other words, if `xs` 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 1.6. `noRedChains xs` needs to check the property for every single node within `xs`. So `noRedChains xs` needs to recursively check the sub-trees of `xs`, unlike `blackRoot xs` that only checks a property of the top-most node of `xs`. *End Hint 1.6*

End Exercise 1.4

Exercise 1.5 (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

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

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

The result of `paths xs` 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

```
countBlackNodes :: Path a   -> Int
pathCounts      :: RBTree a -> [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 Black non-leaf Node, and another 1 for the final Leaf. In addition, `countBlackNodes [(B,1),(R,2)] = 2` as well, because the Red Node 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 Black, and otherwise just be the same count as `path` when `c` is Red.

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

```
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 1.7. 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 1.7*

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

```
equalPaths  :: RBTree a -> 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 xs` should:

1. Calculate the list of all `pathCounts xs` 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 1.8. 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 xs` is equal to the `head` of `pathCounts xs`. 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 1.8*

End Exercise 1.5

2 Trees as Maps (55 points)

As-is, red-black trees provide an efficient (logarithmic) data structure for modeling sets. But what if we need to model a dictionary mapping some type of keys (indexes) to values (items)? There's a quick trick to reuse our code!

The `Indexed i a` type, way back from Assignment 1, can turn a red-black tree into a red-black *map*. Its associated `Eq` and `Ord` instances produce exactly the right behavior of only paying attention to the key for the purpose of inserting and finding, while the value is along for the ride:

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

- Because `Indexed i (Maybe a)` only looks at the `i` value for equality and ordering, so it acts as the "key" by the underlying `RBTree` implementation of `insert` and `find` for deciding what order to put this pair into the tree, and how to access it in logarithmic time.
- If we insert a new item at an index that is already in the tree, the two index-item pairs will look the "same" according to equality `(==)`, so insert will replace the old item with the new one.
- The reason that the item is a `Maybe a` instead of just an `a` is to more easily support a (simple, but sadly memory leaking) deletion operation.

An outline sketching how to repurpose red-black trees as maps is given in `src/Map.hs`. Your job is to fill in the undefined code.

Exercise 2.1 (10 points)**.** First, copy your definition of `Indexed` values from Assignment 1, including:

```
data Indexed i a = ...

item  :: Indexed i a -> a
index :: Indexed i a -> i

instance Eq i => Eq (Indexed i a) where
  ...

instance Ord i => Ord (Indexed i a) where
  ...
```

Second, give your `Indexed` data type a new instance of `Arbitrary`:

```
instance (Arbitrary i, Arbitrary a) => Arbitrary (Indexed i a) where
  ...
```

by implementing the overloaded function `arbitrary :: Gen a`. The instance of `arbitrary :: Indexed i a` needs to `return` multiple different, well-defined values, depending on the randomness of `Gen`. **End Exercise 2.1**

Hint 2.1. `Gen` is itself an instance of `Monad`, which means you can use `do`-notation to stitch together multiple `Gen`-operations, like `arbitrary`. To properly generate an `arbitrary` value of `Indexed i a`, you will (unsurprisingly) need an `arbitrary` value of type `a` and an `arbitrary` value of type `i`. Once you've gotten hold of both values (for example, using a `do` to bind the results), constructing and `return`ing your `Indexed i a` value should be straightforward. Feel free to peak at `test/Spec.hs` for more hints. *End Hint 2.1*

NOTICE: Your definition of `Indexed`, it's operations, and its `Arbitrary` instance are needed to test red-black maps, and so Exercise 2.1 is a prerequisite for the exercises below. Don't invest too much time in them before completely finishing Exercise 2.1.

Exercise 2.2 (10 points)**.** 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.

To be correct, your definitions of `findAt` and `insertAt` need to satisfy these properties:

- We can always `find` an item after it is `insert`ed at the same index:

  ```
  findAt i (insertAt i x t) = Just x
  ```

- `find` is not affected by irrelevant insertions: given that `i /= j`,

  ```
  findAt i (insertAt j x t) = findAt i t
  ```

- The most recent `insert`ion overrides older ones:

  ```
  insertAt i x (insertAt i y t) =~= insertAt i x t
  ```

- Multiple insertions at different indexes can be reordered:

  ```
  insertAt i x (insertAt j y t) =~= insertAt j y (insertAt i x t)
  ```

where `=~=` means comparing two trees only in terms of their index-item mapping set, and ignoring their underlying tree structures. **End Exercise 2.2**

Exercise 2.3 (20 points)**.** 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 `Nothing`. 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 `Data.RedBlack` module, `deleteAt` should not search the entire tree, but only check the single relevant path of the tree based on the order of the indexes.

To be correct, your definition of `deleteAt` needs to satisfy these properties:

- `deleteAt` removes one, and only one, element. Given a non-empty `RBMap i a`, deleting an index that is associated with an item produces a map with exactly one less element (when ignoring all `Nothing` entries).

- Deleted indexes can no longer be found:

  ```
  findAt i (deleteAt i t) = Nothing
  ```

- `find` is not affected by deleting other indexes: given that `i /= j`,

```
findAt i (deleteAt j t) = findAt i t
```

- Deletion overrides insertion at the same index:

  ```
  deleteAt i (insertAt i x t) =~= deleteAt i t
  ```

- Insertion overrides deletion at the same index:

  ```
  insertAt i (deleteAt i x t) =~= insertAt i t
  ```

- Deletion and insertion at different indexes can be reordered: given that `i /= j`,

  ```
  deleteAt i (insertAt j x t) =~= insertAt j x (deleteAt i t)
  ```

- Redundant deletions at the same index are idempotent:

  ```
  deleteAt i (deleteAt i t) =~= deleteAt i t
  ```

- Multiple deletions can always be reordered: regardless if `i == j` or `i /= j`,

  ```
  deleteAt i (deleteAt j t) =~= deleteAt j (deleteAt i t)
  ```

where `=~=` again means comparing two trees only in terms of their index-item mapping set. **End Exercise 2.3**

Exercise 2.4 (15 points)**.** 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`.

To be correct, your definition of `toAssoc` and `fromAssoc` need to satisfy these properties:

- A `toAssoc . fromAssoc` round-trip always produces a sorted list.
- A `toAssoc . fromAssoc` round-trip always produces a list with no duplicate elements.
- A `toAssoc . fromAssoc` round-trip always produces a list that contains every element (up to ==) that was in the input.
- `fromAssoc` produces trees that with all red-black invariants (0–3).

End Exercise 2.4

3 *Bonus:* Proving Correctness (100 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 Bonus Exercises 3.1 and 3.3 are mandatory for this assignment. You can optionally do the Bonus Exercises 3.2 and 3.4 for extra credit.

Bonus Exercise 3.1 (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 Bonus 3.1

Hint 3.1. 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 3.1*

Bonus Exercise 3.2 (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 3.2

Hint 3.2. 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 3.2 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 3.2

Bonus Exercise 3.3 (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 Bonus 3.3

Hint 3.3. You do not need to use induction to prove this property. Instead, you can apply one of the properties from Bonus Exercise 3.2 (whether or not you completed that optional exercise) to calculate the result directly. Which of the assumptions in the two properties of Bonus Exercise 3.2 matches the assumptions that you have here? *End Hint 3.3*

Bonus Exercise 3.4 (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 3.4

Hint 3.4. The ordered property of red-black trees (property 0) will be important for proving Bonus Exercise 3.4. You may also find some of properties proved above in Bonus Exercises 3.1 to 3.3 useful when doing equational reasoning in Bonus Exercise 3.4. *End Hint 3.4*

Bonus Exercise 3.5 (30 extra credit)**.**

1. Use equational reasoning to prove that, `insert` preserves the red-black properties. In other words, prove that for any `x :: a` and `t :: RBTree a`, if `t` satisfies properties 0–3 of red-black trees, then `insert x t` does, too.

2. Prove that, for any list `xs :: [a]`, the tree `fromList xs :: RBTree a` satisfies all red-black properties 0–3.

End Bonus 3.5

Hint 3.5. Because `insert` is defined in terms of a (recursive!) helper function `ins`, you may find it similarly helpful to prove a related lemma directly about `ins` itself. What red-black properties does it preserve? Which might it break? Then use that lemma in your larger proof about `insert` itself. *End Hint 3.5*