COMP50001: Algorithm Design & Analysis

Sheet 5 (Week 6)

Exercise 5.1

When implementing an AVL tree, at each node it is possible to store just the difference in height between the two children. According to the invariants on an AVL tree, the bias can either be -1, 0, or 1.

Implement $insert :: Ord\ a \Rightarrow a \to DTree\ a \to DTree\ a$ and $delete :: Ord\ a \Rightarrow a \to DTree\ a \to DTree\ a$ on *DTree*, an AVL tree which stores differences rather than heights.

```
data HTree a = HTip
             | HNode Int (HTree a) a (HTree a)
data DTree a = DTip
             | DNode Diff (DTree a) a (DTree a)
data Diff = MinusOne | Zero | PlusOne
```

Exercise 5.2

Write a function $fromOrdList :: [a] \to HTree\ a$ which builds an AVL tree from a sorted list in worst case linear time.

```
fromOrdList :: [a] → HTree a
fromOrdList xs = fst (go (length xs) xs)
  where
    go :: Int → [a] → (HTree a, [a])
    go = ...
```

Exercise 5.3

The implementation of *member* presented in lectures uses $2d$ comparisons in the worst case, where d is the depth of the tree. Reimplement *member* to perform at most $d + 1$ comparisons, using only $(\leqslant)$ to compare elements.

```
member :: Ord a ⇒ a → HTree a → Bool
member x HTip = False
member x (HNode _ lte p gt)
  | x < p      = member x lte
  | x ≡ p      = True
  | otherwise = member x gt
```

A possible implementation strategy is to carry the most recent element that *could* be equal to the one being searched for with you; when you hit the bottom of the tree the you can test for equality.

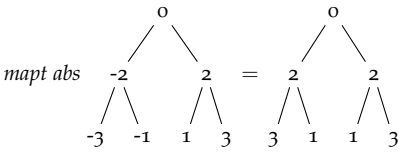
Exercise 5.4

We can define a kind of *map* on search trees as follows:

```
mapt :: (a → b) → HTree a → HTree b
mapt f HTip                = HTip
mapt f (HNode b ls x rs) = HNode b (mapt f ls) (f x) (mapt f rs)
```

Describe the minimal condition to which f must adhere in order for the output of *mapt f* to be a valid search tree, given valid input.

Define a function $mapt' :: Ord\ b \Rightarrow (a \to b) \to Tree\ a \to Tree\ b$ which does not require f to adhere to your condition in order to produce a valid tree.



Exercise 5.5

There is a monoid instance on pairs of lists. Define a monoid instance for *Ordering* such that *uncurry compare* on pairs of lists *of the same length* is a monoid homomorphism.

```
instance Monoid ([a], [a]) where
  ϵ                = ([], [])
  (xl, xr) ◇ (yl, yr) = (xl ++ yl, xr ++ yr)
```

Here is the type of *compare*:

```
compare :: Ord a ⇒ a → a → Ordering
```

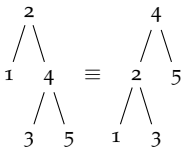
The result of *compare x y* returns *LT*, *EQ* or *GT* depending on whether $x < y$, $x \equiv y$, or $x > y$.

```
data Ordering = LT | EQ | GT
```

Exercise 5.6

Consider a data type representing a set, implemented using AVL trees. The trees will contain no duplicates. Write an *Eq* instance for this type. $(\equiv)$ should be linear, and should not differentiate between trees which are balanced differently.

Given sets xs and ys, f is defined to be *well behaved* if it obeys:



$xs \equiv ys \Rightarrow f\ xs \equiv f\ ys$

Discuss whether the functions *member* x or *insert* x are well behaved. Give other examples of well behaved and non-well behaved functions.

Exercise 5.7

The Boom hierarchy is a classification of data structures based on laws. Given some data structure $S\ a$, with a function $insert :: a \to S\ a \to S\ a$, and $(\cup) :: S\ a \to S\ a \to S\ a$, we can classify it as one of 16 different structures depending on which of the following laws hold:

$$(x \cup y) \cup z = x \cup (y \cup z) \qquad \text{(associativity, 1)}$$
$$\emptyset \cup x = x = x \cup \emptyset \qquad \text{(identity, 2)}$$
$$x \cup y = y \cup x \qquad \text{(commutativity, 3)}$$
$$x \cup x = x \qquad \text{(idempotence, 4)}$$

Name the data structures which obey the following laws:

1. 1 only.
2. 1 and 2 only.
3. 1, 2, and 3 only.
4. 1, 2, 3, and 4.

Exercise 5.8

While AVL trees measure imbalance by the height of siblings, *weight*-balanced trees use the number of elements in each sibling.

As in an AVL tree, the tree is rebalanced by single or double rotations. A rebalancing is triggered when the ratio of sizes of two subtrees exceeds some predefined factor, called Δ. A double rotation is triggered if, inside the larger subtree, the ratio of sizes of its two children exceed some factor Γ.

Write a function $insert :: Ord\ a \Rightarrow a \to WTree\ a \to WTree\ a$. Your code should define Δ and Γ as constants; you may set them to 3 and 2 respectively for testing.

Describe how the performance characteristics of *insert* and *member* might change if we were to raise or lower Δ and Γ.

```
data WTree a = WTip
             | WNode Int (WTree a) a (WTree a)
```

Exercise 5.9

Define a linear-time function $invariants :: Ord\ a \Rightarrow RBTree\ a \to Bool$ which tests that the following invariants hold on an *RBTree*:

1. It is an ordered binary search tree with no duplicates.
2. Every red node has a black parent.
3. There are the same number of black nodes on any path from root to tip.

```
data Colour = R | B deriving Eq
data RBTree a
  = RBTip
  | RBNode Colour (RBTree a) a (RBTree a)
```

Solutions to the Exercises

Solution 5.1

The *insert x* function makes use of an auxiliary function:

$$ins :: Ord\ a \Rightarrow DTree\ a \rightarrow (Bool, DTree\ a)$$

Given the result $ins\ t = (bl, t')$, t' is the tree that results from inserting x into t. If bl is *True* this indicates that the height of t' is one more than the height of t. Otherwise they have the same height.

$$\begin{aligned}&insert :: Ord\ a \Rightarrow a \rightarrow DTree\ a \rightarrow DTree\ a\\&insert\ x\ t = snd\ (ins\ t)\end{aligned}$$

Now the *ins* function is defined by case analysis on the tree. If it is empty then the height is increased by the insertion of x, and the new tree is balanced.

$$\begin{aligned}&\textbf{where}\\&\quad ins\ DTip = (True, DNode\ Zero\ DTip\ x\ DTip)\end{aligned}$$

Otherwise, the tree is made up of subtrees lt and rt, and case analysis is required between x and y. If $x \equiv y$, then the original tree can be returned, and the height has not changed.

$$\begin{aligned}&ins\ (DNode\ bl\ lt\ y\ rt) = \textbf{case}\ compare\ x\ y\ \textbf{of}\\&\quad EQ \rightarrow (False, DNode\ bl\ lt\ x\ rt)\end{aligned}$$

If $x \leqslant y$ then further analysis is required. The value x should be inserted into lt, and what to do depends on whether the result has increased the height of lt. In the first case, the height is unchanged and so the new tree can be constructed with lt' as the new subtree.

$$\begin{aligned}&LT \rightarrow\\&\quad \textbf{case}\ ins\ lt\ \textbf{of}\\&\qquad (False, lt') \rightarrow (False, DNode\ bl\ lt'\ y\ rt)\end{aligned}$$

Otherwise, the height has changed and some adjustment may be needed, depending on the imbalance indicated by bl of the original tree, so case analysis is required.

$$\begin{aligned}&(True, lt') \rightarrow\\&\quad \textbf{case}\ bl\ \textbf{of}\end{aligned}$$

If the difference was originally -1 then the additional height of lt' will restore balance and the overall height is unchanged:

$$MinusOne \rightarrow (False,\quad DNode\ Zero\quad lt'\ y\ rt)$$

If the tree was originally balanced then the additional height of lt' will add one to the difference, and the new tree height has been adjusted.

$$Zero \rightarrow (True,\quad DNode\ PlusOne\ lt'\ y\ rt)$$

Otherwise, the original tree was already larger in the left side, and the additional of height that side forces a rotation to the right to restore balance. This is achieved through a call to *rotr*, which will be explained later.

$$\begin{array}{ll} \mathit{PlusOne} & \to \mathit{rotr}\ \mathit{lt}'\ y\ \mathit{rt} \end{array}$$

The *GT* case is symmetric, rotating to the left when there is too much imbalance.

$$\begin{array}{l} \mathit{GT} \to \\ \quad \textbf{case}\ \mathit{ins}\ \mathit{rt}\ \textbf{of} \\ \quad\quad (\mathit{False}, \mathit{rt}') \to (\mathit{False}, \mathit{DNode}\ \mathit{bl}\ \mathit{lt}\ y\ \mathit{rt}') \\ \quad\quad (\mathit{True}, \mathit{rt}') \to \\ \quad\quad\quad \textbf{case}\ \mathit{bl}\ \textbf{of} \\ \quad\quad\quad\quad \begin{array}{lll} \mathit{PlusOne} & \to (\mathit{False}, & \mathit{DNode}\ \mathit{Zero}\ \mathit{lt}\ y\ \mathit{rt}') \\ \mathit{Zero} & \to (\mathit{True}, & \mathit{DNode}\ \mathit{MinusOne}\ \mathit{lt}\ y\ \mathit{rt}') \\ \mathit{MinusOne} & \to \mathit{rotl}\ \mathit{lt}\ y\ \mathit{rt}' & \end{array} \end{array}$$

Now time to understand how the rotations work. This is essentially the same as rotations for an AVL tree, except that care must be taken to report whether the height of the tree was changed after the rotation, and whether there remains any imbalance. The first two cases are a fairly straightforward adaption of the code:

$$\begin{array}{l} \mathit{rotr} :: \mathit{DTree}\ a \to a \to \mathit{DTree}\ a \to (\mathit{Bool}, \mathit{DTree}\ a) \\ \mathit{rotr}\ (\mathit{DNode}\ \mathit{PlusOne}\ a\ y\ b)\ x\ c = (\mathit{False}, \mathit{DNode}\ \mathit{Zero}\ \ a\ y\ (\mathit{DNode}\ \mathit{Zero}\ b\ x\ c)) \\ \mathit{rotr}\ (\mathit{DNode}\ \mathit{Zero}\ a\ y\ b)\ x\ c\ \ \ = (\mathit{True}, \mathit{DNode}\ \mathit{MinusOne}\ a\ y\ (\mathit{DNode}\ \mathit{PlusOne}\ b\ x\ c)) \end{array}$$

The final case is where a rotation to the right has been required, but the tree to the left is one shorter than that to the right.

$$\begin{array}{l} \mathit{rotr}\ (\mathit{DNode}\ \mathit{MinusOne}\ a\ y\ (\mathit{DNode}\ \mathit{bl}\ b\ z\ c))\ x\ d = \\ \quad (\mathit{False}, \mathit{DNode}\ \mathit{Zero}\ (\mathit{DNode}\ (\mathit{balr}\ \mathit{bl})\ a\ y\ b)\ z\ (\mathit{DNode}\ (\mathit{ball}\ \mathit{bl})\ c\ x\ d)) \\ \mathit{balr}\ \mathit{PlusOne}\ \ = \mathit{Zero} \\ \mathit{balr}\ \mathit{Zero}\ \ \ \ \ = \mathit{Zero} \\ \mathit{balr}\ \mathit{MinusOne} = \mathit{PlusOne} \\ \mathit{ball}\ \mathit{PlusOne}\ \ = \mathit{MinusOne} \\ \mathit{ball}\ \mathit{Zero}\ \ \ \ \ = \mathit{Zero} \\ \mathit{ball}\ \mathit{MinusOne} = \mathit{Zero} \end{array}$$

The code for *rotl* mirrors that of *rotr*.

$$\begin{array}{l} \mathit{rotl} :: \mathit{DTree}\ a \to a \to \mathit{DTree}\ a \to (\mathit{Bool}, \mathit{DTree}\ a) \\ \mathit{rotl}\ c\ x\ (\mathit{DNode}\ \mathit{MinusOne}\ b\ y\ a) = (\mathit{False}, \mathit{DNode}\ \mathit{Zero}\ (\mathit{DNode}\ \mathit{Zero}\ c\ x\ b)\ y\ a) \\ \mathit{rotl}\ c\ x\ (\mathit{DNode}\ \mathit{Zero}\ b\ y\ a)\ \ \ \ \ \ = (\mathit{True}, \mathit{DNode}\ \mathit{PlusOne}\ (\mathit{DNode}\ \mathit{MinusOne}\ c\ x\ b)\ y\ a) \\ \mathit{rotl}\ d\ x\ (\mathit{DNode}\ \mathit{PlusOne}\ (\mathit{DNode}\ \mathit{bl}\ c\ z\ b)\ y\ a) = \\ \quad (\mathit{False}, \mathit{DNode}\ \mathit{Zero}\ (\mathit{DNode}\ (\mathit{balr}\ \mathit{bl})\ d\ x\ c)\ z\ (\mathit{DNode}\ (\mathit{ball}\ \mathit{bl})\ b\ y\ a)) \end{array}$$

Deleting a node works using analysis similar to *insert*, and is a good exercise to try.

```
deleteD :: Ord a ⇒ a → DTree a → DTree a
deleteD k = snd ∘ del
  where
    del DTip = (True, DTip)
    del (DNode b lt k′ rt) =
      case compare k k′ of
        LT →
          case del lt of
            (True, lt′) → (True, DNode b lt′ k′ rt)
            (False, lt′) →
              case b of
                PlusOne → (False, DNode Zero lt′ k′ rt)
                Zero → (True, DNode MinusOne lt′ k′ rt)
                MinusOne → (rotl lt′ k′ rt)
        GT →
          case del rt of
            (True, rt′) → (True, DNode b lt k′ rt′)
            (False, rt′) →
              case b of
                PlusOne → (rotr lt k′ rt′)
                Zero → (True, DNode PlusOne lt k′ rt′)
                MinusOne → (False, DNode Zero lt k′ rt′)
        EQ →
          case rt of
            DTip → (False, lt)
            DNode br tlr kr trr →
              case b of
                PlusOne →
                  case uncons kr br tlr trr of
                    (k″, False, rt′) →
                      rotr lt k″ rt′
                    (k″, True, rt′) →
                      (True, DNode PlusOne lt k″ rt′)
                Zero →
                  case uncons kr br tlr trr of
                    (k″, False, rt′) →
                      (True, DNode PlusOne lt k″ rt′)
                    (k″, True, rt′) →
                      (True, DNode Zero lt k″ rt′)
                MinusOne →
                  case uncons kr br tlr trr of
                    (k″, False, rt′) →
                      (False, DNode Zero lt k″ rt′)
                    (k″, True, rt′) →
                      (True, DNode MinusOne lt k″ rt′)

uncons
    :: a
    → Diff
    → DTree a
```

```
      → DTree a
      → (a, Bool, DTree a)
uncons k' bl' lt' rt' = go k' bl' lt' rt' id
  where
    go k Zero DTip rt c = uncurry ((,,) k) (c (False, rt))
    go k Zero (DNode bl tll kl trl) rt c =
      go kl bl tll trl
      λcase
        (False, ntl) → c (True, (DNode MinusOne ntl k rt))
        (True, ntl) → c (True, (DNode Zero ntl k rt))
    go k MinusOne DTip rt c = uncurry ((,,) k) (c (False, rt))
    go k MinusOne (DNode bl tll kl trl) rt c =
      go kl bl tll trl
      λcase
        (False, ntl) → c (rotl ntl k rt)
        (True, ntl) → c (True, DNode MinusOne ntl k rt)
    go k PlusOne (DNode bl tll kl trl) rt c =
      go kl bl tll trl
      λcase
        (False, ntl) → c (False, DNode Zero ntl k rt)
        (True, ntl) → c (True, DNode PlusOne ntl k rt)
```

Solution 5.2

The standard solution is as follows:

```
fromOrdList :: [a] → HTree a
fromOrdList xs = fst (go (length xs) xs)
  where
    go :: Int → [a] → (HTree a, [a])
    go 0 xs    = (HTip, xs)
    go n xs1   =
      let
        m              = n `div` 2
        (lhs, x : xs2) = go m xs1
        (rhs, xs3)     = go (n − m − 1) xs2
      in (HNode (1 + max (height lhs) (height rhs)) lhs x rhs, xs3)
    height HTip = 0
    height (HNode x _ _ _) = x
```

However we can also implement it in the continuation-passing style:

```
fromOrdList :: [a] → HTree a
fromOrdList xs = go (length xs) xs const
  where
    go :: Int → [a] → (HTree a → [a] → b) → b
    go 0 xs  k = k HTip xs
    go n xs1 k =
      let m = n `div` 2 in
```

```
    go m              xs1 λlhs (x : xs2) →
    go (n − m − 1) xs2 λrhs xs3      →
    k (HNode (1 + max (height lhs) (height rhs)) lhs x rhs) xs3
  height HTip             = 0
  height (HNode x _ _ _) = x
```

Solution 5.3

```
member :: Ord a ⇒ a → HTree a → Bool
member x HTip = False
member x (HNode _ lte p gt)
    | x ⩽ p      = go p x lte
    | otherwise = member x gt
  where
    go y x HTip = y ⩽ x
    go y x (HNode _ lte p gt)
        | x ⩽ p      = go p x lte
        | otherwise = go y x gt

member :: Ord a ⇒ a → HTree a → Bool
member = go Nothing
  where
    go s x HTip = maybe False (⩽ x) s
    go s x (HNode _ lte p gt)
        | x ⩽ p      = go (Just p) x lte
        | otherwise = go s x gt
```

Solution 5.4

mapt f will produce valid output from valid input if f obeys the following condition (for all x and y):

$$x \leqslant y \Rightarrow f\ x \leqslant f\ y$$

A version of *mapt* which does not require f to follow that condition is the following:

```
mapt :: Ord b ⇒ (a → b) → Tree a → Tree b
mapt f = fromList ∘ map f ∘ toList
```

Solution 5.5

The monoid on ordering we'll need is the monoid for *lexicographic comparisons*.

```
instance Monoid Ordering where
  ϵ       = EQ
  EQ ◇ y = y
  x  ◇ _ = x
```

Solution 5.6

To compare for equality while ignoring the balancing, we can convert to a list:

```
hTreeToList :: HTree a → [a]
hTreeToList rt = go rt [ ]
  where
    go HTip ks = ks
    go (HNode _ lt p gt) ks = go lt (p : go gt ks)
```

And then compare those lists, rather than the trees:

```
instance Eq a ⇒ Eq (HTree a) where
  xs ≡ ys = hTreeToList xs ≡ hTreeToList ys
```

Both *member* and *insert* are well behaved.

hTreeToList is an example of another function which is well behaved. A function which doesn't follow the law is something like:

```
getRoot :: HTree a → Maybe a
getRoot HTip = Nothing
getRoot (HNode _ _ x _) = Just x
```

Solution 5.7

1. Trees
2. Lists
3. Bags
4. Sets

Solution 5.8

The main idea of this question is implemented in *Data.Set*. The values *delta* and *gamma* are set as global constants.

```
delta, gamma :: Int
delta = 3
gamma = 2
```

It helps to have *size* defined, and for there to be a smart constructor for nodes that respect this.

```
size (WTip)           = 0
size (WNode s _ _ _) = s
wnode :: WTree a → a → WTree a → WTree a
wnode l x r = WNode (size l + size r + 1) l x r
```

The *insert* function will insert into the tree and rebalance with the smart constructor *balance* that balances teh tree.

```
insert :: Ord a ⇒ a → WTree a → WTree a
insert x WTip = WNode 1 WTip x WTip
insert x t@(WNode s l y r) = case compare x y of
  LT → balance (insert x l) y r
  GT → balance l y (insert x r)
  EQ → WNode s l x r

balance :: WTree a → a → WTree a → WTree a
balance l x r
| sL + sR ⩽ 1     = WNode s' l x r
| sR > delta * sL = rotl l x r
| sL > delta * sR = rotr l x r
| otherwise       = WNode s' l x r
where
  sL = size l
  sR = size r
  s' = sL + sR + 1

  rotl l x r@(WNode _ ly _ ry)
    | size ly < gamma * size ry = singleL l x r
    | otherwise = doubleL l x r

  rotr l@(WNode _ ly _ ry) x r
    | size ry < gamma * size ly = singleR l x r
    | otherwise = doubleR l x r

  singleL a x (WNode _ b y c) = wnode (wnode a x b) y c
  singleR (WNode _ a y b) x c = wnode a y (wnode b x c)

  doubleL a x (WNode _ (WNode _ b z c) y d) = wnode (wnode a x b) z (wnode c y d)
  doubleR (WNode _ a y (WNode _ b z c)) x d = wnode (wnode a y b) z (wnode c x d)
```

Increasing either of the constants will mean fewer rotations, but a potentially more imbalanced tree (and vice-versa). 2 and 3 are chosen here because those values maintain the logarithmic asympotics of these functions, and also have good performance empirically.

Solution 5.9

Checking the colour invariants can be done in one pass:

```
colourInvar :: RBTree a → Bool
colourInvar rt = (colour rt ≡ B) ∧ isJust (go rt)
  where
    go :: RBTree a → Maybe Int
    go RBTip = Just 1
    go (RBNode R xs _ ys) = do
      guard (colour xs ≡ B)
      guard (colour ys ≡ B)
      x ← go xs
      y ← go ys
      guard (x ≡ y)
      pure x
```

```
  go (RBNode B xs _ ys) = do
    x ← go xs
    y ← go ys
    guard (x ≡ y)
    pure (x + 1)
  colour RBTip = B
  colour (RBNode c _ _ _) = c
```

There are two general approaches to checking the order invariant. Perhaps the clearest is to first define a function that checks that a list is strictly increasing:

```
strictlyIncreasing :: Ord a ⇒ [a] → Bool
strictlyIncreasing (x1 : x2 : xs) = x1 < x2 ∧ strictlyIncreasing (x2 : xs)
strictlyIncreasing _              = True
```

And a function which converts an *RBTree* to a list:

```
rbTreetoList :: RBTree a → [a]
rbTreetoList rt = go rt [ ]
  where
    go RBTip                ks = ks
    go (RBNode _ lt p gt) ks = go lt (p : go gt ks)
```

And use them in combination (i.e. *orderInvar* = *strictlyIncreasing* ∘ *rbTreeToList*).

If we give the tree a *Foldable* instance, it is possible to *deforest* the intermediate list:

```
instance Foldable RBTree where
  foldr f b rt = go rt b
    where
      go RBTip b = b
      go (RBNode _ lt p gt) b = go lt (f p (go gt b))
orderInvar :: Ord a ⇒ RBTree a → Bool
orderInvar xs = foldr (f ∘ Just) (const True) xs Nothing
  where
    f y k x = x < y ∧ k y
```

And finally, there is a way that recurses on the tree itself more directly:

```
orderInvar :: Ord a ⇒ RBTree a → Bool
orderInvar = isJust ∘ go
  where
    go RBTip = Just Nothing
    go (RBNode _ lt x gt) = do
      lt′ ← go lt
      gt′ ← go gt
      guard (maybe True (λ(_, s) → s < x) lt′)
      guard (maybe True (λ(s, _) → x < s) gt′)
      pure (Just (maybe x fst lt′, maybe x snd gt′))
```

This function returns the extrema of a valid binary search tree as an intermediate step.

Regardless of the particular implementation of *orderInvar* chosen, the function *invariants* can be implemented as follows:

```
invariants rt = orderInvar rt ∧ colourInvar rt
```

Extended Red-Black Tree implementation

A previous version of this exercise sheet asked for an implementation of *delete* on red-black trees. This was quickly removed once it became clear that it would take far too much time to answer. However, it is still an instructive algorithm, so we will include the solution here.

First we need some of the basic functions already used for *insert*:

```
balance :: Colour → RBTree a → a → RBTree a → RBTree a
balance B (RBNode R (RBNode R a x b) y c) z d = RBNode R (RBNode B a x b) y (RBNode B c z d)
balance B (RBNode R a x (RBNode R b y c)) z d = RBNode R (RBNode B a x b) y (RBNode B c z d)
balance B a x (RBNode R (RBNode R b y c) z d) = RBNode R (RBNode B a x b) y (RBNode B c z d)
balance B a x (RBNode R b y (RBNode R c z d)) = RBNode R (RBNode B a x b) y (RBNode B c z d)
balance c lt p gt = RBNode c lt p gt

blacken :: RBTree a → RBTree a
blacken (RBNode R lt p gt) = RBNode B lt p gt
blacken t = t

redden :: RBTree a → RBTree a
redden (RBNode _ a x b) = RBNode R a x b
redden RBTip = error "cannot redden leaf"

insert :: Ord a ⇒ a → RBTree a → RBTree a
insert x xs = blacken (ins xs)
  where
    ins RBTip = RBNode R RBTip x RBTip
    ins (RBNode c lt p gt) = case compare x p of
      LT → balance c (ins lt) p gt
      EQ → RBNode c lt p gt
      GT → balance c lt p (ins gt)
```

Then, to write *delete*, we need to handle several more imbalanced cases. The algorithm this implementation is based was written by Matt Might, although his solution accomplished its task by adding two new colours (double black and "negative black", which we call double red), which we avoid here. Adding two new colours would mean infecting the insertion code in a way it doesn't need to be. Instead, we define the following types:

```
data Doubling = S | D

data Doubled a = Doubling ▶ a

single :: Doubled a → a
single (_ ▶ x) = x
```

This allows us to have "double black" as $D \blacktriangleright B$, and a tree with a double black root would be $D \blacktriangleright \mathit{RBNode\ B\ lt\ p\ gt}$. This works since it is only ever the root elements which get the "double" colours.

Other than that change, the rest of the functions are relatively unchanged from Might's algorithm. First we have the double balance:

```
redderTree :: Doubled (RBTree a) → Doubled (RBTree a)
redderTree (S ▶ RBTip) = error "cannot redder single B leaf"
redderTree (D ▶ RBTip) = (S ▶ RBTip)
redderTree (S ▶ RBNode R l x r) = (D ▶ RBNode R l x r)
redderTree (S ▶ RBNode B l x r) = (S ▶ RBNode R l x r)
redderTree (D ▶ RBNode B l x r) = (S ▶ RBNode B l x r)
redderTree (D ▶ RBNode R _ _ _) = error "cannot redder negative B RBNode"

blacker :: Doubled Colour → Doubled Colour
blacker (D ▶ R) = (S ▶ R)
blacker (S ▶ R) = (S ▶ B)
blacker (S ▶ B) = (D ▶ B)
blacker (D ▶ B) = error "cannot blacker double black"

balanceD :: Doubled Colour → Doubled (RBTree a) → a → Doubled (RBTree a) → Doubled (RBTree a)
balanceD (S ▶ c) (_ ▶ l) x (_ ▶ r) = (S ▶ balance c l x r)
balanceD (D ▶ B) (S ▶ RBNode R (RBNode R a x b) y c) z (_ ▶ d) =
  (S ▶ RBNode B (RBNode B a x b) y (RBNode B c z d))
balanceD (D ▶ B) (S ▶ RBNode R a x (RBNode R b y c)) z (_ ▶ d) =
  (S ▶ RBNode B (RBNode B a x b) y (RBNode B c z d))
balanceD (D ▶ B) (_ ▶ a) x (S ▶ RBNode R (RBNode R b y c) z d) =
  (S ▶ RBNode B (RBNode B a x b) y (RBNode B c z d))
balanceD (D ▶ B) (_ ▶ a) x (S ▶ RBNode R b y (RBNode R c z d)) =
  (S ▶ RBNode B (RBNode B a x b) y (RBNode B c z d))
balanceD (D ▶ B) (S ▶ a) x (D ▶ RBNode R (RBNode B b y c) z d@(RBNode B _ _ _)) =
  (S ▶ RBNode B (RBNode B a x b) y (balance B c z (redden d)))
balanceD (D ▶ B) (D ▶ RBNode R a@(RBNode B _ _ _) x (RBNode B b y c)) z (S ▶ d) =
  (S ▶ RBNode B (balance B (redden a) x b) y (RBNode B c z d))
balanceD (d ▶ c) (_ ▶ lt) p (_ ▶ gt) = (d ▶ RBNode c lt p gt)
```

And from this we can implement *delete*:

```
removeMax :: RBTree a → (a, Doubled (RBTree a))
removeMax (RBNode c l x RBTip) = (x, remove c l RBTip)
removeMax (RBNode c l x r) = fmap (bubble (S ▶ c) (S ▶ l) x) (removeMax r)
removeMax RBTip = error "cannot remove max from a leaf"

remove :: Colour → RBTree a → RBTree a → Doubled (RBTree a)
remove R RBTip RBTip = (S ▶ RBTip)
remove B RBTip RBTip = (D ▶ RBTip)
remove B RBTip (RBNode R a x b) = (S ▶ RBNode B a x b)
remove B (RBNode R a x b) RBTip = (S ▶ RBNode B a x b)
remove c l r = bubble (S ▶ c) l' mx (S ▶ r)
  where
    (mx, l') = removeMax l

bubble :: Doubled Colour → Doubled (RBTree a) → a → Doubled (RBTree a) → Doubled (RBTree a)
```

```
bubble c l@(D ▶ RBTip) x r = balanceD (blacker c) (redderTree l) x (redderTree r)
bubble c l@(D ▶ (RBNode B _ _ _)) x r = balanceD (blacker c) (redderTree l) x (redderTree r)
bubble c l x r@(D ▶ (RBNode B _ _ _)) = balanceD (blacker c) (redderTree l) x (redderTree r)
bubble c l x r@(D ▶ RBTip) = balanceD (blacker c) (redderTree l) x (redderTree r)
bubble c l x r = balanceD c l x r

delete :: Ord a ⇒ a → RBTree a → RBTree a
delete x xs = blacken (single (del xs))
  where
    del RBTip = (S ▶ RBTip)
    del (RBNode c lt y gt) = case compare x y of
      LT → bubble (S ▶ c) (del lt) y (S ▶ gt)
      EQ → remove c lt gt
      GT → bubble (S ▶ c) (S ▶ lt) y (del gt)
```

A point to note is that there are a few optimisations one could make to the balancing functions: we never need to check both sides for a double colour, as we know which side is changed after an insertion. Therefore we could have functions *balancel* and *balancer* which get called at the appropriate times. For clarity, we have not applied this optimisation here.