Red-black Trees: A Pure Functional Implementation

Arnau Abella

March 13, 2020

1 Introduction

Although binary search trees work very well on random or unordered data, they perform very poorly on ordered data, for which any individual operation migh take up to $\mathcal{O}(n)$ time.

The solution to this problem is to keep each tree approximately balanced. Then no individual operation takes more than $\mathcal{O}(logn)$ time. There are several implementations of self-balancing binary search trees such as *2-3 tree, AA tree, AVL tree, B-tree, Treap, etc.*

This delivery is about one of the most popular families of self-balancing binary search trees, the *red-black trees* [GS78].

A **red-black tree** is a binary search tree in which every node is colored either red or black.

Every red-black tree satisfy the following two balance invariants:

1. No red node has a red child.

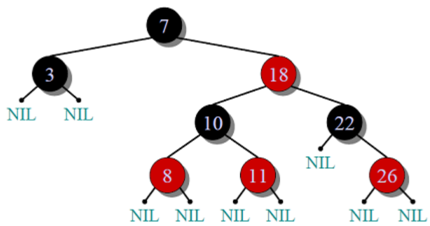


Figure 1: Example of a red-black tree.

2. Every path from the root to an empty node contains the same number of black nodes.

Taken together, these two invariants guarantee that the longest possible path in a red-black tree, one with alternating black and red nodes, is no more than twice as long as the sorthest possible, one with black nodes only.

Theorem 1.1. *The maximum depth of a node in a red-black tree of size* n *is at most* $2\lfloor \log(n+1) \rfloor$

Proof. The maximum number of black nodes in the longest path from the root to a leaf is restricted by the number of nodes in the shortest past (inv. 2).

Assume that the shortest path has depth k. Then, there are $k+1$ nodes in a path of depth k.

From the definition of the shortest path, there is a full and complete binary subtree of depth k (that includes the shortest path). Suposse that the complete binary subtree has n nodes. So, by the definition of a complete binary tree, $n = 2^{k+1} - 1$. Hence, we can compute the number of nodes in the shortest by using the previous equation.

$$n = n^{k+1} - 1$$
$$k + 1 = \log_2(n+1)$$

The number f nodes in the shortest path is $\log_2(n+1)$. So, the largest path can have, at most, $\log_2(n+1)$ black nodes (inv. 2).

Let's make the largest path with $\log_2(n+1)$ black nodes. We want to put as many red nodes as possible because we are limited in the number of black nodes. But, because of the inv. 1, we need to alternate between black and red nodes.

If the root is black, there will be as many red nodes as black ones. The depth k of the largest path is the following:

$$k = \#red \cdot \#black$$
$$k = \log_2(n+1) \cdot \log_2(n+1)$$
$$k = 2\lfloor \log_2(n+1) \rfloor$$
$$k = \mathcal{O}(\log n)$$

$\square$

2 Implementation

The implementation proposed in this first delivery for the **red-black tree** is a persistent representation written in **haskell** [has], an advanced, lazy, purely functional programming language, and based on the implementation from *C. Okasaki* written in *Standard ML* [Oka98].

The proposed implementation has some tweaks *wrt.* the implementation from *C. Okasaki* that improve insertion time. Also, the implementation includes an auxiliar method to construct the *red-black tree* in $\mathcal{O}(n)$ instead of $\mathcal{O}(n \log n)$.

The source code, that can be found in the Appendix A.

Let's have a look at the most interesting parts of the code. First, we will start with the data definition at listing 1, which is pretty straightforward. I included some useful classes instances such as `functor, foldable and traversable`. The attentive readers will notice the exclamation marks in front of the data constructors. This is a language extension called *BangPatterns* that allows changing how an expression is evaluated in haskell.

```haskell
data Color = R -- ^ Red
           | B -- ^ Black
  deriving (Show, Enum)

data RedBlackTree a
  = Bin Color !(RedBlackTree a) !a !(RedBlackTree a)
  | Tip
  deriving (Functor, Foldable, Traversable)
```

Listing 1: Red-black tree data types

The next method we are going to have a look into detail is `insert`. Insert is the hardest method to implement because it must not violate any of the two invariants after the insertion.

To not violate invariant 2 we are going to always insert a **red** node but this may have broken the invariant 1 if the parent was already red so we need to rebalance the tree, recursively.

Figure 2 displays all possible configurations where the invariant 1 is violated.

Figure 3 displays the resulting subtree after the rebalance.

Here is the code of `insert`. You may notice that `balance` method is split in two different cases. This is an optimization because some cases are not possible depending on the branch you came for.

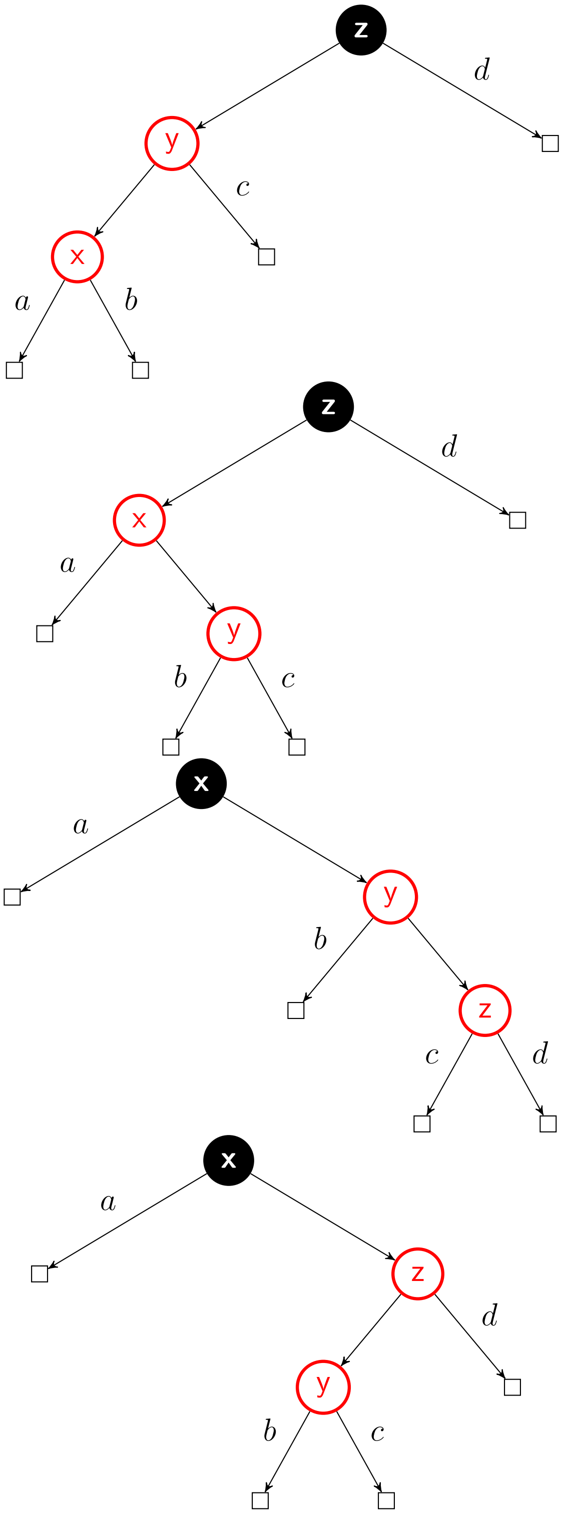


Figure 2: Eliminating red nodes with red parents: before

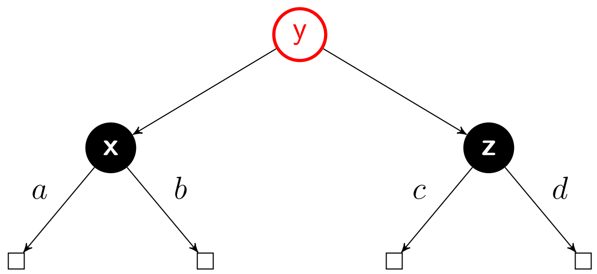


Figure 3: Eliminating red nodes with red parents: after

```haskell
insert
  :: (Ord a)
  => a -> RedBlackTree a -> RedBlackTree a
insert x tree =
  let (Bin _ a y b) = ins tree
  in Bin B a y b
    where
      ins                  Tip  = Bin R Tip x Tip
      ins node@(Bin c a y b) =
        if      x < y   then lbalance (Bin c (ins a) y b)
        else if x > y   then rbalance (Bin c a y (ins b))
        else                 node

lbalance :: RedBlackTree a -> RedBlackTree a
lbalance (Bin B (Bin R (Bin R a x b) y c) z d) =
    Bin R (Bin B a x b) y (Bin B c z d)
lbalance (Bin B (Bin R a x (Bin R b y c)) z d) =
    Bin R (Bin B a x b) y (Bin B c z d)
lbalance tree                                 =
    tree

rbalance :: RedBlackTree a -> RedBlackTree a
rbalance (Bin B a x (Bin R b y (Bin R c z d))) =
    Bin R (Bin B a x b) y (Bin B c z d)
rbalance (Bin B a x (Bin R (Bin R b y c) z d)) =
    Bin R (Bin B a x b) y (Bin B c z d)
rbalance tree                                 =
    tree
```

Listing 2: Red-black tree insert method

Finally, the listing 4 contains the code for the list constructor of a red-black tree.

The code is quite advance but it is basically taking advantage of the fact that we know the position of the node in the given tree, because the precondition of this method is that the given list is ordered, so we can remove the cost of searching the position of the i-th for each i element inserted.

The total cost in time and space of this algorithm is $\mathcal{O}(n)$.

```
fromOrdList :: [a] -> RedBlackTree a
fromOrdList xs' =
  toTree (Tip, ins ([], xs'))
    where
      balance' [(R, v1, t1)] = [(B, v1, t1)]
      balance' ((R, v1, t1):(R, v2, t2):(B, v3, t3):xs) =
        (B, v1, t1):(balance' ((R, v2, (Bin B t3 v3 t2)):xs))
      balance' xs = xs

      ins (ts, [])   = ts
      ins (ts, x:xs) = ins ( balance' ((R, x, Tip):ts), xs)

      toTree (t, [])                =
        t
      toTree (t, ((color, v, t'):ts)) =
        toTree ((Bin color t' v t), ts)
```

Listing 3: Red-black tree fromOrdList method

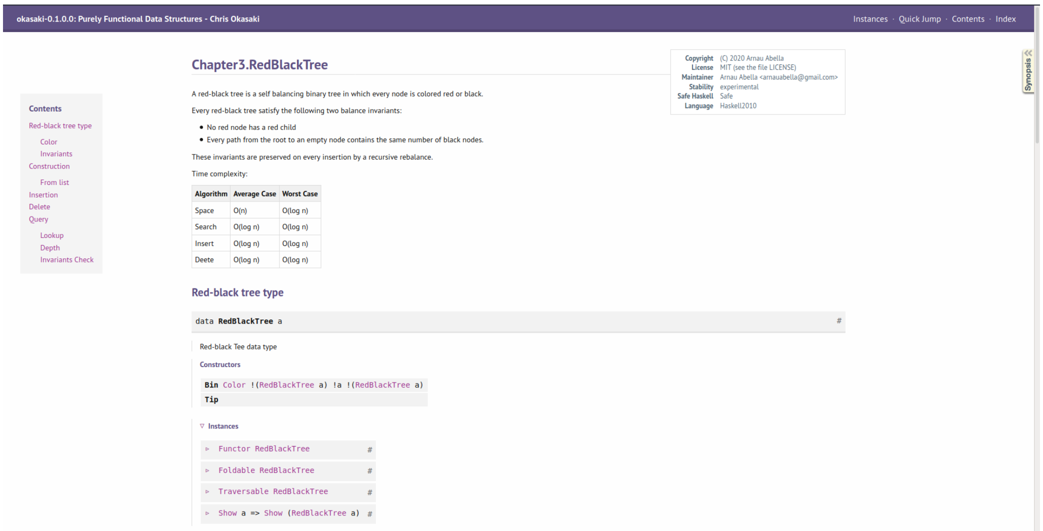
2.1 Documentation

One cool feature of **cabal** [cab], one of the most used building tool in Haskell, is that it can generate documentation *"for free"* of your source code. Underneath, Cabal is using **haddock** [Mar], that can automatically generate documentation from annotated Haskell source code.

Haddock can generate documentation for a myriad of formats including *html* and *latex*. The documentation can be generated by

```
$ cabal haddock
```

Here you can find a screenshot from the generated documentation from RedBlackTree.hs



3 Experimentation

Theoretically, the maximum depth of the tree is $2\lfloor \log(n+1) \rfloor$. Let's test if that fact holds in practice.

Before testing the maximum depth property of the red-black trees, we need to test the correctes of our code.

For this purpose, I created the following tests at `test/Chapter3/RedBlackTreeSpec.hs`

```haskell
spec :: Spec
spec = do
  describe "Red-black tree" $ do
    it "checkInvariants should check if
          invariants are fulfilled" $ do
      RBT.checkInvariants rbt1 `shouldBe` True
      RBT.checkInvariants bad  `shouldBe` False
      RBT.checkInvariants bad2 `shouldBe` False

    it "insert should preserve invariants" $ do
      let rbt = foldr RBT.insert
                      RBT.empty
                      ([1..10000] :: [Int])
      RBT.checkInvariants rbt `shouldBe` True

    it "fromOrdList should preserve invariants" $ do
      let rbt = RBT.fromOrdList [1..10000] :: RedBlackTree Int
      RBT.checkInvariants rbt `shouldBe` True

    it "toOrdList should return an ordered list of the
          elements of the tree" $ do
      let rbt = foldr RBT.insert
                      RBT.empty
                      ([3,5,2,1,8,9,5,4] :: [Int])
      RBT.toOrdList rbt `shouldBe` [1,2,3,4,5,8,9]

    it "toOrdList . fromOrdList == id" $ do
      let xs = [1..1000] :: [Int]
          f = RBT.toOrdList . RBT.fromOrdList
      f xs `shouldBe` xs

    prop "maxDepth = 2*floor[log (n + 1)]" $ do
      property $
        \(n :: Positive Int) ->
          let size = 1000
              rbt = RBT.fromOrdList [1..size]
          in RBT.maxDepth rbt
               <= 2*floor (logBase 2.0 (fromIntegral size + 1.0))
```

Listing 4: Red-black tree fromOrdList method

In order to prove the maximum depth property of the red-black trees we are going to implement the following experiment:

1. Generate n randomly-built red-black trees of size m, where n and m are large enough.
2. Compute the height of the n red-black trees.
3. Compute the max, mean and standard deviation of the depth of each red-black tree.
4. Compare it to the theoretically maximum heigth.

The haskell code to generate the randomly-built red-black tree of size m uses a property-based testing library called QuickCheck [qui] that allows the user to generate a huge amount of uniformly distributed random data.

```haskell
-- | A generator for values of type 'RedBlackTree' of the given size.
genRBT :: (Arbitrary a, Ord a)
       => Int -> Gen (RedBlackTree a)
genRBT = fmap (fromList . unUnique) . genUniqueList

newtype UniqueList a = UniqueList { unUnique :: [a] }
    deriving Show

-- | 90 % of samples are randomly distributed elements
--    10 % are sorted.
genUniqueList :: (Arbitrary a, Ord a)
              => Int -> Gen (UniqueList a)
genUniqueList n =
  frequency [ (9, genUniqueList' n arbitrary)
            , (1, (UniqueList . unSorted) <$>
                      genUniqueSortedList n arbitrary)
            ]

genUniqueList' :: (Eq a) => Int -> Gen a -> Gen (UniqueList a)
genUniqueList' n gen =
  UniqueList <$> vectorOf n gen `suchThat` isUnique

newtype UniqueSortedList a =
    UniqueSortedList { unSorted :: [a] }
  deriving Show

genUniqueSortedList :: (Ord a)
                    => Int -> Gen a -> Gen (UniqueSortedList a)
genUniqueSortedList n gen =
  UniqueSortedList . List.sort . unUnique <$> genUniqueList' n gen

isUnique :: Eq a => [a] -> Bool
isUnique x = List.nub x == x
```

Listing 5: QuickCheck generators.

The code of the experiment is the following

```haskell
-- 1.- Generate a random vector of [n-m, n+m] elements
-- 2.- Transform it into a red-black tree.
-- 3.- Compute the maximum length
-- 4.- Get the statistics
-- 5.- Output them on the stdout
runExperiment
  :: Int -- ^ Number of samples
  -> Int -- ^ Size of the samples
  -> IO ()
runExperiment n size = do
  samples <- generate $ replicateM n (RBT.genRBT @Int size)
  let (Just max', mean, std) =
    L.fold ((,,) <$> L.maximum <*> L.mean <*> L.std) $
                              fmap (fromIntegral . RBT.maxDepth) samples
  report n size max' mean std
```

Listing 6: Experiment of the theoretical depth of a red-black tree.

```
# samples: 100
tree size: 100

max depth(max) : 8.0
max depth(mean): 7.720000000000001
max depth(std) : 0.448998886412873

max depth  (theoretical): 12
perfectly balanced depth: 6
```

Listing 7: Output of the experiment for 100 samples of size 100.

From the output of the experiment, we conclude that the theorical maximum depth holds for an empirical red-black tree. Furthemore, the height is near to a perfectly balanced binary tree.

4 Conclusion

We have coded a simple, yet efficient purely functional implementation of red-black trees in Haskell and have proved that the maximum theoretical depth holds in practice by conducting an empirical experiment over randomly generated red-black trees.

5 Curiosity

One of the reasons this implementation is so much simpler than typical presentations of red-black trees (e.g., Chapter 14 of [CLR90]) is that it uses subtly different rebalancing transformations. Imperative implementations typically split the four dangerous cases considered here into eight cases, according to the color of the sibling of the red node with a red child. Knowing the color of the red parent's sibling allows the transformations to use fewer assignments in some cases and to terminate rebalancing early in others.

However, in a functional setting, where we are copying the nodes in question anyway, we cannot reduce the number of assignments in this fashion, nor can we terminate copying early, so there is no point is using the more complicated transformations.

Appendix A

RedBlackTree.hs

```haskell
{-# LANGUAGE BangPatterns         #-}
{-# LANGUAGE DeriveAnyClass       #-}
{-# LANGUAGE DeriveFoldable       #-}
{-# LANGUAGE DeriveFunctor        #-}
{-# LANGUAGE DeriveTraversable    #-}
{-# LANGUAGE RoleAnnotations      #-}
-----------------------------------------------------------------------------------------
-- |
-- Module      :  Chapter3.RedBlackTree
-- Copyright   :  (C) 2020 Arnau Abella
-- License     :  MIT (see the file LICENSE)
-- Maintainer  :  Arnau Abella <arnauabella@gmail.com>
-- Stability   :  experimental
--
--
-- A red-black tree is a self balancing binary tree
-- in which every node is colored red or black.
--
-- Every red-black tree satisfy the following two balance invariants:
--
--    * No red node has a red child
--    * Every path from the root to an empty node contains the same number of b
--
-- These invariants are preserved on every insertion by a recursive rebalance.
--
-- Time complexity:
--
-- +-----------+--------------+-----------+
-- | Algorithm | Average Case  | Worst Case |
-- +===========+==============+===========+
-- | Space     | O(n)          | O(log n)   |
-- +-----------+--------------+-----------+
-- | Search    | O(log n)      | O(log n)   |
-- +-----------+--------------+-----------+
-- | Insert    | O(log n)      | O(log n)   |
-- +-----------+--------------+-----------+
-- | Deete     | O(log n)      | O(log n)   |
```

```haskell
-- +-----------+--------------+-----------+
--------------------------------------------------------------------------------
module Chapter3.RedBlackTree (
  -- * Red-black tree type
    RedBlackTree (..)
  -- ** Color
  , Color(..)
  -- ** Invariants
  , InvariantException(..)
  -- * Construction
  , empty
  -- ** From list
  , fromList
  , fromOrdList
  , toOrdList
  -- * Insertion
  , insert
  -- * Delete
  , delete
  -- * Query
  -- ** Lookup
  , lookup
  , member
  -- ** Depth
  , depth
  , minDepth
  , maxDepth
  -- ** Invariants Check
  , checkInvariants
  -- * Testing
  , genRBT
  ) where

----------------------------------------------------------------------------

import           Control.Exception
import           Data.Maybe        (isJust)
import           Prelude           hiding (lookup)
import           Test.QuickCheck
import qualified Data.List as List
```

```
--------------------------------------------------------------------------

-- | Red-black Tee data type
data RedBlackTree a
  = Bin Color !(RedBlackTree a) !a !(RedBlackTree a)
  | Tip
  deriving (Functor, Foldable, Traversable)

instance Show a => Show (RedBlackTree a) where
  show = drawTree

type role RedBlackTree nominal

-- | Color of each node.
data Color = R -- ^ Red
           | B -- ^ Black
  deriving (Show, Enum)

-- | Returns an empty red-black tree.
empty :: RedBlackTree a
empty = Tip
{-# INLINEABLE empty #-}

-- | Return the element if it is present in the tree.
-- Otherwise, returns nothing.
--
-- Cost: O(log n)
lookup
  :: (Ord a)
  => a -> RedBlackTree a -> Maybe a
lookup = go
  where
    go _ Tip = Nothing
    go x (Bin _ l y r) =
      if x < y          then go x l
      else if x > y     then go x r
      else                   Just y
{-# INLINEABLE lookup #-}

-- | Checks if the element 'a' is present in the given red-black tree.
--
```

```haskell
-- Cost: O(log n)
member
  :: (Ord a)
  => a -> RedBlackTree a -> Bool
member a = isJust . lookup a
{-# INLINEABLE member #-}

-- | Inserts an element while keeping the two invariants.
--
-- Cost: O(log n)
insert
  :: (Ord a)
  => a -> RedBlackTree a -> RedBlackTree a
insert x tree =
  let (Bin _ a y b) = ins tree
  in Bin B a y b
    where
      ins               Tip  = Bin R Tip x Tip
      ins node@(Bin c a y b) =
        if      x < y   then lbalance (Bin c (ins a) y b)
        else if x > y   then rbalance (Bin c a y (ins b))
        else                 node
{-# INLINEABLE insert #-}

-- | Balance the tree after a recursive insert on the left subtree.
--
-- Cost: O(1)
lbalance :: RedBlackTree a -> RedBlackTree a
lbalance (Bin B (Bin R (Bin R a x b) y c) z d) = Bin R (Bin B a x b) y (Bin B c
lbalance (Bin B (Bin R a x (Bin R b y c)) z d) = Bin R (Bin B a x b) y (Bin B c
lbalance tree                                  = tree
{-# INLINEABLE lbalance #-}


-- | Balance the tree after a recursive insert on the right subtree.
--
-- Cost: O(1)
rbalance :: RedBlackTree a -> RedBlackTree a
rbalance (Bin B a x (Bin R b y (Bin R c z d))) = Bin R (Bin B a x b) y (Bin B c
rbalance (Bin B a x (Bin R (Bin R b y c) z d)) = Bin R (Bin B a x b) y (Bin B c
rbalance tree                                  = tree
```

```haskell
{-# INLINEABLE rbalance #-}


delete
 :: (Ord a)
 => a -> RedBlackTree a -> RedBlackTree a
delete = error "Not implemented."
{-# INLINEABLE delete #-}

-- | Construct a red-black tree from an arbitrary list.
--
-- Cost: O(n*log(n))
--
-- If you **know** that the list is sorted, use 'fromOrdList'.
fromList :: (Ord a) => [a] -> RedBlackTree a
fromList = foldr insert empty

-- | Given an ordered list with no duplicate, returns a red-black tree.
--
-- @
-- fromOrdList [1.10000]
-- @
--
-- Cost: O(n) (notice this is faster than n inserts.)
fromOrdList :: [a] -> RedBlackTree a
fromOrdList xs' =
  toTree (Tip, ins ([], xs'))
    where
      balance' :: [(Color, a, RedBlackTree a)] -> [(Color, a, RedBlackTree a)]
      balance'                               [(R, v1, t1)] = [(B, v1, t1)]
      balance'   ((R, v1, t1):(R, v2, t2):(B, v3, t3):xs) = (B, v1, t1):(balance
      balance'                                          xs = xs

      -- ^^^^ Use the list of (color, value, left sub-tree) in order to avoid s
      --      This list represents the right spine of the red-black tree from t
      ins :: ( [(Color, a, RedBlackTree a)], [a] ) -> [(Color, a, RedBlackTree a
      ins (ts, [])    = ts
      ins (ts, x:xs) = ins ( balance' ((R, x, Tip):ts), xs)

      toTree :: ( RedBlackTree a, [(Color, a, RedBlackTree a)] ) -> RedBlackTre
      toTree (t, [])                  = t
```

```haskell
      toTree (t, ((color, v, t'):ts)) = toTree ((Bin color t' v t), ts)
      -- The amortized cost of ins is O(1), and ins is called n times.
      -- The complexity of toTree is O(log(n)).
      -- The total complexity is n*O(1) + O(log(n)) = O(n).
{-# INLINEABLE fromOrdList #-}

-- | Given a red-black tree, returns an ordered list with no duplicates.
toOrdList :: RedBlackTree a -> [a]
toOrdList           Tip = []
toOrdList (Bin _ l y r) = toOrdList l ++ [y] ++ toOrdList r
{-# INLINEABLE toOrdList #-}

-- | The depth of a node is the number of edges from the node to the root.
--
-- 'heigth' and 'depth' are equivalent in this context.
--
-- The heigth and depth are properties of the nodes, not of the trees.
-- e.g. on a tree of size 3, a node of heigth 2 has depth 0, and viceversa.
depth :: (Int -> Int -> Int) -> RedBlackTree a -> Int
depth choice = (subtract 1) . go -- ^ The root has depth 0
  where
    go           Tip = 0
    go (Bin _ l _ r) = choice (go l) (go r) + 1
{-# INLINEABLE depth #-}

-- | Depth of the shortest path in the red-black tree from the root to the lea
--
-- Cost O(n)
minDepth :: RedBlackTree a -> Int
minDepth = depth min
{-# INLINEABLE minDepth #-}


-- | Depth of the largest path in the red-black tree from the root to the leaf
--
-- Cost O(n)
maxDepth :: RedBlackTree a -> Int
maxDepth = depth max
{-# INLINEABLE maxDepth #-}

data InvariantException
```

```haskell
    = Invariant1Exception -- ^ No red node has a red child
    | Invariant2Exception -- ^ Every path from the root to an empty node contai
  deriving (Show, Exception)

-- | Every red-black tree satisfy the following two balance invariants:
--   * No red node has a red child
--   * Every path from the root to an empty node contains the same number of bl
checkInvariants :: RedBlackTree a -> Bool
checkInvariants t = go B 0 t
  where
    blackNodes = countBlackNodes t -- arbitrary branch

    bothRed R R = True
    bothRed _ _ = False

    go _ acc Tip = acc == blackNodes
    go c1 !acc (Bin c2 l _ r)
      | bothRed c1 c2 = False
      | otherwise =
          let l' = go c2 (acc + fromEnum c2) l
              r' = go c2 (acc + fromEnum c2) r
          in l' && r'
{-# INLINEABLE checkInvariants #-}


-- | Count the black nodes of the right most path.
countBlackNodes :: RedBlackTree a -> Int
countBlackNodes = go 0
  where
    go !acc Tip           = acc
    go !acc (Bin c _ _ r) = go (acc + fromEnum c) r
{-# INLINEABLE countBlackNodes #-}


-- | Neat 2-dimensional drawing of a tree.
drawTree :: (Show a) => RedBlackTree a -> String
drawTree  = unlines . draw

-- | Inner drawing of a tree
draw :: (Show a) => RedBlackTree a -> [String]
draw Tip = ["x"]
```

```haskell
draw (Bin c l x r) =
  (show x ++ "[" ++ show c ++ "]") : drawSubTrees [l, r]
    where
      shift first other =
        zipWith (++) (first : repeat other)

      drawSubTrees []     = []
      drawSubTrees [t]    = "|" : shift "l- " "   " (draw t)
      drawSubTrees (t:ts) = "|" : shift "r- " "|  " (draw t) ++ drawSubTrees ts


-- | A generator for values of type 'RedBlackTree' of the given size.
genRBT :: (Arbitrary a, Ord a) => Int -> Gen (RedBlackTree a)
genRBT = fmap (fromList . unUnique) . genUniqueList -- Otherwise the size would

newtype UniqueList a = UniqueList { unUnique :: [a] }
    deriving Show

-- | 90 % of samples are randomly distributed elements
--   10 % are sorted.
genUniqueList :: (Arbitrary a, Ord a) => Int -> Gen (UniqueList a)
genUniqueList n =
  frequency [ (9, genUniqueList' n arbitrary)
            , (1, (UniqueList . unSorted) <$> genUniqueSortedList n arbitrary)
            ]

genUniqueList' :: (Eq a) => Int -> Gen a -> Gen (UniqueList a)
genUniqueList' n gen =
  UniqueList <$> vectorOf n gen `suchThat` isUnique

newtype UniqueSortedList a = UniqueSortedList { unSorted :: [a] }
    deriving Show

genUniqueSortedList :: (Ord a) => Int -> Gen a -> Gen (UniqueSortedList a)
genUniqueSortedList n gen =
  UniqueSortedList . List.sort . unUnique <$> genUniqueList' n gen

isUnique :: Eq a => [a] -> Bool
isUnique x = List.nub x == x
```

References

[cab] Cabal: User guide. https://cabal.readthedocs.io/en/latest/.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. *IEE Symposium of Foundations of Computer Science*, pages 8–21, 1978.

[has] Haskell: An advanced, purely functional programming language. https://www.haskell.org/.

[Mar] Simon Marlow. Haddock: User guide. https://haskell-haddock.readthedocs.io/en/latest/index.html.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[qui] Quickcheck: Automatic testing of haskell programs. https://hackage.haskell.org/package/QuickCheck-2.13.2.