Haskell Practice Test
Binary Radix Trees

The maximum mark is 25.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

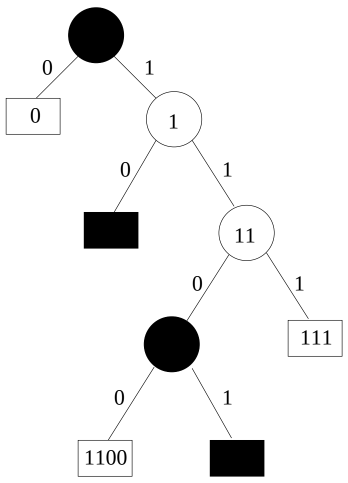
WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You are therefore advised to define your own tests to complement the ones provided.

Introduction

A (binary) radix tree is a special type of binary tree which is used to store a set of arbitrary-length bit strings. A bit string is a list of integers each of which is either 0 or 1. The edges emanating from each internal node correspond to binary digits (bits) and the rule is that the left outgoing edge is notionally numbered 0 and the right edge is notionally numbered 1. Any non-zero length path through a radix tree starting at the root therefore corresponds to a unique sequence of 0's and 1's and therefore a unique bit string.

Radix trees can be used to store bit strings by marking the nodes and leaves either *white* or *black*. A path from the root to a white node or a white leaf, corresponding to a bit string B, indicates that B is *encoded* in the tree. If a path B ends at a black node or a black leaf, it indicates that B is not encoded in the tree.

Note that a bit string corresponds to the binary representation of an integer, so a radix tree can be seen as a data structure for storing a set of integers: if an integer is encoded in the tree then it is a member of the set of integers associated with the tree. In this exercise all integers will be non-negative.



As an example the radix tree shown in the figure above encodes the bit strings 0, 1, 11, 111 and 1100 corresponding to the integers 0, 1, 3, 7 and 12. Each is arrived at by following a path from the root to a white node or leaf. For example, 1100 (the integer 12) is encoded by virtue of the fact that the path right (1), right (1), left (0), left (0) leads to a white leaf. The integers 2 and 6 (bit strings 10 and 110 respectively) are not encoded in the tree, as the corresponding paths lead to a black leaf and black node respectively.

The encoded bit strings are drawn in the respective nodes and leaves but this is for illustrative purposes only. It is not necessary to store them explicitly. It is also not necessary to label the edges explicitly—left always corresponds to 0 and right to 1.

Representation

The following Haskell data type and type synonyms can be used to represent both conventional binary trees containing integers (type `IntTree`) and radix trees (type `RadixTree`). For the former the leaves and nodes have an associated integer value, i.e. the type variable `a` is `Int`. For the latter `a` is the data type `Bit` and this is used to mark the leaves and nodes: by convention, `One` corresponds to white and `Zero` to black.

```
data Tree a = Empty | Leaf a | Node a (Tree a) (Tree a)
            deriving (Eq, Show)

type IntTree = Tree Int

data Bit = Zero | One
         deriving (Eq, Show)

type RadixTree = Tree Bit

type BitString = [Int]
```

As an example, the radix tree in the figure above will be represented by:

```
Node Zero (Leaf One)
          (Node One (Leaf Zero)
                    (Node One (Node Zero (Leaf One)
                                         (Leaf Zero))
                              (Leaf One)))
```

Note that radix trees do not make use of the `Empty` constructor, so an "empty" radix tree corresponds to `Leaf Zero`, rather than `Empty`. The example above is included in the template as `figure :: RadixTree`.

An equivalent `IntTree` containing the same integers (0, 1, 3, 7, 12) might be following:

```
Node 1 (Leaf 0) (Node 3 Empty (Node 7 Empty (Leaf 12)))
```

There are many other possibilities, depending on how the tree is built. A function, `buildIntTree`, for building an `IntTree`, is provided in the template.

The final goal of this exercise is to compare the space efficiency of radix trees and conventional binary trees when storing a given set of integers. Your main task is to define functions for building and manipulating radix trees, so that you can perform the comparison.

What to do

In the following there is a universal precondition that the integers are non-negative.

1. Define functions `and :: Bit -> Bit -> Bit` and `or :: Bit -> Bit -> Bit` that implement the equivalent of boolean and (`&&`) and or (`||`) respectively, e.g.

   ```
   *Radix> and One One
   One
   *Radix> or Zero One
   One
   ```

   Note: the existing prelude functions `and` and `or` are hidden via the import statement at the top of the template. **[1 marks]**

2. Define a function `sizeIT :: IntTree -> Int` that will compute the number of bytes of memory that a "smart" Haskell implementation will use to represent an `IntTree`. In this smart implementation each `Empty` tree occupies 1 byte, each `Leaf` node occupies 4 bytes and each `Node` occupies 12 bytes.

   Now do the same for radix trees by defining a function `sizeRT :: RadixTree -> Int`. The "smart" implementation of radix trees occupies less memory per node because the nodes contain no values – just a boolean, which encodes the node's marker (white or black). Each `Leaf` node, including its marker, now occupies 1 byte and each internal `Node` occupies 8 bytes. Recall that radix trees do not make use of the `Empty` constructor. For example (`t1 :: IntTree` and `t2 :: RadixTree` are defined in the template):

```
*Radix> sizeIT t1
40
*Radix> sizeRT t2
28
```

[2 marks]

3. Define a *tail-recursive* function `binary :: Int -> BitString`, i.e. one that uses a helper function with type `bin :: Int -> [Int] -> [Int]` (or similar) which will convert an integer into its binary representation as a `BitString`, without leading zeros. A bit string here will be represented by a list of integers, each being either 0 or 1. For example,

   ```
   *Main> binary 0
   [0]
   *Main> binary 13
   [1,1,0,1]
   ```

   Note: The maximum mark for a non tail-recursive verion is 2. **[3 marks]**

4. Define a function `insert :: BitString -> RadixTree -> RadixTree` which will add a bit string to a radix tree. If the original tree encodes the set of `BitStrings` S then inserting b should yield a tree which encodes the set $S \cup \{b\}$. The insertion rules are:

   (a) Inserting an empty bit string into a radix tree will cause the topmost `Node` or `Leaf` to be marked white.

   (b) Inserting a non-empty bit string into a `Node` will cause the insertion to proceed in either the left or right sub-tree depending on the most significant (left-most) bit of the bit string. If this is 0 the remainder of the bit string (i.e. with the 0 removed) is inserted into the left sub-tree; otherwise the remainder (with the 1 removed) is inserted into the right sub-tree. The node's marker is unchanged.

   (c) Inserting a non-empty bit string into a `Leaf` has the effect of creating a new internal `Node` with the same marker as the original `Leaf` and two black `Leaf` sub-trees. The insertion then continues either left or right, as per the previous rule.

   For example,

   ```
   *Radix> insert [0] (Leaf Zero)
   Node Zero (Leaf One) (Leaf Zero)
   *Radix> insert [1] (Leaf Zero)
   Node Zero (Leaf Zero) (Leaf One)
   *Radix> insert [0] (insert [1] (Leaf Zero))
   Node Zero (Leaf One) (Leaf One)
   *Radix> insert [1,0] (insert [1] (Leaf Zero))
   Node Zero (Leaf Zero) (Node One (Leaf One) (Leaf Zero))
   ```

   [6 marks]

5. Define a function `buildRadixTree :: [Int] -> RadixTree` which uses `insert` and `binary` to build a radix tree from a given list of integers. For example,

   ```
   *Radix> buildRadixTree []
   Leaf Zero
   *Radix> buildRadixTree [5,3,2]
   Node Zero (Leaf Zero) (Node Zero (Node One (Leaf Zero) (Leaf One)) (Leaf One))
   *Main> buildRadixTree [0,1,3,7,12] == figure
   True
   *Main> buildRadixTree [12,0,3,1,7] == figure
   True
   ```

[4 marks]

6. Define a function `member :: Int -> RadixTree -> Bool` that will determine whether a given integer is a member of the set of integers encoded in a given radix tree. For example,

```
*Main> member 4 (buildRadixTree [1,3,7])
False
*Main> member 7 (buildRadixTree [1,3,7])
True
```

[4 marks]

7. Define the functions `union, intersection :: RadixTree -> RadixTree -> RadixTree` that will respectively compute the union and intersection of two radix trees. For example,

```
*Radix> rt1 = buildRadixTree [1,3,7]
*Radix> rt2 = buildRadixTree [2,3,4,7]
*Radix> union rt1 rt2
Node Zero (Leaf Zero) (Node One (Node One (Leaf One) (Leaf Zero)) (Node One
(Leaf Zero) (Leaf One)))
*Radix>
*Radix> [member n (union rt1 rt2) | n <- [1,2,3,4,7,9]]
[True,True,True,True,True,False]
*Radix> intersection rt1 rt2
Node Zero (Leaf Zero) (Node Zero (Leaf Zero) (Node One (Leaf Zero) (Leaf One)))
```

[4 marks]

8. The template file also defines a (constant) list of random (positive) integers, `rs`. By experimenting with increasingly longer prefixes of `rs` (e.g. via `take n rs`) find the value of `n` beyond which it becomes more economical, in terms of bytes of memory, to use radix trees than it is to use conventional `IntTree`s. Edit the comment at the end of the template file:

```
   --
   -- CONCLUSION: Break-even point is n = xxx
   --
```

and fill in the break-even point in place of `xxx`.

[1 mark]

You can use this page as draft