Assignment 2 - Red-Black Tree Set Implementation

Rory Stephenson 300160212 stepherory

14 April 2014

Introduction

Note: Comments for functions which also exist in `Set` will only comment be included where there are differences in the implementations/implications of their counterparts.

This module provides an implemention of sets for which the underlying data structure is a red-black tree. Chris Okasaki's method for functional red black tree insert is used. It has been modified to remove duplicates and to fit this module's data constructor which includes the notion of double black and negative black node colours.

Double black and negative black colours are an artifact of Matt Might's functional red-black tree deletion method. Rather than outlining lengthy processes for dealing with the more difficult red-black tree removal cases, nodes can be marked with double/negative blacks and then handled easily with follow up actions to remove these colours. The tree is still a red-black tree as the two added shades of black only exist during the removal process, they are not present once a deletion is complete.

Module

```
module BSet (
  Set,
  add,
  remove,
  isEmpty,
  empty,
  singleton,
```

```haskell
  member,
  size,
  isSubsetOf,
  union,
  intersection,
  difference,
  filter,
  valid,
  map) where

import Prelude hiding(foldr, filter, map)
import Data.Foldable
```

The Set constructor has been modified to include colours in the nodes and leaves. The instance declarations have not changed but because the red-black tree is balanced the fact that fold is implemented in order no longer results in a linked list when fold is used to add to a set.

Core

```haskell
data Set a = EmptySet Colour | Node a (Set a) (Set a) Colour
data Colour = B | R | BB | NB

instance Show a => Show (Set a) where
    show a = "{" ++ drop 2 (foldr (\e r -> ", " ++ show e ++ r) "" a) ++ "}"

instance Foldable Set where
  foldr _ z (EmptySet _) = z
  foldr f z (Node k l r _ ) = foldr f (f k (foldr f z r)) l

instance Ord a => Eq (Set a) where
  (EmptySet _) == (EmptySet _) = True
  a == b = (foldr (\e r -> r && (member e b)) True a) &&
           (foldr (\e r -> r && (member e a)) True b)
```

add has been modified to include fixing of the structure which results from adding a new node. This is required to balance the tree and maintain the validity of the structure.

fix is called on the ancestors of the node which is inserted/removed so that the fix function can operate on the parents, grandparents and siblings of the new/removed node. Each case of fix is commented to describe what transformation is taking place. These include rotations and other transformation which will either leave the tree in a valid state or leave it in a state which a higher up call of `fix` will repair.

```
add :: Ord a => a -> Set a -> Set a
add e s = toB (addHelper e s)

addHelper :: Ord a => a -> Set a -> Set a
addHelper a (EmptySet _) = Node a (EmptySet B) (EmptySet B) R
addHelper a (Node k l r c) | a < k = fix k (addHelper a l) r c
addHelper a (Node k l r c) | a == k = Node k l r c
addHelper a (Node k l r c) | otherwise = fix k l (addHelper a r) c

fix :: a -> Set a -> Set a -> Colour -> Set a
fix e (Node e1 (Node e2 l2 r2 R) r1 R) r c | isBorBB c =
  Node e1 (Node e2 l2 r2 B) (Node e r1 r B) (subtractB c)
   -- lchild to parent, llchild to B sibling (right rotate)
fix e (Node e2 l2 (Node e1 r2 r1 R) R) r c | isBorBB c =
  Node e1 (Node e2 l2 r2 B) (Node e r1 r B) (subtractB c)
   -- lchild to B sibling, lrchild to parent
fix e2 l2 (Node e (Node e1 r2 r1 R) r R) c | isBorBB c =
  Node e1 (Node e2 l2 r2 B) (Node e r1 r B) (subtractB c)
   -- rchild to B sibling, rlchild to parent
fix e2 l2 (Node e1 r2 (Node e r1 r R) R) c | isBorBB c =
  Node e1 (Node e2 l2 r2 B) (Node e r1 r B) (subtractB c)
   -- rchild to parent, rr child to B sibling (left rotate)
fix e2 l2 (Node e (Node e1 r2 r1 B) r@(Node _ _ _ B) NB) BB =
  Node e1 (Node e2 l2 r2 B) (fix e r1 (toR r) B) B
   -- Remove BB and NB by push up right side
fix e (Node e2 l2@(Node _ _ _ B) (Node e1 r2 r1 B) NB) r BB =
  Node e1 (fix e2 (toR l2) r2 B) (Node e r1 r B) B
   -- Remove BB and NB by push up left side
fix e l r c = Node e l r c -- No consecutive Reds thus no fixing required
```

`remove` has been modified for this red-black implementation in a similar way to `add`. The difference is that `bubble` is called which in turn calls fix. `bubble` attempts to eliminate a double black by recolouring its parent, otherwise it bubbles the double black up to its parent until it can be fixed. `rmove` gives the immediate result of removing a node which is then passed up to bubble etc.

```haskell
remove :: Ord a => a -> Set a -> Set a
remove e s = toB(removeHelper e s)

removeHelper :: Ord a => a -> Set a -> Set a
removeHelper e node@(Node v _ _ _) | e == v = rmove (node)
removeHelper e (Node v l r c) | e < v = bubble v (removeHelper e l) r c
-- Keep recursing down the left child
removeHelper e (Node v l r c) | e > v = bubble v l (removeHelper e r) c
-- keep recursing down the right child
removeHelper _ s = s

{- element of node -> node l -> node r -> node colour -> result set -}
bubble :: a -> Set a -> Set a -> Colour -> Set a
bubble v l@(Node _ _ _ BB) r c =
  fix v (subtractBSet l) (subtractBSet r) (addB c)
bubble v l r@(Node _ _ _ BB) c =
  fix v (subtractBSet l) (subtractBSet r) (addB c)
bubble v l@(EmptySet BB) r c =
  fix v (subtractBSet l) (subtractBSet r) (addB c)
bubble v l r@(EmptySet BB) c =
  fix v (subtractBSet l) (subtractBSet r) (addB c)
bubble v l r c = Node v l r c

rmove :: Ord a => Set a -> Set a
rmove (Node _ (EmptySet _) (EmptySet _) R) = EmptySet B
-- Delete R node with no children
rmove (Node _ (EmptySet _) (EmptySet _) B) = EmptySet BB
-- Delete B node with no children
rmove (Node _ lNode@(Node _ _ _ R) (EmptySet _) B) = toB lNode
-- delete B node with one (R) child
rmove (Node _ (EmptySet _) rNode@(Node _ _ _ R) B) = toB rNode
-- delete B node with one (R) child
rmove (Node _ lNode@(Node _ _ _ _) rNode@(Node _ _ _ _) c) =
      bubble (rChild lNode) (removeMax lNode) rNode c
      -- handle delete with two children
rmove s = s
```

Unchanged Functions

The following functions are unchanged aside from the inclusion of colours in their pattern matchings and constructors. There will, however, be a significant difference in the performance of several of the functions. Where fold is used to create a new set, the linked list problem no longer exists which means that the resulting sets will much shorter trees (and therefore quicker to operate on). There will be a slight overhead in the construction of these sets in some cases because of the need to rebalance during additions. On some sets the addition will perform faster because the number of nodes to traverse before the bottom is reached will be significantly reduced due to the balancing.

```
isEmpty :: Set a -> Bool
isEmpty (EmptySet _) = True
isEmpty _ = False

empty :: Set a
empty = (EmptySet B)

singleton :: a -> Set a
singleton a = Node a (EmptySet B) (EmptySet B) B

member :: Ord a => a -> Set a -> Bool
member _ (EmptySet _) = False
member a (Node b _ _ _) | a == b = True
member a (Node b c _ _) | a < b = member a c
member a (Node b _ d _) | a > b = member a d
                        | otherwise = False

size :: Set a -> Int
size (EmptySet _) = 0
size (Node _ b c _) = 1 + size b + size c

isSubsetOf :: Ord a => Set a -> Set a -> Bool
isSubsetOf (EmptySet _) _ = True
isSubsetOf s1 s2 = foldr (\e r -> r && (member e s2)) True s1

union :: Ord a => Set a -> Set a -> Set a
union = foldr add
```

```haskell
intersection :: Ord a => Set a -> Set a -> Set a
intersection s1 =
  foldr (\e r -> addIf (\ el -> member el s1 ) e r) (EmptySet B)

difference :: Ord a => Set a -> Set a -> Set a
difference s1 s2 =
  foldr (\e r -> addIf (\ el -> False == (member el s2)) e r) (EmptySet B) s1

filter :: Ord a => (a -> Bool) -> Set a -> Set a
filter f = foldr (\ e r -> addIf f e r) (EmptySet B)

addIf :: Ord a => (a -> Bool) -> a -> Set a -> Set a
addIf f a r | f a = add a r
                | otherwise = r

map :: Ord b => (a -> b) -> Set a -> Set b
map f = foldr (\ e r -> add (f e) r) (EmptySet B)
```

Valid

`valid` in the red-black tree implementation checks far more than the BST implementation. It first checks that the root of the tree is black before passing the tree to `validStructure`, `wellOrderedNoDup` and `bHeightEqual` which ensure that:

1. There are any duplicates in the set.
2. Every `Node`'s element is greater than those in its left subtree and less than those in its right subtree.
3. Every node is red or black.
4. All leaves are black and contain no data.
5. Every red node has two black children.
6. Every path from a node to a descendant leaf has the same number of black nodes in it.

```haskell
valid :: Ord a => Set a -> Bool
valid (EmptySet B) = True
valid root@(Node _ _ _ B) = validStructure root &&
```

```
                              snd (bHeightEqual root) &&
                              wellOrdNoDup root
valid _ = False

validStructure :: Ord a => Set a -> Bool
validStructure (EmptySet R) = False   -- No red leaves
validStructure (EmptySet BB) = False  -- No double black leaves
validStructure (EmptySet NB) = False  -- No negative black leaves
validStructure (EmptySet B) = True    -- Only black leaves are OK
validStructure (Node _ _ _ BB) = False -- No double black nodes
validStructure (Node _ _ _ NB) = False -- No negative black nodes
validStructure (Node _ l@(Node _ _ _ B) r@(Node _ _ _ B) R) =
  validStructure l && validStructure r -- Red node two black children
validStructure (Node _ l r B) =
  validStructure l && validStructure r -- Black node
validStructure (Node _ (EmptySet B) (EmptySet B) R) =
  True -- Red node with two black emptysets
validStructure (Node _ _ _ R) =
  False -- A red node should only have two black children or empty sets

wellOrdNoDup :: Ord a => Set a -> Bool
wellOrdNoDup (EmptySet _) =
  True -- Empty set is well ordered and cant have duplicates
wellOrdNoDup (Node _ (EmptySet _) (EmptySet _) _) =
  True -- Node with no children
wellOrdNoDup (Node a l@(Node b c d _) (EmptySet _) _) =
    a > b && wellOrdNoDup l && (not (member a c || member a d))
wellOrdNoDup (Node a (EmptySet _) r@(Node b c d _) _) =
    a < b && wellOrdNoDup r && (not (member a c || member a d))
wellOrdNoDup (Node a l@(Node b _ _ _) r@(Node e _ _ _) _) =
    a > b && a < e &&
    wellOrdNoDup l && -- The left child is correctly ordered
    wellOrdNoDup r && -- The right child is correctly ordered
    not (member a l) && -- Element is not contained in left side
    not (member a r) && -- Element is not contained in right side

bHeightEqual :: Set a -> (Int, Bool)
bHeightEqual (EmptySet B) = (1, True)
bHeightEqual (EmptySet BB) =
  error "shouldnt be getting double blacks in settled tree"
bHeightEqual (EmptySet NB) =
```

```
  error "shouldnt be getting negative blacks in settled tree"
bHeightEqual (EmptySet R) =
  error "shouldnt be getting reds in leaf of settled tree"
bHeightEqual (Node _ l r B) =
  ((fst lResult) + 1, (snd lResult) && (snd rResult))
  where
    lResult = bHeightEqual l
    rResult = bHeightEqual r
bHeightEqual (Node _ l r R) =
  ((fst lResult) , (snd lResult) && (snd rResult))
  where
    lResult = bHeightEqual l
    rResult = bHeightEqual r
bHeightEqual _ =
  (0, False) -- NB or BB node, shouldnt be there so no height defined for it
```

Helper Functions

The remaining functions in this module are all helper functions used internally. They facilitate the following actions:

1. `removeMax` removes the rightmost child of a node and calls `bubble` to fix any inconsistencies in the resulting tree.
2. `isBorBB` determines if a node is black or double black, it was created to shorten code where this repeatedly needed to be checked.
3. `toB` and `toR` make a nodes/leaves black/red respectively.
4. `subtractBSet` `addB` and `subtractB` provide colour addition/subtraction for nodes, leaves and colours.
5. `rightChild` is equivalent to getRightChild in the module `Set`.

```
removeMax :: Ord a => Set a -> Set a
removeMax s@(Node _ _ (EmptySet _) _)  = rmove s
removeMax (Node e l r c) = bubble e l (removeMax r) c
removeMax (EmptySet _) = error "Cannot call remove max on an element that doesn’

isBorBB :: Colour -> Bool
isBorBB (B) = True
isBorBB (BB) = True
```

```haskell
isBorBB _ = False

toB :: Set a -> Set a
toB (Node k l r _) = Node k l r B
toB (EmptySet _) = EmptySet B

toR :: Set a -> Set a
toR (Node k l r _) = Node k l r R
toR (EmptySet _) = error "Can't make empty set red"


subtractBSet :: Set a -> Set a
subtractBSet (EmptySet c) = EmptySet (subtractB c)
subtractBSet (Node e l r c) = Node e l r (subtractB c)

{- Defining arithmetic for colours -}
addB :: Colour -> Colour
addB BB = error "Cannot add black to double black"
addB B = BB
addB R = B
addB NB = R

subtractB :: Colour -> Colour
subtractB BB = B
subtractB B = R
subtractB R = NB
subtractB NB = error "Cannot subtract black from negative black"

rChild :: Set a -> a
rChild (Node b _ (EmptySet _) _) = b
rChild (Node _ _ d _) = rChild d
rChild (EmptySet _) = error "Cannot get right child of an empty node"
```