Deletion from Okasaki’s Red-Black Trees: A Functional Pearl

Matt Might
University of Utah
matt.might.net
@mattmight
Red-black delete?
exercise to the reader
functional red black delete
MIT Scheme
delete x _ = empty
delete x (Next u) = rbdelete x u

delete :: Ord a => a -> Tree a -> Tree a
blacken2 (R p) = Next(Base(B p))
blacken2 (C(R(a,x,b))) = Next(Base(B(C a,x,C b)))
blacken2 (C(C t)) = Base t

blacken2 :: RR t a -> RB t a
rbdelete x (Base t) = blacken2 (del x t)
rbdelete x (Next t) = Next (rbdelete x t)

rbdelete :: (Ord a, Deletion t) => a -> RB (AddLayer t) a -> RB t a
instance Deletion t => Deletion (AddLayer t)

d e l f o r m R i g h t ( R  t )  =  R ( a , x , d e l T u p  z  t )
delformLeft(R t) = R(delTup z t,x,b)
where delformLeft(C t) = delLeft z t x b
|  o t h e r w i s e  =  a p p R e d  a  b
|  z > x  =  d e l f o r m R i g h t  b
|  z < x  =  d e l f o r m L e f t  a

d e l T u p  z  ( a , x , b )
instance Deletion t => DelRed (AddLayer t) where
d e l R i g h t  x  a  y  _  =  R ( a , y , C  E )
d e l L e f t  x  _  y  b  =  R ( C  E , y , b )
d e l T u p  z  t @ ( _ , x , _ )  =  i f  x = = z  t h e n  C  E  e l s e  R  t
instance DelRed Unit where
c l a s s (DelRed t, Del t) => Deletion t
d e l : :  O r d  a  = >  a  - >  A d d L a y e r  t  a  - >  R R  t  a
c l a s s Append t => Del t where
d e l R i g h t  : :  O r d  a  = >  a  - >  R e d  t  a  - >  a  - >  t  a  - >  R R  t  a
d e l L e f t  : :  O r d  a  = >  a  - >  t  a  - >  a  - >  R e d  t  a  - >  R R  t  a

c l a s s Append t => DelRed t where
b a l r i g h t B (B y) x t = balance (R y) x t
b a l r i g h t (C t) x b = C (balrightB t x b)
b a l r i g h t (R(a,x,B(b,y,c))) z (C d) = R(balrightB a x (C b),y,C(B(c,z,d)))

b a l l e f t B :: RR t a -> a -> AddLayer t a -> RL t a
b a l l e f t b x (C t) = C (balleftB b x t)
b a l l e f t (R a) y c = R(C(B a),y,c)
b a l l e f t :: RR t a -> a -> RL t a -> RR (AddLayer t) a
threeformR a x (C b) y c = R(R(a,x,b),y,C c)
threeformR a x (R(b,y,c)) z d = R(R(a,x,b),y,R(c,z,d))
ap p R e d (R(a,x,b))(R(c,y,d)) = threeformR a x (app b c) y d
appRed (R(a,x,b)) (C t) = R(C a,x,app b t)
appRed (C t) (R(a,x,b)) = R(app t a,x,C b)
appRed (C x) (C y) = C(app x y)
appRed :: Append t => Red t a -> Red t a -> RR t a
threeformB a x (C b) y c = balleftB (C a) x (B(b,y,c))
threeformB a x (R(b,y,c)) z d = R(B(a,x,b),y,B(c,z,d))
threeformB :: Red t a -> a -> RR t a -> a -> Red t a -> RL t a
instance Append t => Append (AddLayer t) where
instance Append Unit where app _ _ = C E

balance a x (C b) = balanceL a x b
balance (C a) x b = balanceR a x b
balance (R a) y (R b) = R(B a,y,B b)
b a l a n c e :: RR t a -> a -> RR t a -> RL t a

i n s  x  t @ ( R ( a , y , b ) )
|  x > y  =  R ( C  a , y , i n s  x  b )
|  x < y  =  R ( i n s  x  a , y , C  b )
i n s  x  t @ ( B ( l , y , r ) )
|  o t h e r w i s e  =  C  t

b a l a n c e ( C  l )  y  ( i n s  x  r )
b a l a n c e ( i n s  x  l )  y  ( C  r )
b a l a n c e R :: Red t a -> a -> RR t a -> RL t a
b a l a n c e L :: RR t a -> a -> Red t a -> RL t a

b a l a n c eL (C a) x b = C(B(a,x,b))
b a l a n c eL (R(R(a,x,b),y,c)) z d = R(B(C a,x,C b),y,B(c,z,d))

b a l a n c eR a x (C b) = C(B(a,x,b))
b a l a n c eR a x (R(C b,y,C c)) = C(B(a,x,R(b,y,c)))
b a l a n c eR a x (R(b,y,R(c,z,d))) = R(B(a,x,b),y,B(C c,z,C d))
b a l a n c eR a x (R(R(b,y,c),z,d)) = R(B(a,x,C b),y,B(C c,z,d))

b a l a n c eR :: Red t a -> a -> RR t a -> RL t a
b a l a n c eL :: RR t a -> a -> Red t a -> RL t a

b a l a n c eL (R(R(a,x,b),y,c)) z d = R(B(C a,x,C b),y,B(c,z,d))

b a l a n c eR a x (C b) = C(B(a,x,b))

b a l a n c eR a x (R(C b,y,C c)) = C(B(a,x,R(b,y,c)))

b a l a n c eR a x (R(b,y,R(c,z,d))) = R(B(a,x,b),y,B(C c,z,C d))

b a l a n c eR a x (R(R(b,y,c),z,d)) = R(B(a,x,C b),y,B(C c,z,d))

b a l a n c eR :: Red t a -> a -> RR t a -> RL t a

b a l a n c eL :: RR t a -> a -> Red t a -> RL t a

b a l a n c eL (R(R(a,x,b),y,c)) z d = R(B(C a,x,C b),y,B(c,z,d))

b a l a n c eR a x (C b) = C(B(a,x,b))

b a l a n c eR a x (R(C b,y,C c)) = C(B(a,x,R(b,y,c)))

b a l a n c eR a x (R(b,y,R(c,z,d))) = R(B(a,x,b),y,B(C c,z,C d))

b a l a n c eR a x (R(R(b,y,c),z,d)) = R(B(a,x,C b),y,B(C c,z,d))

b a l a n c eR :: Red t a -> a -> RR t a -> RL t a

b a l a n c eL :: RR t a -> a -> Red t a -> RL t a

empty = Base E
empty :: Tree a

type RL t a = Red (AddLayer t) a

show (Next t) = show t
show (Base t) = show t

b m e m x m (B(l,y,r))
b m e m :: Ord a => a -> (t a->Bool) -> AddLayer t a -> Bool

r m e m x m (R(l,y,r))

r m e m :: Ord a => a -> (t a->Bool) -> Red t a->Bool

m e m x t = rbmember x t (bmem x m)
m e m :: Ord a => a -> Tree a -> Bool

t y p e R B t a = RB Unit a

t y p e A d d L a y e r t a = B(Tr(Red t) a)

t y p e R e d t a = C (t a)

d a t a R e d t a = C (t a)
t y p e R e d t a = C (t a)
t y p e R e d t a = R (Tr t a)

d a t a R B t a = Base (t a)
t i m p l e c t S h o w I n s t a n c e a s w e w o r k w i t h 3 r d o r d e r t y p e c o n s t r u c t o r s

d a t a E = E
d a t a B = (t a,a,t a)
d a t a R e d = C (t a)

K a h r s ,  2 0 0 1 )
data Color = R | B deriving Show

data RB a = E | T Color (RB a) a (RB a) deriving Show

{- Insertion and membership test as by Okasaki -}
insert :: Ord a => a -> RB a -> RB a
insert x s =
  where
  T _ a b = ins s
  ins E = T R E x E
  ins s @ ( T B a y b ) =
    |  x < y  = balance (ins a) y b
    |  x > y  = balance a y (ins b)
    | otherwise = s

member :: Ord a => a -> RB a -> Bool
member x E = False
member x (T _ a y b) =
  |  x < y  = member x a
  |  x > y  = member x b
  | otherwise = True

{- balance: first equation is new,
to make it work with a weaker invariant -}
balance :: RB a -> a -> RB a -> RB a
balance (T R a x b) y (T R c z d) = T R (T B a x b) y (T B c z d)
balance (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance a x b = T B a x b

{- deletion a la SMK -}
delete :: Ord a => a -> RB a -> RB a
delete x t =
  case del t of { T _ a y b; _ -> E }
    where
    del E = E
del (T _ a y b) =
    |  x y = deleteLeft a y b
    |  x y = deleteRight a y b
    | otherwise = app a b
deleteLeft a @ ( T B _ _ _ ) y b = balanceLeft (delete a) y b
deleteLeft a y b = T R (delete a) y b
deleteRight a y b = T R a y (delete b)
deleteRight a y b = T B a x b
deleteRight a y b = T B a x b

(- Untyped "Kahrs")

// Based on Stefan Kahrs' Haskell version of Okasaki's RedBlack Trees
// http://www.cse.unsw.edu.au/~dons/data/RedBlackTree.html

def del(k: A): Tree[B] = {
def balance(x: A, xv: B, tl: Tree[B], tr: Tree[B]) = (tl, tr) match {
case (RedTree(y, yv, a, b), RedTree(z, zv, c, d)) => RedTree(x, xv, BlackTree(y, yv, a, b), BlackTree(z, zv, c, d))
case (RedTree(y, yv, RedTree(z, zv, a, b), c), d) => RedTree(y, xv, BlackTree(z, zv, a, b), BlackTree(x, xv, c, d))
case (RedTree(y, yv, a, RedTree(z, zv, b, c)), d) => RedTree(y, xv, BlackTree(z, zv, b, c), BlackTree(x, xv, a, d))
case (RedTree(y, yv, a, b), RedTree(z, zv, c, d)) => RedTree(y, xv, BlackTree(x, xv, a, b), BlackTree(z, zv, c, d))
case (a, RedTree(y, yv, RedTree(z, zv, b, c), d)) => RedTree(y, xv, BlackTree(z, zv, b, c), BlackTree(x, xv, a, d))
case (a, RedTree(y, yv, a, RedTree(z, zv, b, c))) => RedTree(y, xv, BlackTree(z, zv, b, c), BlackTree(x, xv, a, d))
case (a, b) => BlackTree(x, xv, a, b)
}
def subl(t: Tree[B]) = t match {
case BlackTree(x, xv, a, b) => RedTree(x, xv, a, b)
case _ => error("Defect: invariance violation; expected black, got " + t)
}
def balLeft(x: A, xv: B, tl: Tree[B], tr: Tree[B]) = (tl, tr) match {
case (RedTree(y, yv, a, b), c) => RedTree(x, xv, BlackTree(y, yv, a, b), c)
case (bl, BlackTree(y, yv, a, b)) => balance(x, xv, bl, RedTree(y, yv, a, b))
case (bl, RedTree(y, yv, BlackTree(z, zv, a, b), c)) => RedTree(z, xv, BlackTree(x, xv, bl, a), balance(y, yv, b, subl(c)))
case _ => error("Defect: invariance violation at " + right)
}
def balRight(x: A, xv: B, tl: Tree[B], tr: Tree[B]) = (tl, tr) match {
case (a, RedTree(y, yv, b, c)) => RedTree(x, xv, a, BlackTree(y, yv, b, c))
case (BlackTree(y, yv, a, b), bl) => balance(x, xv, RedTree(y, yv, a, b), bl)
case (RedTree(y, yv, a, BlackTree(z, zv, b, c)), bl) => RedTree(x, xv, balance(y, yv, subl(a), b), BlackTree(x, xv, c, bl))
case _ => error("Defect: invariance violation at " + left)
}
def delLeft = left match {
case _ : BlackTree[_] => balLeft(key, value, left.del(k), right)
case _ => RedTree(key, value, left.del(k), right)
}
def delRight = right match {
case _ : BlackTree[_] => balRight(key, value, left, right.del(k))
case _ => RedTree(key, value, left, right.del(k))
}
def append(tl: Tree[B], tr: Tree[B]): Tree[B] = (tl, tr) match {
case (Empty, t) => t
case (t, Empty) => t
case (RedTree(x, xv, a, b), RedTree(y, yv, c, d)) => append(b, c) match {
case RedTree(z, zv, bb, cc) => RedTree(z, xv, RedTree(x, yv, a, b), RedTree(y, cc, d))
case bc => RedTree(x, xv, a, RedTree(y, yv, bc, d))
}
case (BlackTree(x, xv, a, b), BlackTree(y, yv, c, d)) => append(b, c) match {
case RedTree(x, xv, bb, cc) => RedTree(z, zv, RedTree(x, yv, a, b), BlackTree(y, cc, d))
case bc => balleft(x, xv, a, BlackTree(y, yv, bc, d))
}
case (a, RedTree(x, xv, b, c)) => RedTree(x, xv, append(a, b), c)
case (RedTree(x, xv, a, b), c) => RedTree(x, xv, a, append(b, c))
}
// RedBlack is neither A : Ordering[A], nor A <% Ordered[A]
k match {
case _ if isSmaller(k, key) => delLeft
case _ if isSmaller(key, k) => delRight
case _ => append(left, right)
}
let rec min tree =  
  match tree with  
| Node (_, Leaf _, x, _) -> x  
| Node (_, l, _, _) -> min l  
| Leaf _ -> failwith "Impossible"  

let unBB tree =  
  match tree with  
| Leaf BB -> Leaf BB  
| Node (_, l, _, _) -> Node (_, l, _, _)  
| _ -> failwith "Impossible"  

let addB tree =  
  match tree with  
| Node (R, l, x, _) -> Node (B, l, x, _)  
| Node (B, l, x, _) -> Node (BB, l, x, _)  
| Leaf B -> Leaf BB  
| _ -> failwith "Impossible"  

let value tree =  
  match tree with  
| Node (_, _, x, _) -> x  
| Leaf _ -> failwith "Impossible"  

let left tree =  
  match tree with  
| Node (_, _, x, _) -> Node (_, _, x, _)  
| Leaf _ -> failwith "Impossible"  

let right tree =  
  match tree with  
| Node (_, _, _, r) -> r  
| Leaf _ -> failwith "Impossible"  

let isBlack tree =  
  match tree with  
| Leaf B -> true  
| Node (B, _, _, _) -> true  
| _ -> false  

let isRed tree =  
  match tree with  
| Node (R, _, _, _) -> true  
| _ -> false  

let double tree =  
  match tree with  
| Node (BB, _, _, _) -> true  
| Leaf BB -> true  
| _ -> false  

let rec balDelL node =  
  match node with  
  | (B, d, y, Node (R, l, z, r))  
  | (c, d, y, Node (B, l, z, r))  
  | (c, l, x, r)  
  | Node (B, l, x, r)  
  | Node (L, _, x, r)  
  | Leaf _  
  | Leaf _  
  | Leaf _  
  | Leaf _  
  | _ -> node  

let rec balDelR node =  
  match node with  
  | (B, Node (R, l, z, r), y, d)  
  | (c, Node (B, l, z, r), y, d)  
  | (c, l, x, r)  
  | Node (B, l, x, r)  
  | Node (L, _, x, r)  
  | Leaf _  
  | Leaf _  
  | Leaf _  
  | Leaf _  
  | _ -> node  

let rec del (e, t) =  
  let rec aux tree =  
  match tree with  
  | Node (R, Leaf _, x, Leaf _)  
  | Node (B, Leaf _, x, Leaf _)  
  | Node (R, Node (_, l, y, r), x, Leaf _)  
  | Node (B, Node (_, l, y, r), x, Leaf _)  
  | Node (_, Node (B, l, x, r), y, Leaf _)  
  | Node (_, Node (_, l, x, r), y, Leaf _)  
  | (m, a)  
  | (m, b)  
  | (m, c)  
  | (m, d)  
  | (m, e)  
  | Leaf _  
  | Leaf _  
  | Leaf _  
  | Leaf _  
  | _ -> aux tree  

"Untyped" Kahrs / OCaml
local
  datatype zipper
    = TOP
    | LEFT of (color * int * tree * zipper)
    | RIGHT of (color * tree * int * zipper)

in
  fun delete (SET(nItems, t), k) = let
    fun zip (TOP, t) = t
    | zip (LEFT(color, x, b, z), a) = zip(z, T(color, a, x, b))
    | zip (RIGHT(color, a, x, z), b) = zip(z, T(color, a, x, b))
    (* bbzip propagates a black deficit up the tree until either the top
      * is reached, or the deficit can be covered. It returns a boolean
      * that is true if there is still a deficit and the zipped tree.
      *)
    fun bbzip (TOP, t) = (true, t)
    | bbzip (LEFT(B, x, T(R, c, y, d), z), a) = (* case 1L *)
      bbzip (LEFT(R, y, d, z), a)
    | bbzip (LEFT(color, x, T(B, c, y, d), w, e), z), a) = (* case 3L *)
      bbzip (LEFT(color, x, T(B, c, y, d), w, e), z), a)
    | bbzip (LEFT(color, x, T(B, c, y, T(R, d, w, e), x), z), a) = (* case 4L *)
      (false, zip (z, T(color, T(B, a, x, c), y, T(B, d, w, e))))
    | bbzip (LEFT(R, x, T(B, c, y, d), z), a) = (* case 2L *)
      (false, zip (z, T(B, a, x, T(R, c, y, d))))
    | bbzip (LEFT(B, x, T(B, c, y, d), z), a) = (* case 2L *)
      bbzip (z, T(B, a, x, T(R, c, y, d)))
    | bbzip (RIGHT(color, T(R, c, y, d), x, z), b) = (* case 1R *)
      bbzip (RIGHT(R, d, x, RIGHT(B, c, y, z), b)
    | bbzip (RIGHT(color, T(B, T(R, c, w, d), y, e), x, z), b) = (* case 3R *)
      bbzip (RIGHT(color, T(B, T(R, c, w, d), y, e), x, z), b)
    | bbzip (RIGHT(color, T(B, c, y, T(R, d, w, e), y, x), z), b) = (* case 4R *)
      (false, zip (z, T(color, T(B, c, w, t), y, T(B, d, w, e))))
    | bbzip (RIGHT(B, T(B, c, y, d), x, z), b) = (* case 2R *)
      (false, zip (z, T(B, T(R, c, y, d), x, b)))
    | bbzip (RIGHT(B, T(B, c, y, d), x, z), b) = (* case 2R *)
      bbzip (z, T(B, T(R, c, y, d), x, b))
    | bbzip (z, t) = (false, zip(z, t))
  fun delMin (T(R, E, y, b), z) = (y, (false, zip(z, b)))
  | delMin (T(B, E, y, b), z) = (y, z)
  | delMin (T(color, a, y, b), z) = delMin(a, LEFT(color, y, b, z))
  | delMin (E, _) = raise Match
  fun join (R, E, z) = zip(z, E)
  | join (_, a, E, z) = #2(bbzip(z, a))  (* color = black *)
  | join (_, E, b, z) = #2(bbzip(z, b))  (* color = black *)
  | join (color, a, b, z) = let
      val (x, (needB, b')) = delMin(b, TOP)
      in
        if needB
          then #2(bbzip(z, T(color, a, x, b')))  (* color = black *)
        else z
      end
  fun del (E, z) = raise LibBase.NotFound
  | del (T(color, a, y, b), z) = if (k < y)
      then del(a, LEFT(color, y, b, z))
    else if (k = y)
      then join (color, a, b, z)
    else del (b, RIGHT(color, a, y, z))
    in
      SET(nItems-1, del(t, TOP))
  end
end (* local * )
type Elem = Element.T

datatype Color = R | B

datatype Tree = E | T of Color × Tree × Elem × Tree

type Set = Tree

val empty = E

fun member (x, E) = false
    | member (x, T (_, a, y, b)) =
        if Element.lt (x, y) then member (x, a)
        else if Element.lt (y, x) then member (x, b)
        else true

fun insert (x, s) =
    let fun ins E = T (R, E, x, E)
        | ins (s as T (color, a, y, b)) =
            if Element.lt (x, y) then (color, ins a, y, b)
            else if Element.lt (y, x) then (color, a, y, ins b)
            else s
        in T (_, a, y, b) = ins s (* guaranteed to be non-empty *)
    in T (B, a, y, b) end
type Elem = Element.T

datatype Color = R | B

datatype Tree = E | T of Color × Tree × Elem × Tree

type Set = Tree

val empty = E

fun member (x, E) = false
  | member (x, T (_, a, y, b)) =
      if Element.lt (x, y) then member (x, a)
    else if Element.lt (y, x) then member (x, b)
    else true

fun balance (B, T (R, T (R, a, x, b), y, c), z, d) = T (R, T (B, a, x, b), y, T (B, c, z, d))
  | balance (B, T (R, a, x, T (R, b, y, c)), z, d) = T (R, T (B, a, x, b), y, T (B, c, z, d))
  | balance (B, a, x, T (R, T (R, b, y, c), z, d)) = T (R, T (B, a, x, b), y, T (B, c, z, d))
  | balance (B, a, x, T (R, b, y, T (R, c, z, d))) = T (R, T (B, a, x, b), y, T (B, c, z, d))
  | balance body = T body

fun insert (x, s) =
  let
    fun ins E = T (R, E, x, E)
      | ins (s as T (color, a, y, b)) =
          if Element.lt (x, y) then balance (color, ins a, y, b)
        else if Element.lt (y, x) then balance (color, a, y, ins b)
        else s
    val T (_, a, y, b) = ins s (* guaranteed to be non-empty *)
  in
    T (B, a, y, b)
  end
delete x _ = empty

delete:: Ord a => a -> Tree a -> Tree a

blacken2 (C(R(a,x,b))) = Next(Base(B(C a,x,C b)))

blacken2 (C(C t)) = Base t

blacken2 :: RR t a -> RB t a

rbdelete x (Base t) = blacken2 (del x t)

rbdelete x (Next t) = Next (rbdelete x t)

rbdelete :: (Ord a,Deletion t) => a -> RB (AddLayer t) a -> RB t a

instance Deletion Unit

instance Deletion t => Deletion (AddLayer t)

delformRight(C t) = delRight z a x t

where delformLeft(C t) = delLeft z t x b

| otherwise = appRed a b

| z > x  =  delformRight b

| z < x  =  delformLeft a

| z = x  =  Base (C a,x,b)

instance DelRed t => Del t where

| z > x  =  balrightB a x (C b)

| z < x  =  balleftB b x (C a)

instance Deletion t => DelRed (AddLayer t) where

| z > x  =  R (a,y,C(E))

| z < x  =  C(E,b)

instance DelRed Unit where

class (DelRed t, Del t) => Deletion t

del :: Ord a => a -> AddLayer t a -> RR t a

delR :: Ord a => a -> Red t a -> RR t a

delL :: Ord a => a -> Red t a -> RR t a

delT :: Ord a => a -> Tree t a -> RR t a

instance Append t => DelRed t where

| x > y  =  R (a,x,b)

| x < y  =  R (b,y,c)

instance Append Unit where app _ _ = C E

class Append t where app :: t a -> t a -> Red t a

| x > y  =  R (a,x,b)

| x < y  =  R (b,y,c)

instance Insertion t => Insertion (Red t) where

| x > y  =  balance (C l) y (ins x r)

| x < y  =  balance (ins x l) y (C r)

ins x t @ (B l,y,r) = balanceR a x (C b)

balanceR a x (C t) = C (balanceR a x t)

balanceR a x (R (C t,x,b),y,c) = R (C (balanceR a x t),y,balanceR (C b) z c)

balanceR a x (R (a,y,b),z,d) = R (C (balanceR a x t),y,balanceR (C b) z d)

balanceR :: Red t a -> a -> RR t a -> RL t a

balanceL (C t) x (R (a,y,b)) = R (C (balanceL (C b) z c),y,balanceL (C a) x b)

balanceL :: RL t a -> a -> RR t a -> RL t a

threeformR a x (C t) = R (R (a,x,b),y,C c)

threeformR a x (R (b,y,c)) z d = R (R (a,x,b),y,R (c,z,d))

threeformR:: t a -> a -> Red t a -> a -> Red t a -> RR t a

appRed (C t) (R (a,x,b)) = R (app t a,x,C b)

threeformB a x (R (b,y,c)) z d = R (B a,x,b,C d)

threeformB :: Red t a -> a -> RR t a -> a -> Red t a -> RL t a

instance Insertion t where

ins x t @ (B l,y,r) = balance a x (C t)

balance (C a) x b = balanceR a x (C b)

balance (R a) y (R b) = R (B a,y,B b)

balance :: RR t a -> a -> RR t a -> RL t a

| otherwise = C t

| x < y  =  R (ins x a , y , C b)

ins x t @ (R (a,y,b)) = R (ins x a , y , C b)

instance Insertion (Red t) where

ins x t @ (B l,y,r) = balance a x (C t)

balanceL (C t) x (R (a,y,b)) = R (C (balanceL (C b) z c),y,appRed (C a) x b)

balanceL :: RL t a -> a -> RR t a -> RL t a

instance Insertion Unit where ins x E = R(E,x,E)

class Insertion t where ins :: Ord a => a -> t a -> Red t a

insert = rbinsert

insert :: Ord a => a -> Tree a -> Tree a
Easier way?
BST delete + balance' = red-black delete?
Color
Bubble
Balance
Balance
Quiz
Problem: Paths to leaves must have same number of blacks.
Problem: Reds cannot have red children.
Insertion
(define (balance-node node)
  (match node
    [(or (B (R (R a x b) y c) z d)
         (B (R a x (R b y c)) z d)
         (B a x (R (R b y c) z d))
         (B a x (R b y (R c z d))))
     ; =>
     (R (B a x b) y (B c z d))]
    [else node])))
Deletion?
Black

Red
Double black
Black
Red
Negative black
Double black
Black
Red
Negative black
Case 0
Case 1
Case 2
Case I
But, what about?

?
“Bubbling”

```plaintext
y

x
-Black

z
-Black
```

+Black
(define (balance-node node)
  (match node
    [(or (B (R (R a x b) y c) z d))
      (B (R a x (R b y c)) z d))
    [(or (B a x (R (R b y c) z d))
          (B a x (R b y (R c z d))))
      ; =>
      (R (B a x b) y (B c z d))]
    [else node]))
(define (balance-node node)
  (match node
    [(or (B/BB (R (R a x b) y c) z d)
         (B/BB (R a x (R b y c)) z d)
         (B/BB a x (R (R b y c) z d))
         (B/BB a x (R b y (R c z d))))
     ; =>
     (T (black-1 node) (B a x b) y (B c z d))]
    [else node])))
(define (balance node)
  (match node
    [(or (B/BB (R (R a x b) y c) z d)
        (B/BB (R a x (R b y c)) z d)
        (B/BB a x (R (R b y c) z d))
        (B/BB a x (R b y (R c z d))))
      ; =>
      (T (black-1 node) (B a x b) y (B c z d))]
    [else node])))
(define (balance node)
  (match node
    [(or (B/BB (R (R a x b) y c) z d)
        (B/BB (R a x (R b y c)) z d)
        (B/BB a x (R (R b y c) z d))
        (B/BB a x (R b y (R c z d))))
     ; =>
     (T (black-1 node) (B a x b) y (B c z d)))]
    [(BB a x (-B (B b y c) z (and d (B))))
     ; =>
     (B (B a x b) y (balance (B c z (redden d))))])
    [else node])))
(define (balance node)
  (match node
    [(or (B/BB (R (R a x b) y c) z d)
         (B/BB (R a x (R b y c)) z d)
         (B/BB a x (R (R b y c) z d))
         (B/BB a x (R b y (R c z d)))
      ) ; =>
      (T (black-1 node) (B a x b) y (B c z d))] )

    [(BB a x (-B (B b y c) z (and d (B))))
      ; =>
      (B (B a x b) y (balance (B c z (redden d)))))]

    [(BB (-B (and a (B)) x (B b y c)) z d)
      ; =>
      (B (balance (B (redden a) x b)) y (B c z d))] )

  [else node])))
Lesson
+2 colors

BST remove

Bubble & Balance
Questions?

matt.might.net/articles/red-black-delete
@mattmight
(define (sorted-map-delete node key)

(define cmp (sorted-map-compare node))

(define/match (del node)
    [(T! c l k v r)
     (switch-compare (cmp key k)
         [<  (bubble c (del l) k v r)]
         [=  (remove node)]
         [>  (bubble c l k v (del r))]]
    [else     node])

(define/match (remove node)
    [(R (L!)) (L!))  (L cmp)]
    [(B (L!)) (L!))  (BBL cmp)]

    [(or (B (R l k v r) (L!))
          (B (L!)) (R l k v r))
     (T cmp 'B l k v r)]

    [(T! c (and l (T!)) (and r (T!)))]
    (match-let (((cons k v) (sorted-map-max l))
                 (l*         (remove-max l)))
        (bubble c l* k v r))]

    [(T! c l k v r   )  (bubble c l k v (remove-max r))]

    (blacken (del node)))
(define (sorted-map-delete node key)
  (define cmp (sorted-map-compare node))
  (define/match (del node)
    [(T! c l k v r)
      (switch-compare (cmp key k)
        [<   (bubble c (del l) k v r)]
        [=   (remove node)]
        [>   (bubble c l k v (del r))])]
    [else     node])
  (define/match (remove node)
    [(R (L!)) (L!)) (L cmp)]
    [(B (L!)) (L!)) (BBL cmp)]
    [(or (B (R l k v r) (L!))
           (B (L!)) (R l k v r))
      (T cmp 'B l k v r)]
    [(T! c (and l (T!)) (and r (T!)))
      (match-let (((cons k v) (sorted-map-max l))
                    (l*      (remove-max l)))
        (bubble c l* k v r))]
  (define (bubble c l k v r)
    (cond
     [(or (double-black? l) (double-black? r))
      (balance cmp (black+1 c) (black-1 l) k v (black-1 r))]
     [else (T cmp c l k v r)])
  (define/match (remove-max node)
    [(T! l    (L!)) (remove node)]
    [(T! c l k v r ) (bubble c l k v (remove-max r))])
  (blacken (del node)))
Proof