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

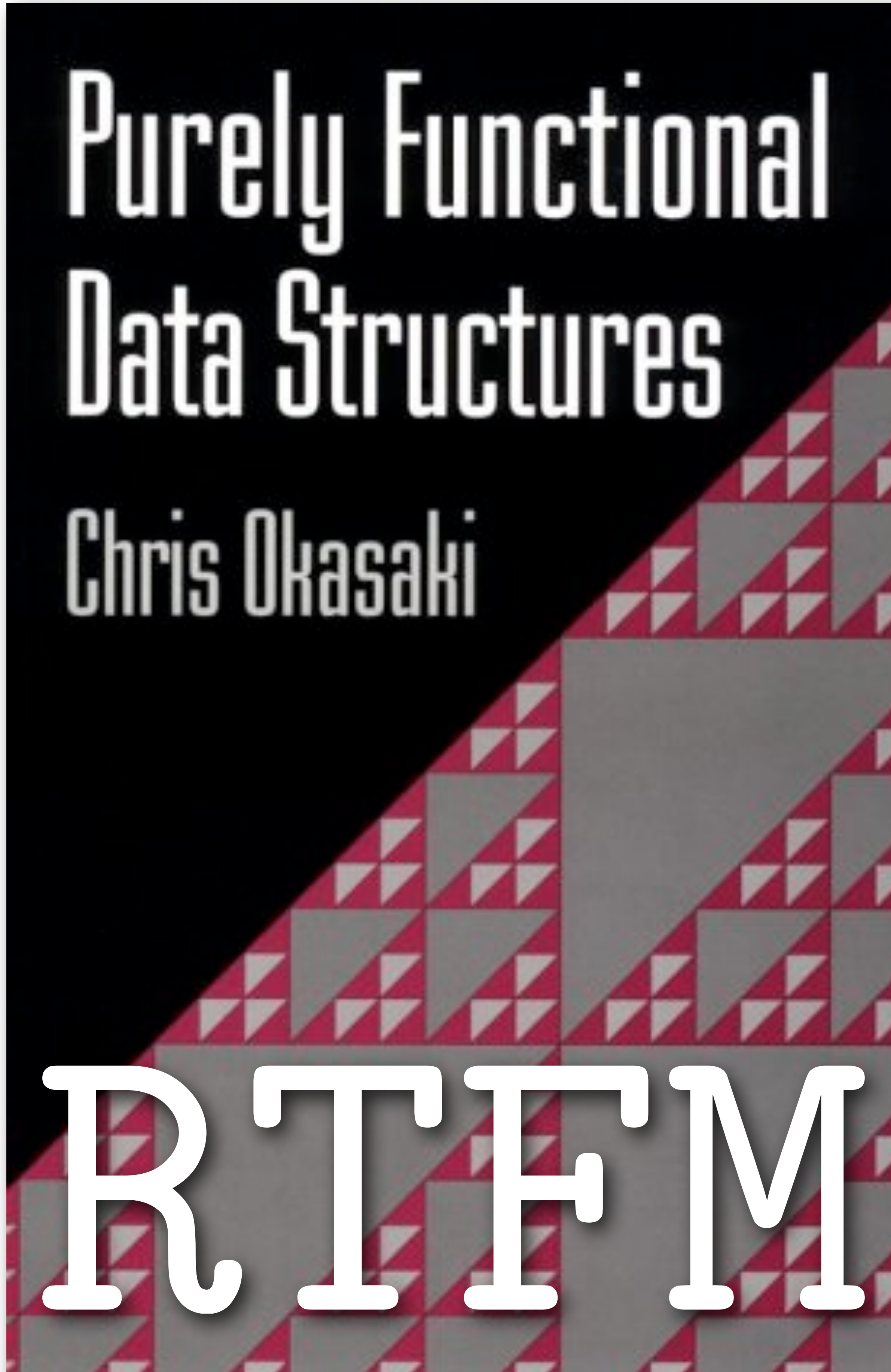
Matt Might

University of Utah

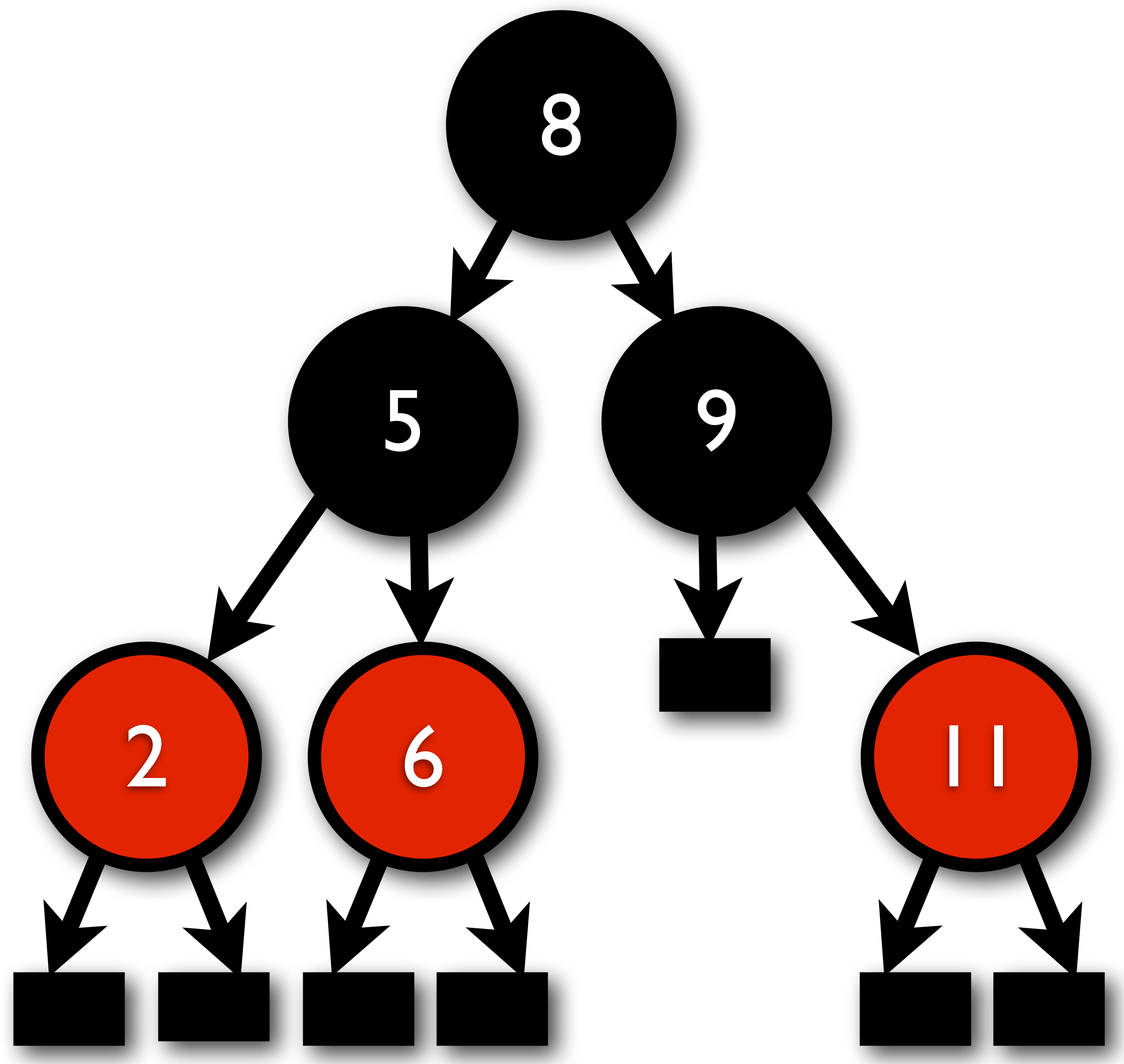
matt.might.net

@mattmight

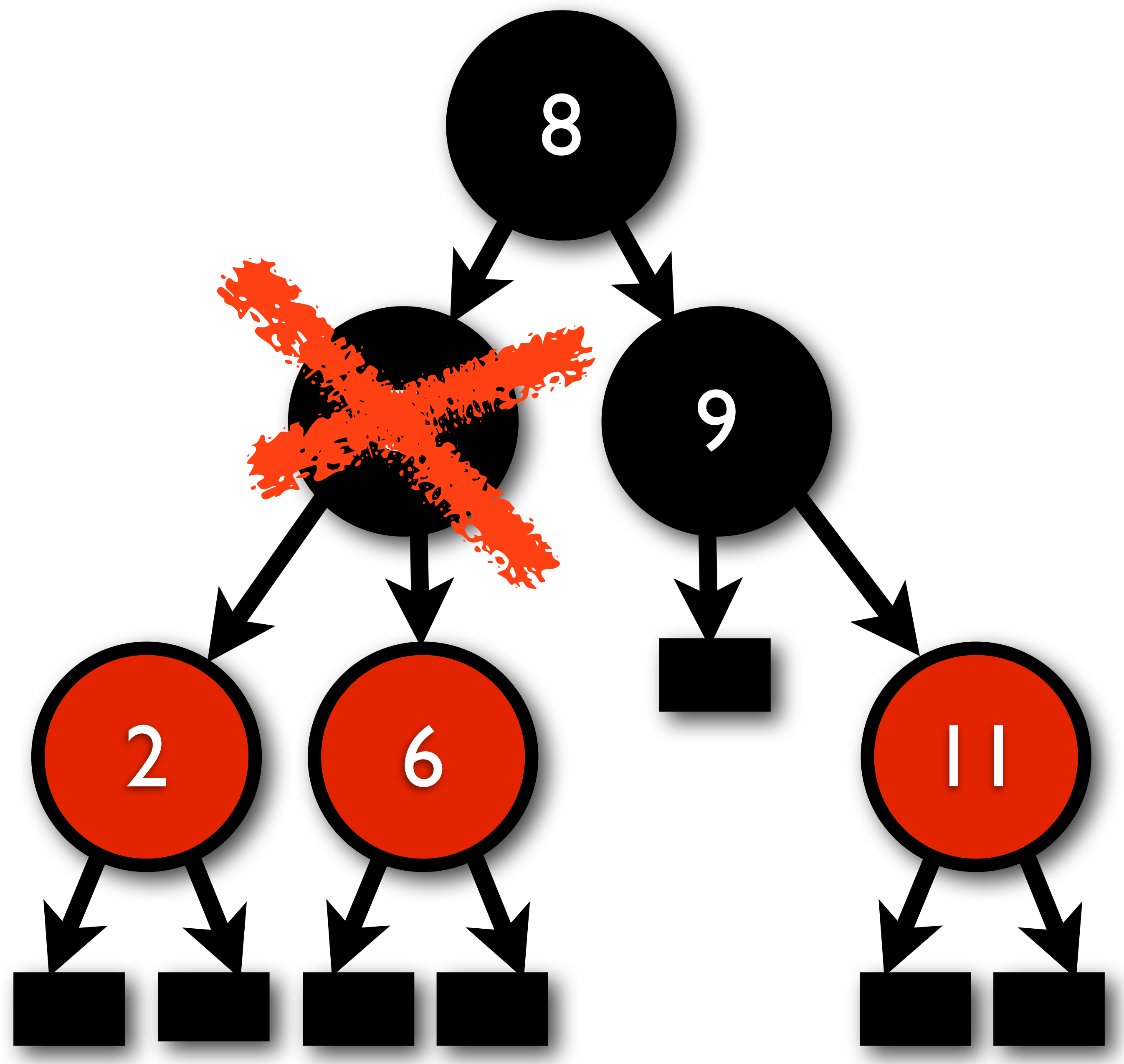

Red-black delete?

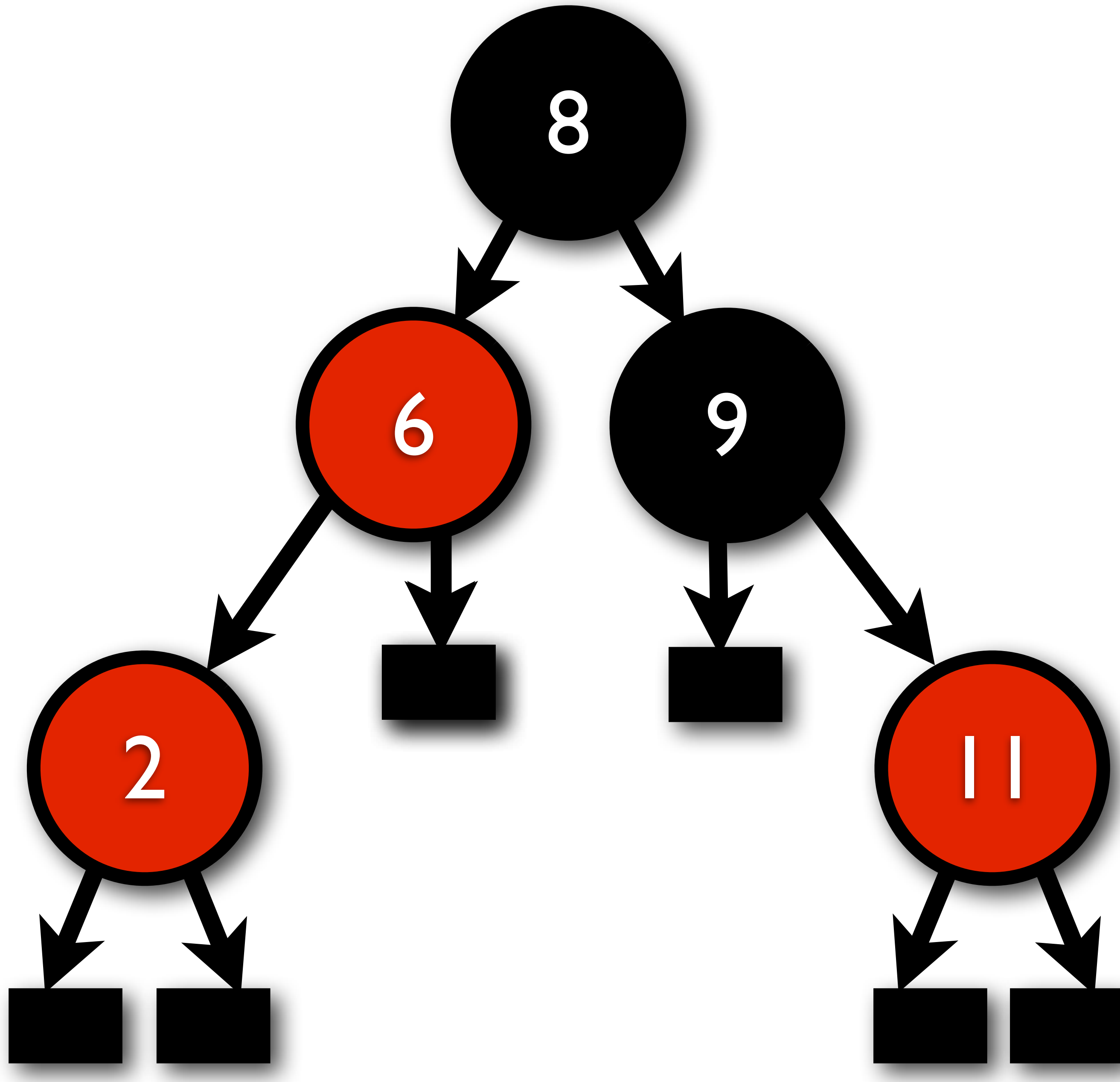

Purely Functional
Data Structures
Chris Okasaki
RTFM

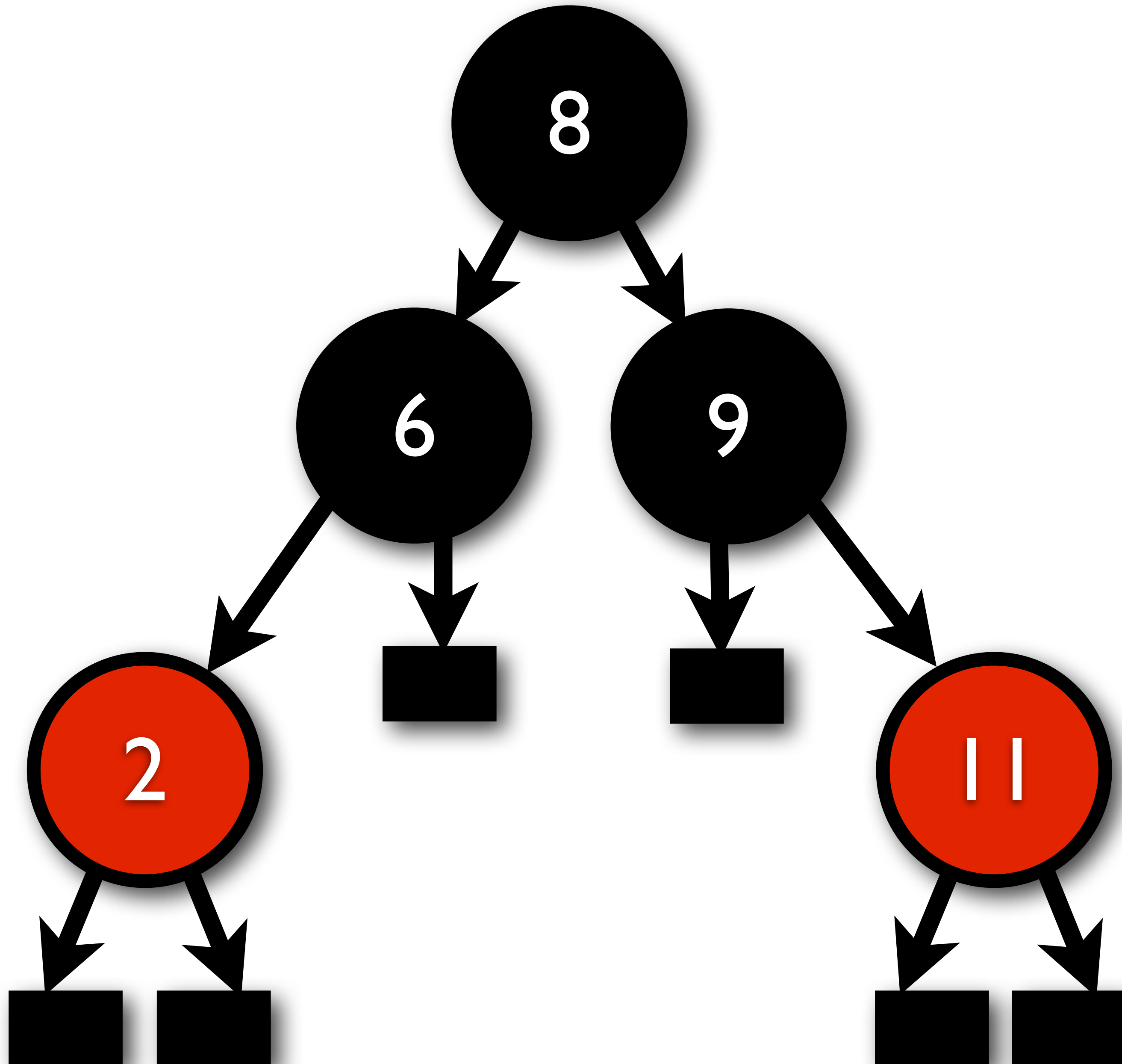
exercise to the reader

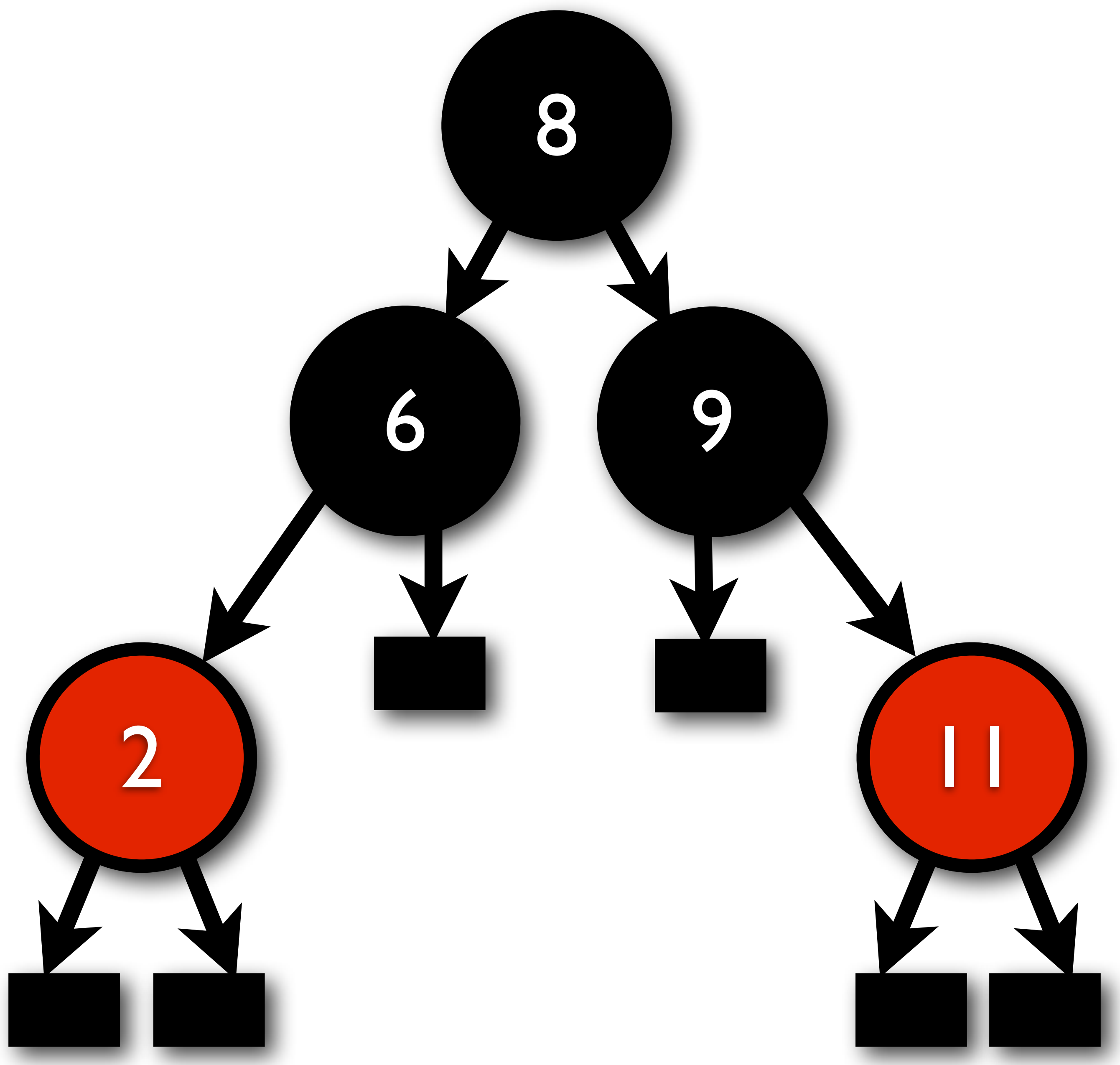


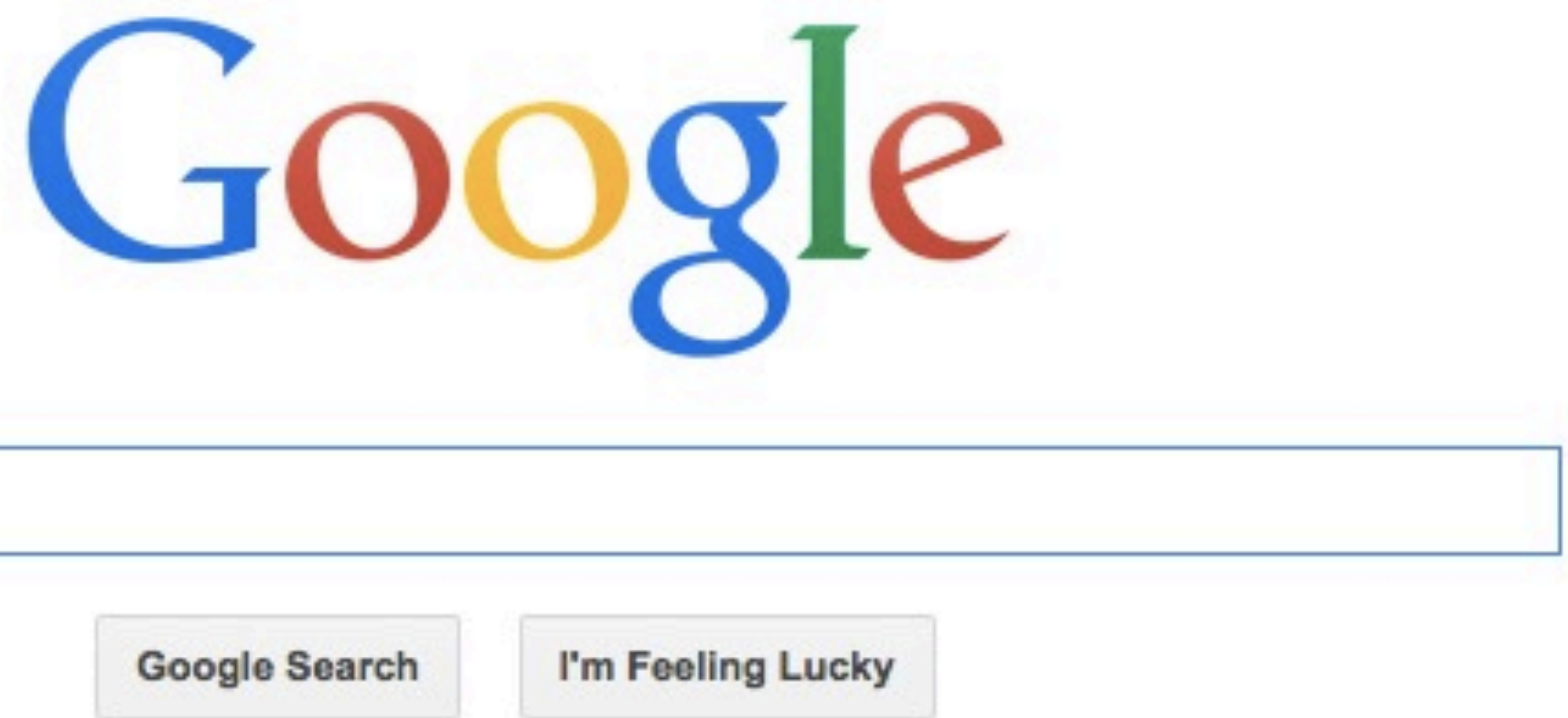
8
5
9
2
6
11

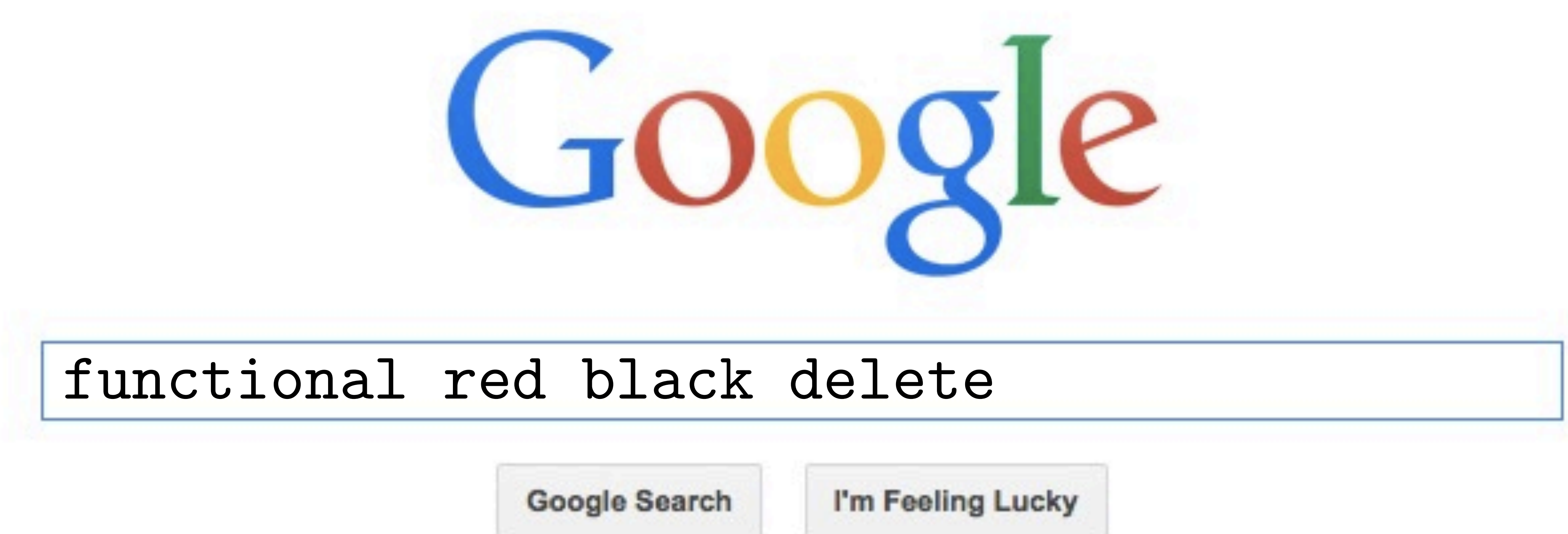

8
9
2
6
11


8
6
9
2
11


8
6
9
2
11


8
6
9
2
11


Google
Google Search
I'm Feeling Lucky


Google
functional red black delete
Google Search
I'm Feeling Lucky

MIT Scheme

!

```racket
(: balance : (All (A) ((RBTree A) -> (RBTree A))))
(define (balance tree)
  (RBTree (RBTree-comparer tree)
          (balance-helper (RBTree-tree tree))))

(: balance-helper : (All (A) ((Tree A) -> (Tree A))))
(define (balance-helper tree)
  (match tree
    [(struct RBNode
       ('black (struct RBNode ('red (struct RBNode ('red a x b)) y c)) z d))
     (RBNode red (RBNode black a x b) y (RBNode black c z d))]
    [(struct RBNode
       ('black (struct RBNode ('red a x (struct RBNode ('red b y c)))) z d))
     (RBNode red (RBNode black a x b) y (RBNode black c z d))]
    [(struct RBNode
       ('black a x (struct RBNode ('red (struct RBNode ('red b y c)) z d))))
     (RBNode red (RBNode black a x b) y (RBNode black c z d))]
    [(struct RBNode
       ('black a x (struct RBNode ('red b y (struct RBNode ('red c z d))))))
     (RBNode red (RBNode black a x b) y (RBNode black c z d))]
    [else tree]))


(: color : (All (A) ((Tree A) -> Color)))
(define (color tree)
  (if (null? tree)
      black
      (RBNode-color tree)))


(: insert : (All (A) (A (RBTree A) -> (RBTree A))))
(define (insert elem tree)
  (let ([func (RBTree-comparer tree)])
    (RBTree func (rake-black (ins elem (RBTree-tree tree) func)))))


(: ins : (All (A) (A (Tree A) (A A -> Boolean) -> (Tree A))))
(define (ins elem tree func)
  (if (null? tree)
      (RBNode red empty elem empty)
      (ins-helper elem tree func)))

(: ins-helper : (All (A) (A (RBNode A) (A A -> Boolean) -> (Tree A))))
(define (ins-helper elem tree func)
  (let* ([nod-elem (RBNode-elem tree)]
         [left-cmp (func elem nod-elem)]
         [right-cmp (func nod-elem elem)]
         [left (RBNode-left tree)]
         [right (RBNode-right tree)]
         [color (RBNode-color tree)])
    (cond
      [(and left-cmp right-cmp) tree]
      [left-cmp
       (balance-helper (RBNode color (ins elem left func) nod-elem right))]
      [else
       (balance-helper (RBNode color left nod-elem (ins elem right func)))])))


(: rake-black : (All (A) ((Tree A) -> (Tree A))))
(define (rake-black tre)
  (if (null? tre) tre
      (RBNode black (RBNode-left tre) (RBNode-elem tre) (RBNode-right tre))))

(: delete-root : (All (A) ((RBTree A) -> (RBTree A))))
(define (delete-root redblacktree)
  (if (empty? redblacktree)
      (error 'delete-root "given tree is empty")
      (delete (root redblacktree) redblacktree)))

(: delete : (All (A) (A (RBTree A) -> (RBTree A))))
(define (delete key redblacktree)
  (let ([func (RBTree-comparer redblacktree)]
        [tree (RBTree-tree redblacktree)])
    (RBTree func (delete-helper key tree func))))

(: delete-helper : (All (A) (A (Tree A) (A A -> Boolean) -> (Tree A))))
(define (delete-helper key tre func)
  (if (null? tre)
      (error 'delete "given key not found in the tree")
      (del-help key tre func)))

(: del-help : (All (A) (A (RBNode A) (A A -> Boolean) -> (Tree A))))
(define (del-help key tre func)
  (: del-left : (Tree A) A (Tree A) -> (Tree A))
  (define (del-left left x right)
    (if (symbol=? (color left) 'black)
        (bal-left (delete-helper key left func) x right)
        (RBNode red (delete-helper key left func) x right)))
  (: del-right : (Tree A) A (Tree A) -> (Tree A))
  (define (del-right left x right)
    (if (symbol=? 'black (color right))
        (bal-right left x (delete-helper key right func))
        (RBNode red left x (delete-helper key right func))))
  (let ([root (RBNode-elem tre)]
        [left (RBNode-left tre)]
        [right (RBNode-right tre)])
    (cond
      [(func key root)
       (if (func root key) (append left right) (del-left left root right))]
      [(func root key) (del-right left root right)]
      [else (append left right)])))

(: rake-red : (All (A) ((Tree A) -> (Tree A))))
(define (rake-red tre)
  (if (null? tre)
      tre
      (RBNode red
              (RBNode-left tre)
              (RBNode-elem tre)
              (RBNode-right tre))))

(: get-left : (All (A) ((Tree A) -> (Tree A))))
(define (get-left tree)
  (if (null? tree)
      (error "Tree empty" 'left)
      (RBNode-left tree)))

(: get-right : (All (A) ((Tree A) -> (Tree A))))
(define (get-right tree)
  (if (null? tree)
      (error "Tree empty" 'right)
      (RBNode-right tree)))

(: bal-left : (All (A) ((Tree A) A (Tree A) -> (Tree A))))
(define (bal-left left x right)
```

Racket

```
        [right (RBNode-right tre)])
    (cond
      [(func key root)
       (if (func root key) (append left right) (del-left left root right))]
      [(func root key) (del-right left root right)]
      [else (append left right)])))

(: rake-red : (All (A) ((Tree A) -> (Tree A))))
(define (rake-red tre)
  (if (null? tre)
      tre
      (RBNode red
              (RBNode-left tre)
              (RBNode-elem tre)
              (RBNode-right tre))))

(: get-left : (All (A) ((Tree A) -> (Tree A))))
(define (get-left tree)
  (if (null? tree)
      (error "Tree empty" 'left)
      (RBNode-left tree)))

(: get-right : (All (A) ((Tree A) -> (Tree A))))
(define (get-right tree)
  (if (null? tree)
      (error "Tree empty" 'right)
      (RBNode-right tree)))

(: bal-left : (All (A) ((Tree A) A (Tree A) -> (Tree A))))
(define (bal-left left x right)
  (cond
    [(symbol=? 'red (color left))
     (RBNode red (rake-black left) x right)]
    [(symbol=? 'black (color right))
     (balance-helper (RBNode black left x (rake-red right)))]
    [(and (symbol=? 'red (color right)) (symbol=? 'black (color (get-left right))))
     (RBNode red (RBNode black left x (get-left (get-left right)))
             (elem (get-left right))
             (balance-helper (RBNode black
                                     (get-right (get-left right))
                                     (elem right)
                                     (sub1 (get-right right)))))]
    [else (RBNode black left x right)]))

(: bal-right : (All (A) ((Tree A) A (Tree A) -> (Tree A))))
(define (bal-right left x right)
  (cond
    [(symbol=? 'red (color right))
     (RBNode red left x (rake-black right))]
    [(symbol=? 'black (color left))
     (balance-helper (RBNode black (rake-red left) x right))]
    [(and (symbol=? 'red (color left))
          (symbol=? 'black (color (get-right left))))
     (RBNode red (balance-helper (RBNode black
                                         (sub1 (get-left left))
                                         (elem left)
                                         (get-left (get-right left))))
             (elem (get-right left))
             (RBNode black (get-right (get-right left)) x right))]
    [else (RBNode black left x right)]))

(: sub1 : (All (A) ((Tree A) -> (Tree A))))
(define (sub1 tree)
  (cond
    [(null? tree) tree]
    [(symbol=? 'black (color tree))
     (RBNode red
             (RBNode-left tree)
             (RBNode-elem tree)
             (RBNode-right tree))]
    [else (error "Invariaance violation" 'sub1)]))

(: append : (All (A) ((Tree A) (Tree A) -> (Tree A))))
(define (append tree1 tree2)
  (let ([t1-color (color tree1)]
        [t2-color (color tree2)])
    (cond
      [(null? tree1) tree2]
      [(null? tree2) tree1]
      [(and (symbol=? 'red t1-color) (symbol=? 'red t2-color)) (appendRR tree1 tree2)]
      [(and (symbol=? 'black t1-color) (symbol=? 'black t2-color)) (appendBB tree1 tree2)]
      [(symbol=? 'red t2-color) (RBNode red (append tree1 (RBNode-left tree2))
                                        (RBNode-elem tree2) (RBNode-right tree2))]
      [else (RBNode red (RBNode-left tree1) (RBNode-elem tree1)
                    (append (RBNode-right tree1) tree2))])))

(: appendRR : (All (A) ((RBNode A) (RBNode A) -> (Tree A))))
(define (appendRR node1 node2)
  (let ([bc (append (RBNode-right node1) (RBNode-left node2))])
    (if (and (RBNode? bc) (symbol=? 'red (color bc)))
        (RBNode red
                (RBNode red
                        (RBNode-left node1)
                        (RBNode-elem node1)
                        (RBNode-left bc))
                (RBNode-elem bc)
                (RBNode red
                        (RBNode-right bc)
                        (RBNode-elem node2)
                        (RBNode-right node2)))
        (RBNode red
                (RBNode-left node1)
                (RBNode-elem node1)
                (RBNode red bc
                        (RBNode-elem node2)
                        (RBNode-right node2))))))

(: appendBB : (All (A) ((RBNode A) (RBNode A) -> (Tree A))))
(define (appendBB node1 node2)
  (let ([bc (append (RBNode-right node1) (RBNode-left node2))])
    (if (and (RBNode? bc) (symbol=? 'red (color bc)))
        (RBNode red
                (RBNode red
                        (RBNode-left node1)
                        (RBNode-elem node1)
                        (RBNode-left bc))
                (RBNode-elem bc)
                (RBNode red
                        (RBNode-right bc)
                        (RBNode-elem node2)
                        (RBNode-right node2)))
        (bal-left (RBNode-left node1)
                  (RBNode-elem node1)
                  (RBNode black bc
                          (RBNode-elem node2)
                          (RBNode-right node2))))))
```

Racket

```haskell
{- Version 2, 1st typed version -}
data Unit a = E deriving Show
type Tr t a = (t a,a,t a)
data Red t a = C (t a) | R (Tr t a)

{- explicit Show instance as we work with 3rd order type constructors -}
instance (Show (t a), Show a) => Show (Red t a)
 where showsPrec _ (C t) = shows t
       showsPrec _ (R(a,b,c)) =
                        ("R("++) . shows a . (","++) . shows b . (","++) . shows c . (")"++)

data AddLayer t a = B(Tr(Red t) a) deriving Show

data RB t a = Base (t a) | Next (RB (AddLayer t) a)

{- this Show instance is not Haskell98, but hugs -98 accepts it -}
instance (Show (t a),Show a) => Show (RB t a)
    where
    show (Base t) = show t
    show (Next t) = show t

type Tree a = RB Unit a
empty :: Tree a
empty = Base E

type RR t a = Red (Red t) a
type RL t a = Red (AddLayer t) a

member :: Ord a => a -> Tree a -> Bool
member x t = rbmember x t (\ _ -> False)

rbmember :: Ord a => a -> RB t a -> (t a->Bool) -> Bool
rbmember x (Base t) m = m t
rbmember x (Next u) m = rbmember x u (bmem x m)

bmem :: Ord a => a -> (t a->Bool) -> AddLayer t a -> Bool
bmem x m (B(l,y,r))
 | x<y = rmem x m l
 | x>y = rmem x m r
 | otherwise = True

rmem :: Ord a => a -> (t a->Bool) -> Red t a->Bool
rmem x m (C t) = m t
rmem x m (R(l,y,r))
 | x<y = m l
 | x>y = m r
 | otherwise = True

insert :: Ord a => a -> Tree a -> Tree a
insert = rbinsert

class Insertion t where ins :: Ord a => a -> t a -> Red t a
instance Insertion Unit where ins x E = R(E,x,E)

rbinsert :: (Ord a,Insertion t) => a -> RB t a -> RB t a
rbinsert x (Next t) = Next (rbinsert x t)
rbinsert x (Base t) = blacken(ins x t)

blacken :: Red t a -> RB t a
blacken (C u) = Base u
blacken (R(a,x,b)) = Next(Base(B(C a,x,C b)))

balanceL :: RR t a -> a -> Red t a -> RL t a
balanceL (R(R(a,x,b),y,c)) z d = R(B(C a,x,C b),y,B(c,z,d))
balanceL (R(a,x,R(b,y,c))) z d = R(B(a,x,C b),y,B(C c,z,d))
balanceL (R(C a,x,C b)) z d = C(B(R(a,x,b),z,d))
balanceL (C a) x b = C(B(a,x,b))

balanceR :: Red t a -> a -> RR t a -> RL t a
balanceR a x (R(R(b,y,c),z,d)) = R(B(a,x,C b),y,B(C c,z,d))
balanceR a x (R(b,y,R(c,z,d))) = R(B(a,x,b),y,B(C c,z,C d))
balanceR a x (R(C b,y,C c)) = C(B(a,x,R(b,y,c)))
balanceR a x (C b) = C(B(a,x,b))

instance Insertion t => Insertion (AddLayer t) where
                    ins x t@(B(l,y,r))
                        | x<y = balance(ins x l) y (C r)
                        | x>y = balance(C l) y (ins x r)
                        | otherwise = C t
instance Insertion t => Insertion (Red t) where
                    ins x (C t) = C(ins x t)
                    ins x t@(R(a,y,b))
                        | x<y = R(ins x a,y,C b)
                        | x>y = R(C a,y,ins x b)
                        | otherwise = C t

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

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

instance Append Unit where app _ _ = C E

instance Append t => Append (AddLayer t) where
    app (B(a,x,b)) (B(c,y,d)) = threeformB a x (appRed b c) y d

threeformB :: Red t a -> a -> RR t a -> a -> Red t a -> RL t a
threeformB a x (R(b,y,c)) z d = R(B(a,x,b),y,B(c,z,d))
threeformB a x (C b) y c = balleftB (C a) x (B(b,y,c))

appRed :: Append t => Red t a -> Red t a -> RR t a
appRed (C x) (C y) = C(app x y)
appRed (C t) (R(a,x,b)) = R(app t a,x,C b)
appRed (R(a,x,b)) (C t) = R(C a,x,app b t)
appRed (R(a,x,b))(R(c,y,d)) = threeformR a x (app b c) y d

threeformR:: t a -> a -> Red t a -> a -> t a -> RR t a
threeformR a x (R(b,y,c)) z d = R(R(a,x,b),y,R(c,z,d))
threeformR a x (C b) y c = R(R(a,x,b),y,C c)

balleft :: RR t a -> a -> RL t a -> RR (AddLayer t) a
balleft (R a) y c = R(C(B a),y,c)
balleft (C t) x (R(B(a,y,b),z,c)) = R(C(B(t,x,a)),y,balleftB (C b) z c)
balleft b x (C t) = C (balleftB b x t)

balleftB :: RR t a -> a -> AddLayer t a -> RL t a
balleftB bl x (B y) = balance bl x (R y)

balright :: RL t a -> a -> RR t a -> RR (AddLayer t) a
balright a x (R b) = R(a,x,C(B b))
balright (R(a,x,B(b,y,c))) z (C d) = R(balrightB a x (C b),y,C(B(c,z,d)))
balright (C t) x b = C (balrightB t x b)

balrightB :: AddLayer t a -> a -> RR t a -> RL t a
balrightB (B y) x t = balance (R y) x t

class Append t => DelRed t where
                    delTup :: Ord a => a -> Tr t a -> Red t a
                    delLeft :: Ord a => a -> t a -> a -> Red t a -> RR t a
                    delRight :: Ord a => a -> Red t a -> a -> t a -> RR t a

class Append t => Del t where
                    del :: Ord a => a -> AddLayer t a -> RR t a

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

instance DelRed Unit where
                    delTup z t@(_,x,_) = if x==z then C E else R t
                    delLeft x _ y b = R(C E,y,b)
                    delRight x a y _ = R(a,y,C E)

instance Deletion t => DelRed (AddLayer t) where
                    delTup z (a,x,b)
                                  | z<x = balleftB (del z a) x b
                                  | z>x = balrightB a x (del z b)
                                  | otherwise = app a b
                    delLeft x a y b = balleft (del x a) y b
                    delRight x a y b = balright a y (del x b)

instance DelRed t => Del t where
                    del z (B(a,x,b))
                        | z<x = delformLeft a
                        | z>x = delformRight b
                        | otherwise = appRed a b
              where delformLeft(C t) = delLeft z t x b
                    delformLeft(R t) = R(delTup z t,x,b)
                    delformRight(C t) = delRight z a x t
                                        delformRight(R t) = R(a,x,delTup z t)

instance Deletion t => Deletion (AddLayer t)
instance Deletion Unit

rbdelete :: (Ord a,Deletion t) => a -> RB (AddLayer t) a -> RB t a
rbdelete x (Next t) = Next (rbdelete x t)
rbdelete x (Base t) = blacken2 (del x t)

blacken2 :: RR t a -> RB t a
blacken2 (C(C t)) = Base t
blacken2 (C(R(a,x,b))) = Next(Base(B(C a,x,C b)))
blacken2 (R p) = Next(Base(B p))

delete:: Ord a => a -> Tree a -> Tree a
delete x (Next u) = rbdelete x u
delete x _ = empty
```

(Kahrs, 2001)

```haskell
{- Version 1, 'untyped' -}
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 =
        T B a z b
        where
        T _ a z 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
        ins s@(T R a y b)
                | x<y = T R (ins a) y b
                | x>y = T R 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 (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
balance 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 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 -> T B a y b; _ -> E}
        where
        del E = E
        del (T _ a y b)
            | x<y = delformLeft a y b
            | x>y = delformRight a y b
            | otherwise = app a b
        delformLeft a@(T B _ _ _) y b = balleft (del a) y b
        delformLeft a y b = T R (del a) y b
        delformRight a y b@(T B _ _ _) = balright a y (del b)
        delformRight a y b = T R a y (del b)

balleft :: RB a -> a -> RB a -> RB a
balleft (T R a x b) y c = T R (T B a x b) y c
balleft bl x (T B a y b) = balance bl x (T R a y b)
balleft bl x (T R (T B a y b) z c) = T R (T B bl x a) y (balance b z (sub1 c))

balright :: RB a -> a -> RB a -> RB a
balright a x (T R b y c) = T R a x (T B b y c)
balright (T B a x b) y bl = balance (T R a x b) y bl
balright (T R a x (T B b y c)) z bl = T R (balance (sub1 a) x b) y (T B c z bl)

sub1 :: RB a -> RB a
sub1 (T B a x b) = T R a x b
sub1 _ = error "invariance violation"

app :: RB a -> RB a -> RB a
app E x = x
app x E = x
app (T R a x b) (T R c y d) =
        case app b c of
            T R b' z c' -> T R(T R a x b') z (T R c' y d)
            bc -> T R a x (T R bc y d)
app (T B a x b) (T B c y d) =
        case app b c of
            T R b' z c' -> T R(T B a x b') z (T B c' y d)
            bc -> balleft a x (T B bc y d)
app a (T R b x c) = T R (app a b) x c
app (T R a x b) c = T R a x (app b c)
```

(“Untyped” Kahrs)

```
// Based on Stefan Kahrs' Haskell version of Okasaki's Red&Black 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, yv, BlackTree(z, zv, a, b), BlackTree(x, xv, c, d))
    case (RedTree(y, yv, a, RedTree(z, zv, b, c)), d) =>
      RedTree(z, zv, BlackTree(y, yv, a, b), BlackTree(x, xv, c, d))
    case (a, RedTree(y, yv, b, RedTree(z, zv, c, d))) =>
      RedTree(y, yv, BlackTree(x, xv, a, b), BlackTree(z, zv, c, d))
    case (a, RedTree(y, yv, RedTree(z, zv, b, c), d)) =>
      RedTree(z, zv, BlackTree(x, xv, a, b), BlackTree(y, yv, c, 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, zv, 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(z, zv, 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, zv, RedTree(x, xv, a, bb), RedTree(y, yv, 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(z, zv, bb, cc) => RedTree(z, zv, BlackTree(x, xv, a, bb), BlackTree(y, yv, 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)
  }
}
```

(“Untyped” Kahrs / Scala)

```ocaml
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 B
    | Node (BB, l, x, r) -> Node (B, l, x, r)
    | _ -> failwith "Impossible"

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

let rigth 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)) ->
            if double d
            then Node (B, balDelL (R, d, y, l), z, r)
            else Node (B, d, y, Node (R, l, z, r))
    | (c, d, y, Node (B, l, z, r)) ->
            if double d
            then
              if isBlack l && isBlack r
              then addB (Node (c, unBB d, y, Node (R, l, z, r)))
              else if isRed l && isBlack r
              then balDelL (c, d, y, Node (B, left l, value l, Node (R, rigth l, z, r)))
              else Node (c, Node (B, unBB d, y, l), z, addB r)
            else Node (c, d, y, Node (B, l, z, r))
    | (c, l, x, r) -> Node (c, l, x, r)

let rec balDelR node =
  match node with
          (B, Node (R, l, z, r), y, d) ->
            if double d
            then Node (B, l, z, balDelR (R, r, y, d))
            else Node (B, Node (R, l, z, r), y, d)
    | (c, Node (B, l, z, r), y, d) ->
            if double d
            then
              if isBlack l && isBlack r
              then addB (Node (c, Node (R, l, z, r), y, unBB d))
              else if isBlack l && isRed r
              then balDelR (c, Node (B, Node (R, l, z, left r), value r, rigth r), y, d)
              else Node (c, addB l, z, Node (B, r, y, unBB d))
            else Node (c, Node (B, l, z, r), y, d)
    | (c, l, x, r) -> Node (c, l, x, r)

let rec del (e, t) =
  let rec aux tree =
    match tree with
            Node (R, Leaf _, x, Leaf _) ->
              if El.comp (e, x) = Eq then Leaf B else tree
          | Node (B, Leaf _, x, Leaf _) ->
              if El.comp (e, x) = Eq then Leaf BB else tree
          | Node (_, Leaf _, x, Node (_, l, y, r)) ->
              if El.comp (e, x) = Eq
              then Node (B, l, y, r)
              else if El.comp (e, y) = Eq
              then Node (B, Leaf B, x, Leaf B)
              else tree
          | Node (_, Node (_, l, y, r), x, Leaf _) ->
              if El.comp (e, x) = Eq
              then Node (B, l, y, r)
              else if El.comp (e, y) = Eq
              then Node (B, Leaf B, x, Leaf B)
              else tree
          | Node (c, l, x, r) ->
              (match El.comp (e, x) with
                       Lt -> balDelL (c, aux l, x, r)
                 | Gt -> balDelR (c, l, x, aux r)
                 | Eq ->
                         let m = min r
                         in balDelR (c, l, m, del (m, r)))
          | Leaf _ -> tree
  in
    aux t
```

("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, x, c, LEFT(B, y, d, z)), a)
        | bbZip (LEFT(color, x, T(B, T(R, c, y, d), w, e), z), a) = (* case 3L *)
            bbZip (LEFT(color, x, T(B, c, y, T(R, d, w, e)), z), a)
        | bbZip (LEFT(color, x, T(B, c, y, T(R, d, w, e)), 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, c, w, T(R, d, y, e)), x, z), b)
        | bbZip (RIGHT(color, T(B, c, y, T(R, d, w, e)), x, z), b) = (* case 4R *)
            (false, zip (z, T(color, c, y, T(B, T(R, d, w, e), x, b))))
        | bbZip (RIGHT(R, 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, bbZip(z, b))
        | delMin (T(color, a, y, b), z) = delMin(a, LEFT(color, y, b, z))
        | delMin (E, _) = raise Match
      fun join (R, E, 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')))
                else zip(z, T(color, a, x, b'))
            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 *)
```

(Reppy, SML/NJ)

```
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
          val 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
```

```haskell
{- Version 2, 1st typed version -}
data Unit a = E deriving Show
type Tr t a = (t a,a,t a)
data Red t a = C (t a) | R (Tr t a)

{- explicit Show instance as we work with 3rd order type constructors -}
instance (Show (t a), Show a) => Show (Red t a)
 where showsPrec _ (C t) = shows t
       showsPrec _ (R(a,b,c)) =
                        ("R("++) . shows a . (","++) . shows b . (","++) . shows c . (")"++)

data AddLayer t a = B(Tr(Red t) a) deriving Show

data RB t a = Base (t a) | Next (RB (AddLayer t) a)

{- this Show instance is not Haskell98, but hugs -98 accepts it -}
instance (Show (t a),Show a) => Show (RB t a)
    where
    show (Base t) = show t
    show (Next t) = show t

type Tree a = RB Unit a
empty :: Tree a
empty = Base E

type RR t a = Red (Red t) a
type RL t a = Red (AddLayer t) a

member :: Ord a => a -> Tree a -> Bool
member x t = rbmember x t (\ _ -> False)

rbmember :: Ord a => a -> RB t a -> (t a->Bool) -> Bool
rbmember x (Base t) m = m t
rbmember x (Next u) m = rbmember x u (bmem x m)

bmem :: Ord a => a -> (t a->Bool) -> AddLayer t a -> Bool
bmem x m (B(l,y,r))
 | x<y = rmem x m l
 | x>y = rmem x m r
 | otherwise = True

rmem :: Ord a => a -> (t a->Bool) -> Red t a->Bool
rmem x m (C t) = m t
rmem x m (R(l,y,r))
 | x<y = m l
 | x>y = m r
 | otherwise = True

insert :: Ord a => a -> Tree a -> Tree a
insert = rbinsert

class Insertion t where ins :: Ord a => a -> t a -> Red t a
instance Insertion Unit where ins x E = R(E,x,E)

rbinsert :: (Ord a,Insertion t) => a -> RB t a -> RB t a
rbinsert x (Next t) = Next (rbinsert x t)
rbinsert x (Base t) = blacken(ins x t)

blacken :: Red t a -> RB t a
blacken (C u) = Base u
blacken (R(a,x,b)) = Next(Base(B(C a,x,C b)))

balanceL :: RR t a -> a -> Red t a -> RL t a
balanceL (R(R(a,x,b),y,c)) z d = R(B(C a,x,C b),y,B(c,z,d))
balanceL (R(a,x,R(b,y,c))) z d = R(B(a,x,C b),y,B(C c,z,d))
balanceL (R(C a,x,C b)) z d = C(B(R(a,x,b),z,d))
balanceL (C a) x b = C(B(a,x,b))

balanceR :: Red t a -> a -> RR t a -> RL t a
balanceR a x (R(R(b,y,c),z,d)) = R(B(a,x,C b),y,B(C c,z,d))
balanceR a x (R(b,y,R(c,z,d))) = R(B(a,x,b),y,B(C c,z,C d))
balanceR a x (R(C b,y,C c)) = C(B(a,x,R(b,y,c)))
balanceR a x (C b) = C(B(a,x,b))

instance Insertion t => Insertion (AddLayer t) where
                    ins x t@(B(l,y,r))
                        | x<y = balance(ins x l) y (C r)
                        | x>y = balance(C l) y (ins x r)
                        | otherwise = C t
instance Insertion t => Insertion (Red t) where
                    ins x (C t) = C(ins x t)
                    ins x t@(R(a,y,b))
                        | x<y = R(ins x a,y,C b)
                        | x>y = R(C a,y,ins x b)
                        | otherwise = C t

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

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

instance Append Unit where app _ _ = C E

instance Append t => Append (AddLayer t) where
    app (B(a,x,b)) (B(c,y,d)) = threeformB a x (appRed b c) y d

threeformB :: Red t a -> a -> RR t a -> a -> Red t a -> RL t a
threeformB a x (R(b,y,c)) z d = R(B(a,x,b),y,B(c,z,d))
threeformB a x (C b) y c = balleftB (C a) x (B(b,y,c))

appRed :: Append t => Red t a -> Red t a -> RR t a
appRed (C x) (C y) = C(app x y)
appRed (C t) (R(a,x,b)) = R(app t a,x,C b)
appRed (R(a,x,b)) (C t) = R(C a,x,app b t)
appRed (R(a,x,b))(R(c,y,d)) = threeformR a x (app b c) y d

threeformR:: t a -> a -> Red t a -> a -> t a -> RR t a
threeformR a x (R(b,y,c)) z d = R(R(a,x,b),y,R(c,z,d))
threeformR a x (C b) y c = R(R(a,x,b),y,C c)

balleft :: RR t a -> a -> RL t a -> RR (AddLayer t) a
balleft (R a) y c = R(C(B a),y,c)
balleft (C t) x (R(B(a,y,b),z,c)) = R(C(B(t,x,a)),y,balleftB (C b) z c)
balleft b x (C t) = C (balleftB b x t)

balleftB :: RR t a -> a -> AddLayer t a -> RL t a
balleftB bl x (B y) = balance bl x (R y)

balright :: RL t a -> a -> RR t a -> RR (AddLayer t) a
balright a x (R b) = R(a,x,C(B b))
balright (R(a,x,B(b,y,c))) z (C d) = R(balrightB a x (C b),y,C(B(c,z,d)))
balright (C t) x b = C (balrightB t x b)

balrightB :: AddLayer t a -> a -> RR t a -> RL t a
balrightB (B y) x t = balance (R y) x t

class Append t => DelRed t where
                    delTup :: Ord a => a -> Tr t a -> Red t a
                    delLeft :: Ord a => a -> t a -> a -> Red t a -> RR t a
                    delRight :: Ord a => a -> Red t a -> a -> t a -> RR t a

class Append t => Del t where
                    del :: Ord a => a -> AddLayer t a -> RR t a

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

instance DelRed Unit where
                    delTup z t@(_,x,_) = if x==z then C E else R t
                    delLeft x _ y b = R(C E,y,b)
                    delRight x a y _ = R(a,y,C E)

instance Deletion t => DelRed (AddLayer t) where
                    delTup z (a,x,b)
                                        | z<x = balleftB (del z a) x b
                                        | z>x = balrightB a x (del z b)
                                        | otherwise = app a b
                    delLeft x a y b = balleft (del x a) y b
                    delRight x a y b = balright a y (del x b)

instance DelRed t => Del t where
                    del z (B(a,x,b))
                        | z<x = delformLeft a
                        | z>x = delformRight b
                        | otherwise = appRed a b
              where delformLeft(C t) = delLeft z t x b
                    delformLeft(R t) = R(delTup z t,x,b)
                    delformRight(C t) = delRight z a x t
                                        delformRight(R t) = R(a,x,delTup z t)

instance Deletion t => Deletion (AddLayer t)
instance Deletion Unit

rbdelete :: (Ord a,Deletion t) => a -> RB (AddLayer t) a -> RB t a
rbdelete x (Next t) = Next (rbdelete x t)
rbdelete x (Base t) = blacken2 (del x t)

blacken2 :: RR t a -> RB t a
blacken2 (C(C t)) = Base t
blacken2 (C(R(a,x,b))) = Next(Base(B(C a,x,C b)))
blacken2 (R p) = Next(Base(B p))

delete:: Ord a => a -> Tree a -> Tree a
delete x (Next u) = rbdelete x u
delete x _ = empty
```

(Kahrs, 2001)

Easier way?

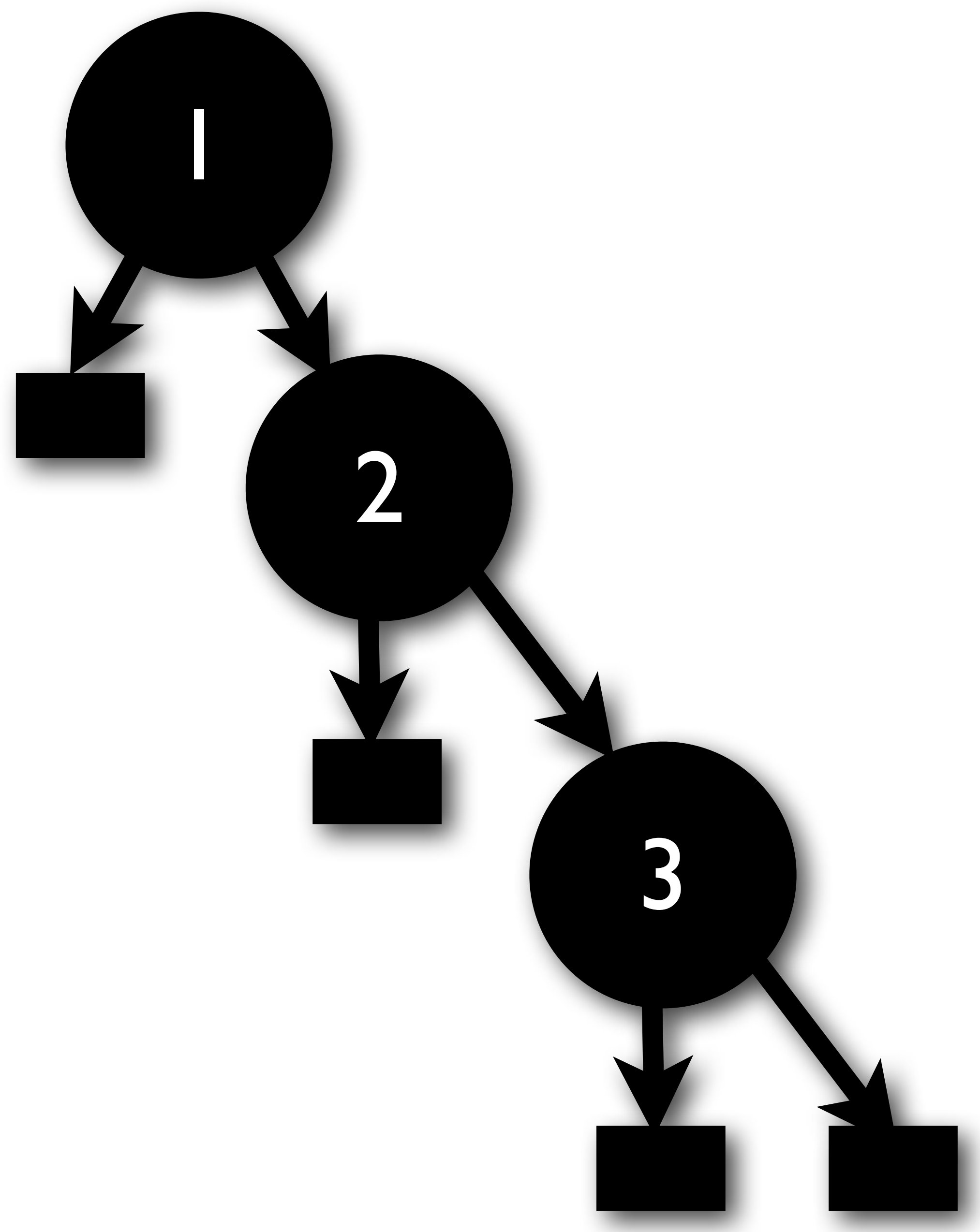
BST delete + balance' = red-black delete?

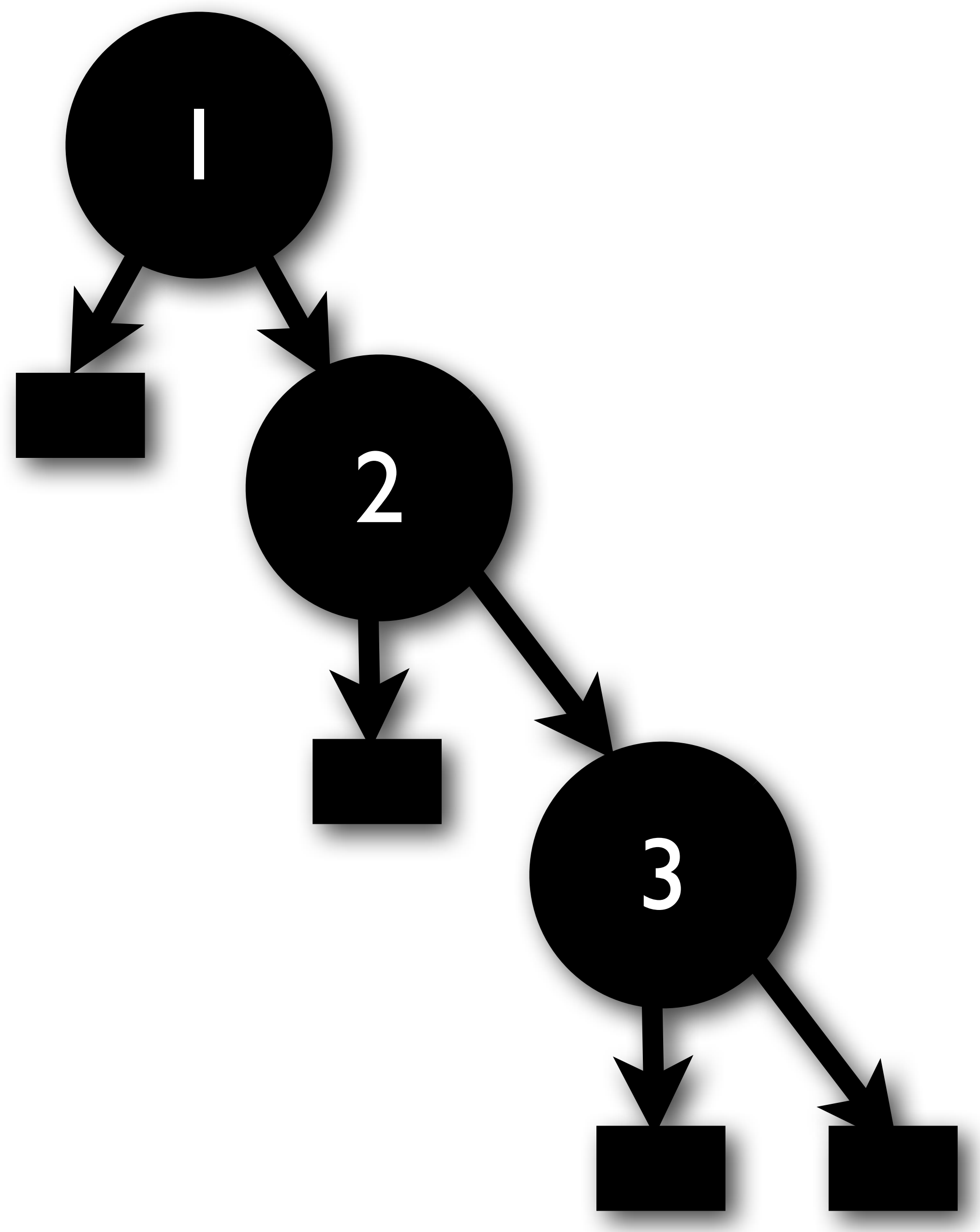
Color

Bubble

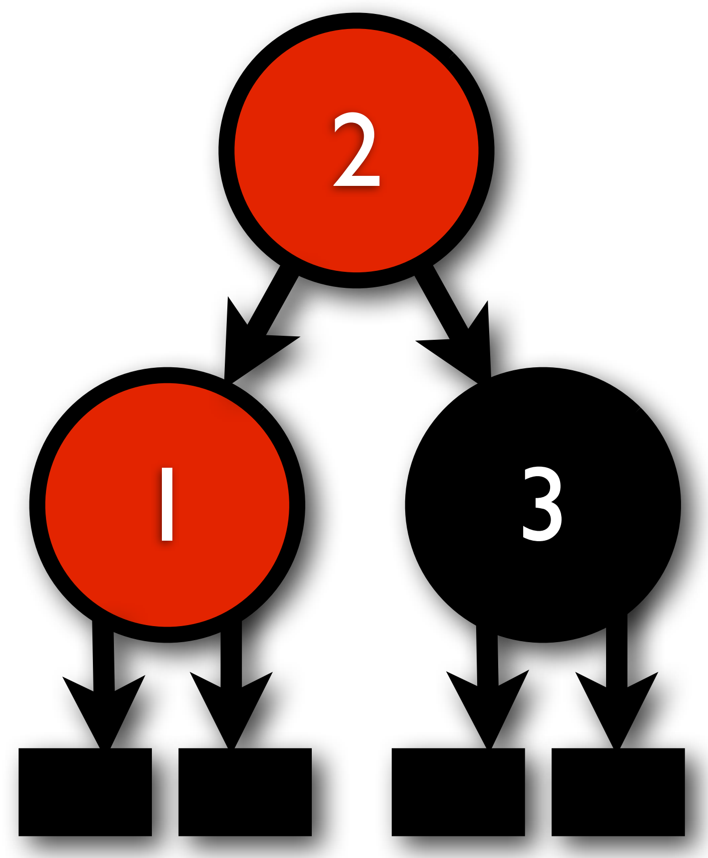
Balance

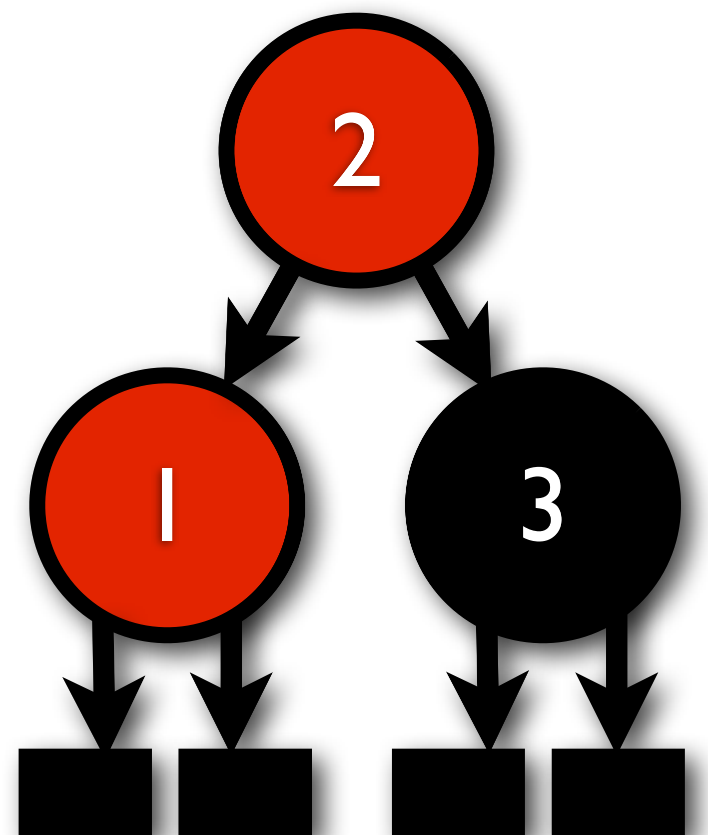
Quiz


1
2
3

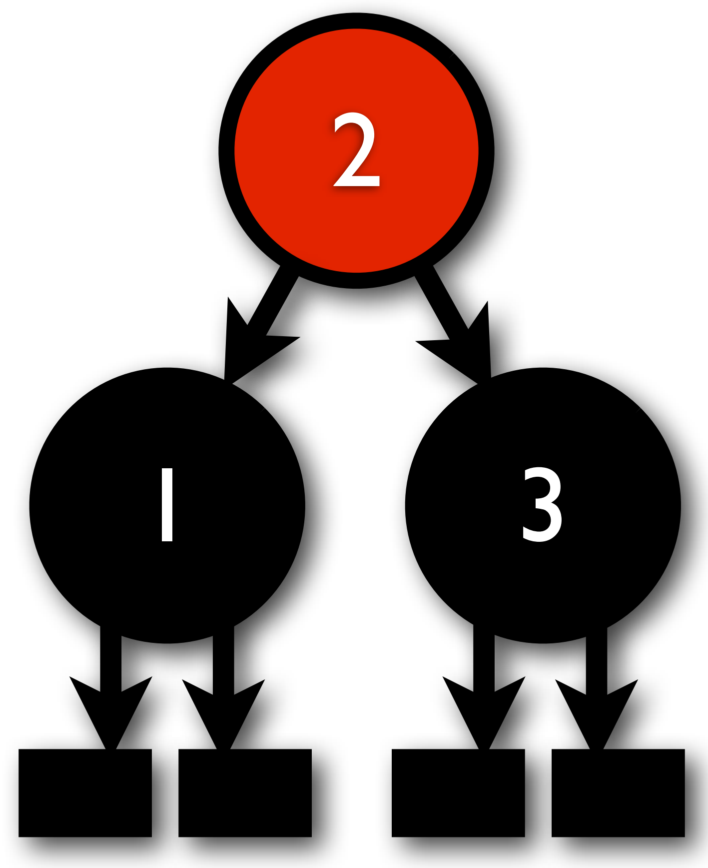


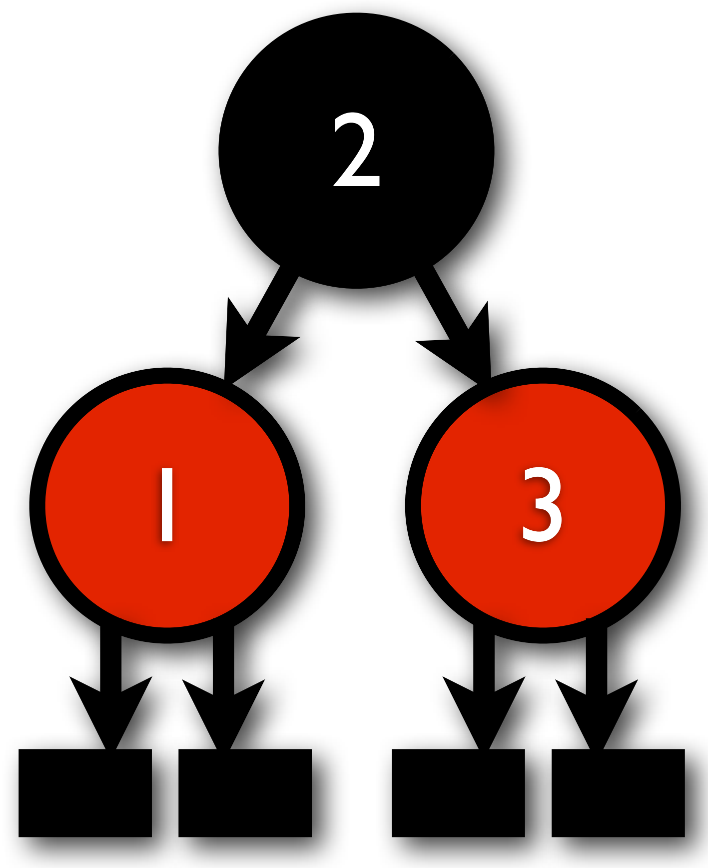
Problem: Paths to leaves must have same number of blacks.

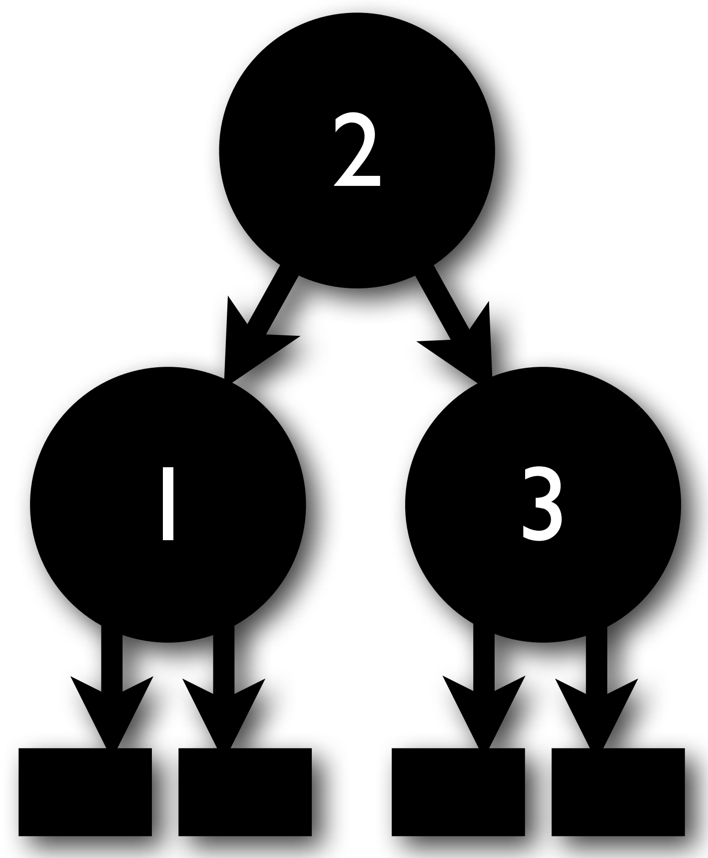

2
1
3



Problem: Reds cannot have red children.


2
1
3

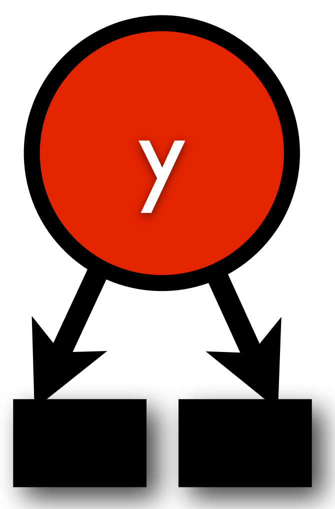

2
1
3

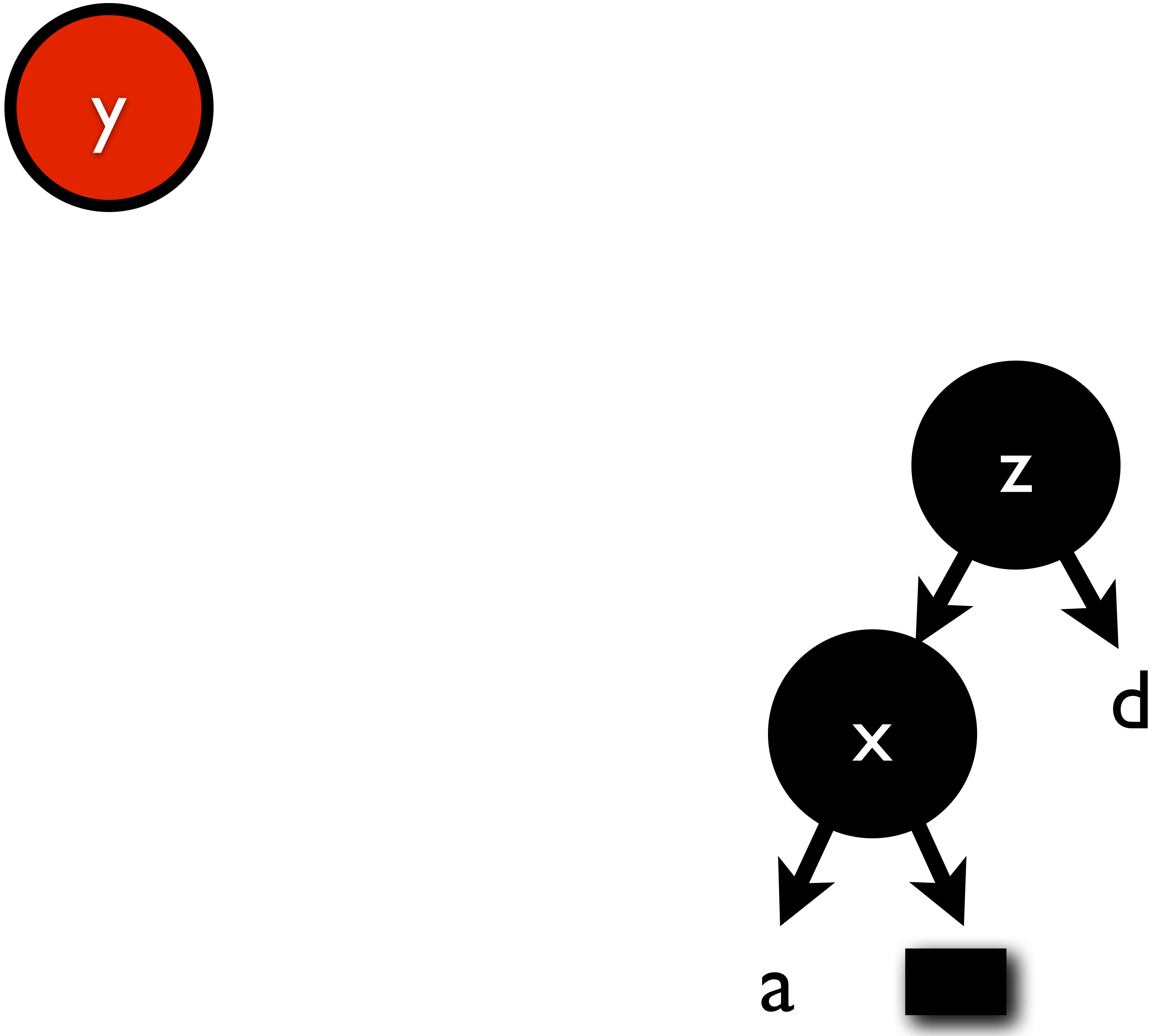

2
1
3

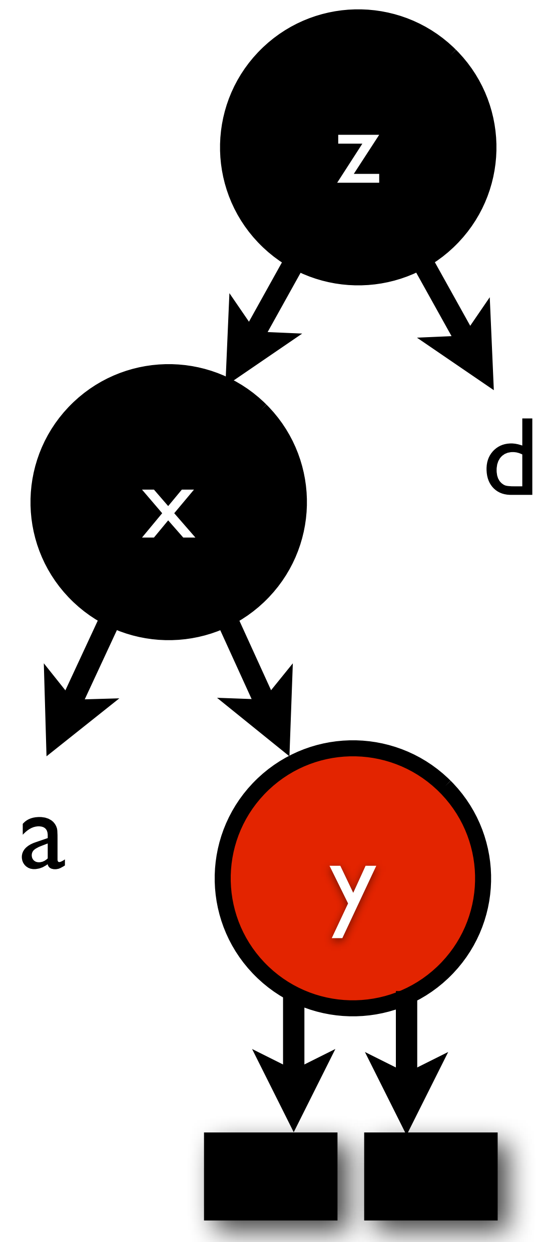
Insertion

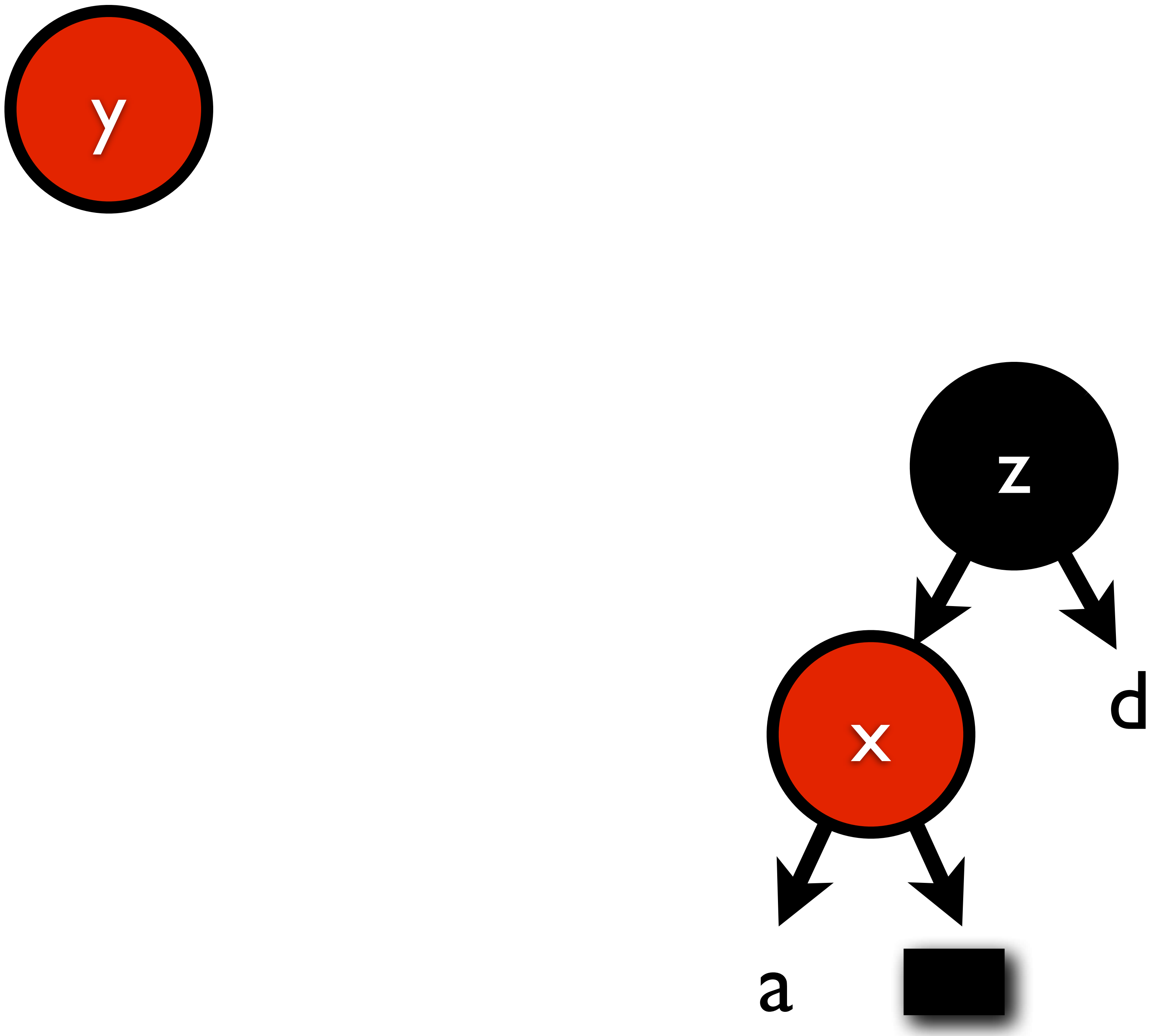
[redacted]

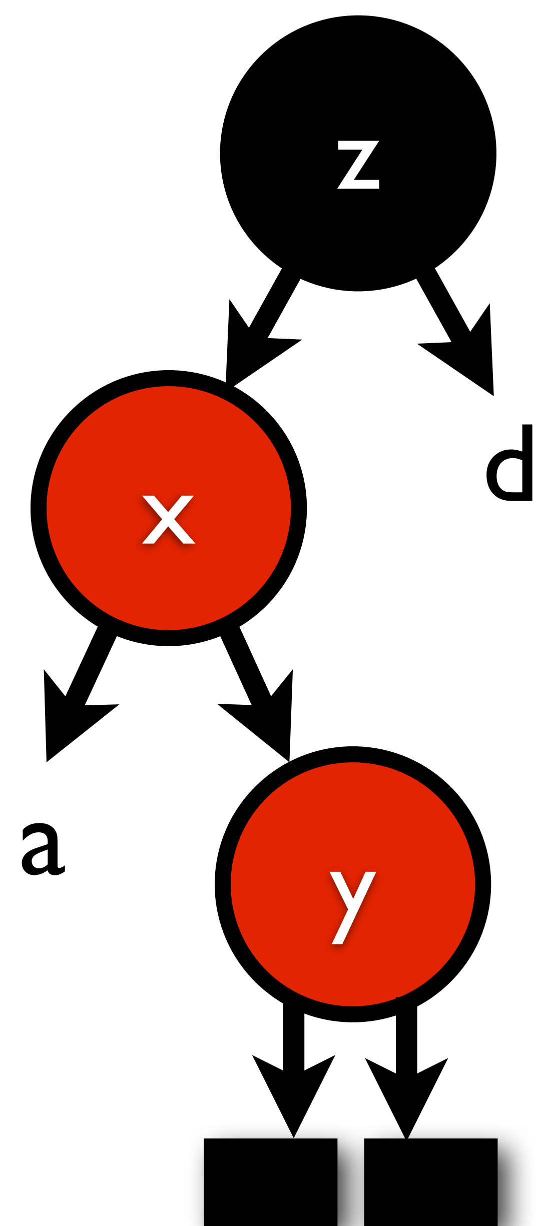
y

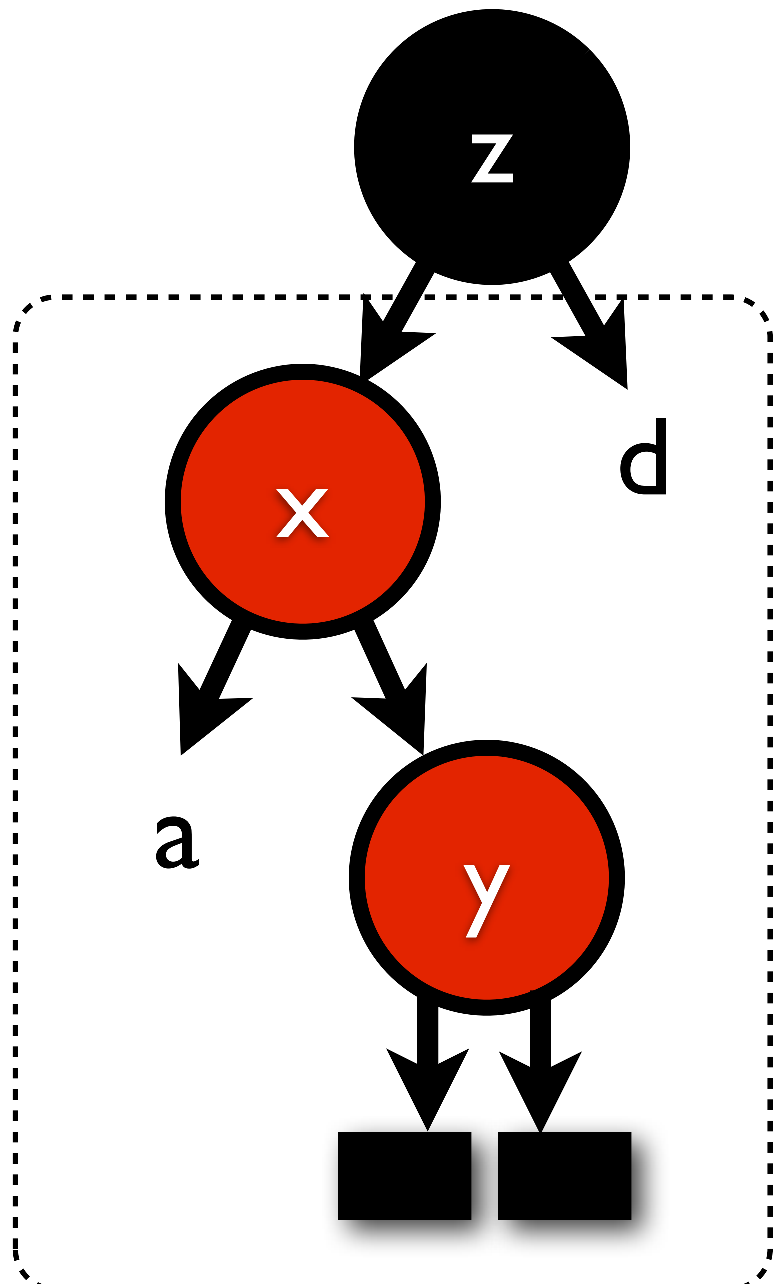

y

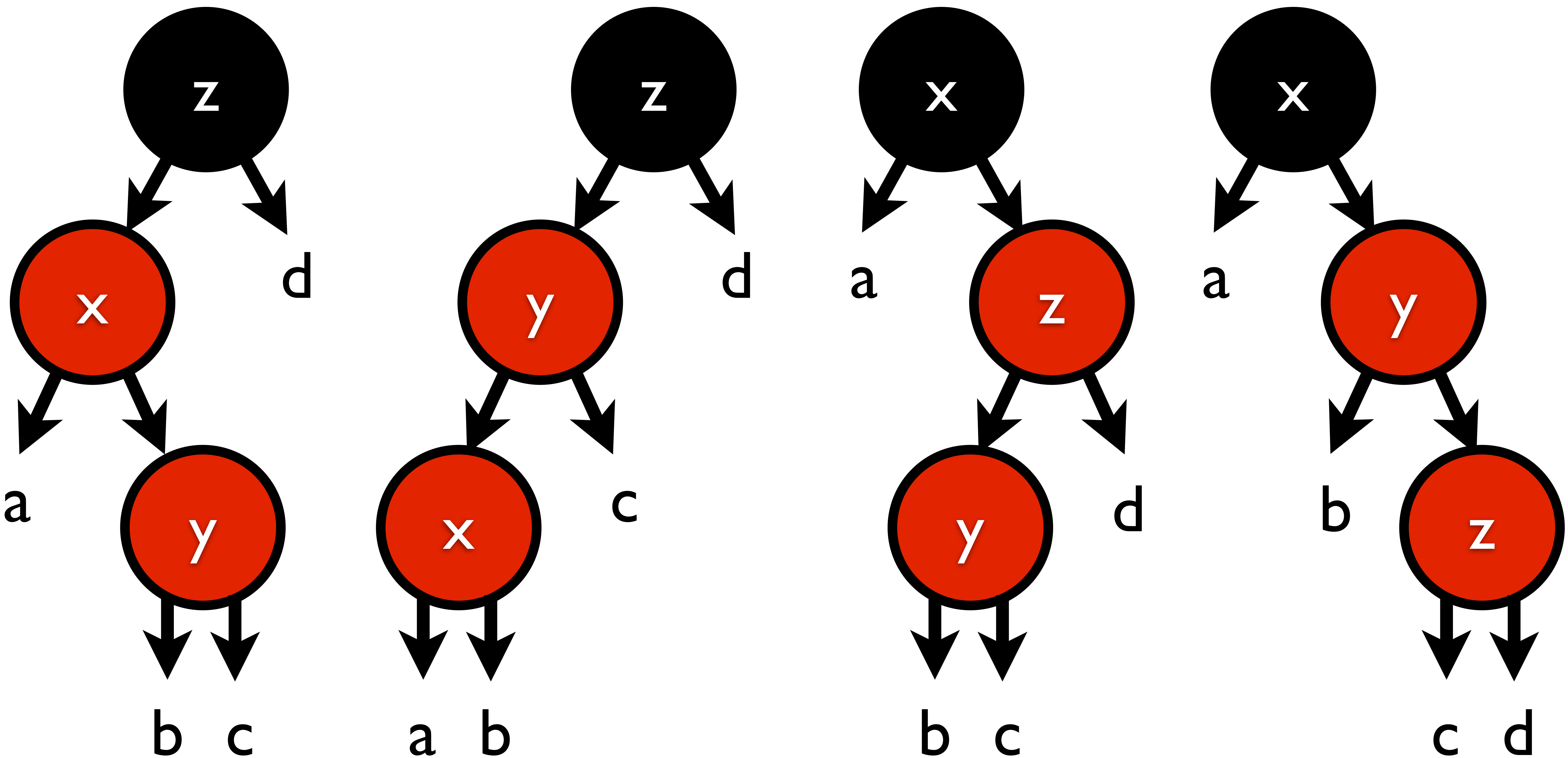

y
z
x
d
a

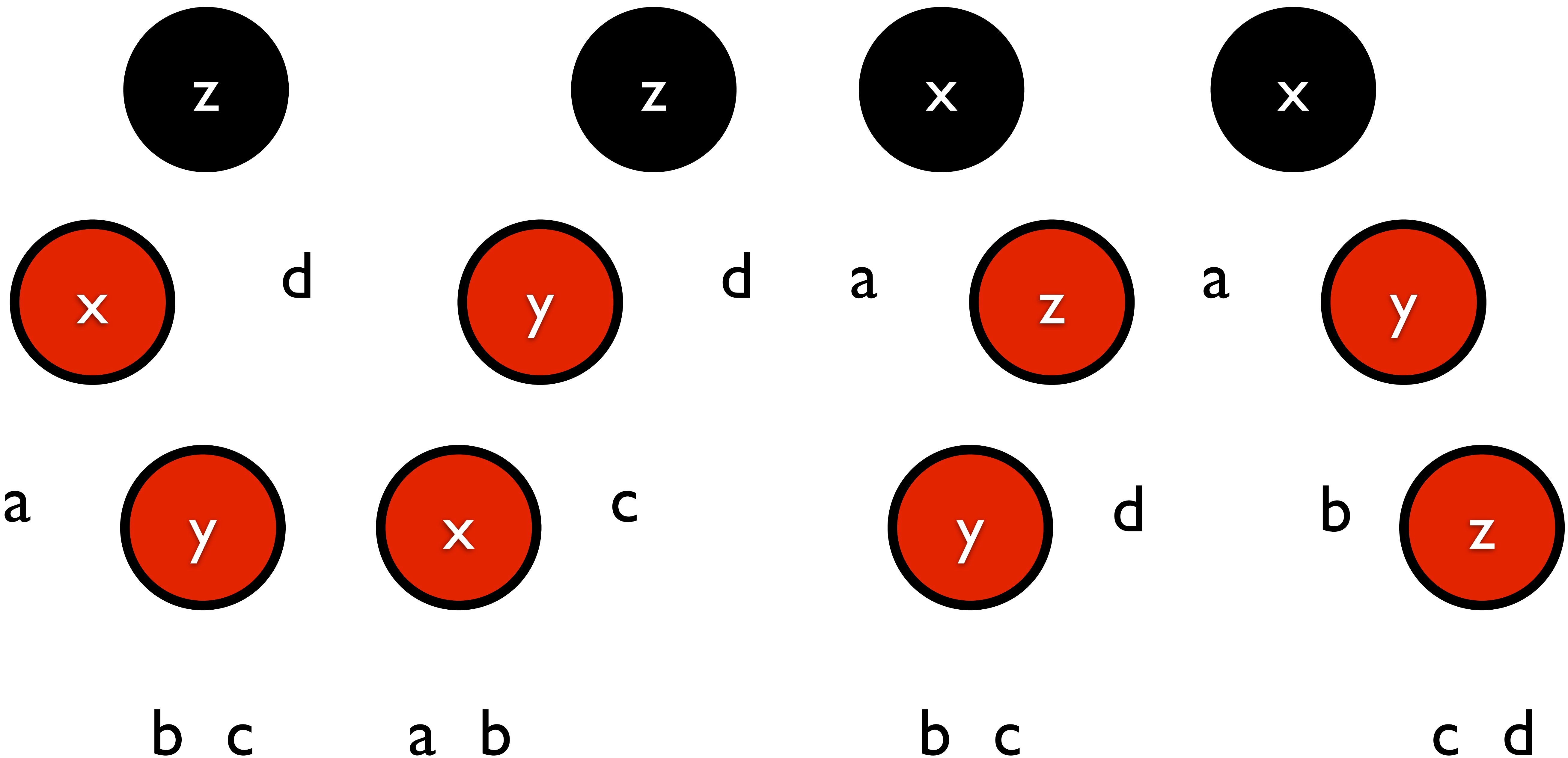

z
x
d
a
y

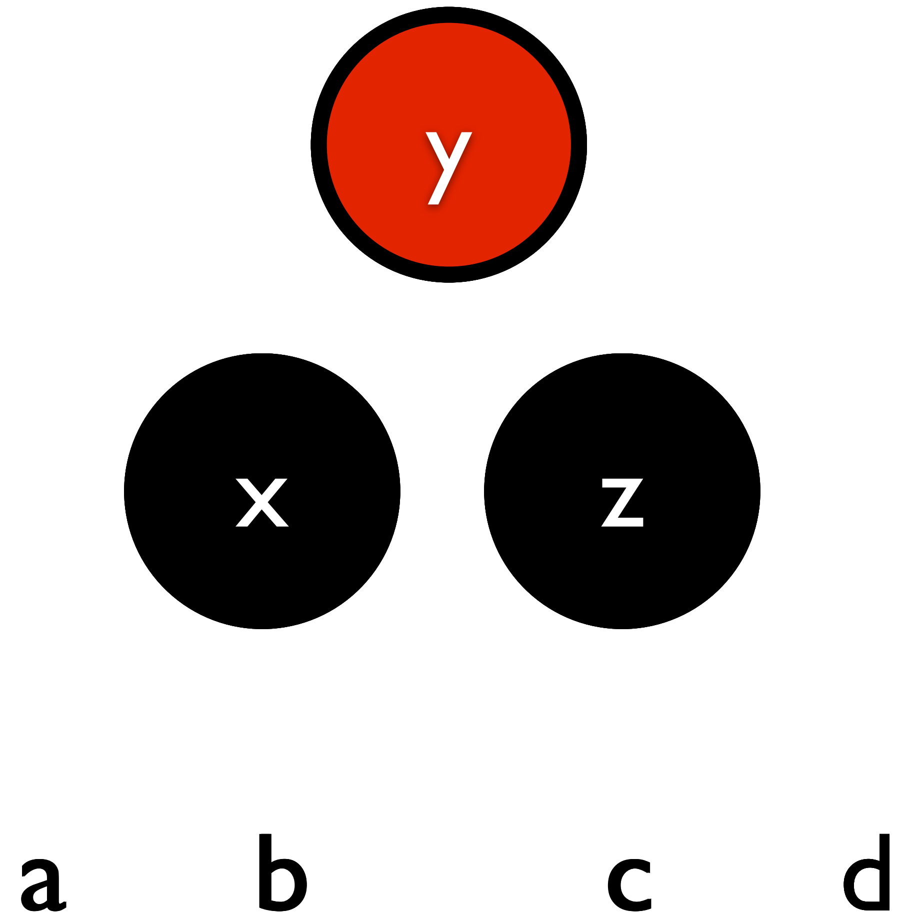

y
z
x
d
a

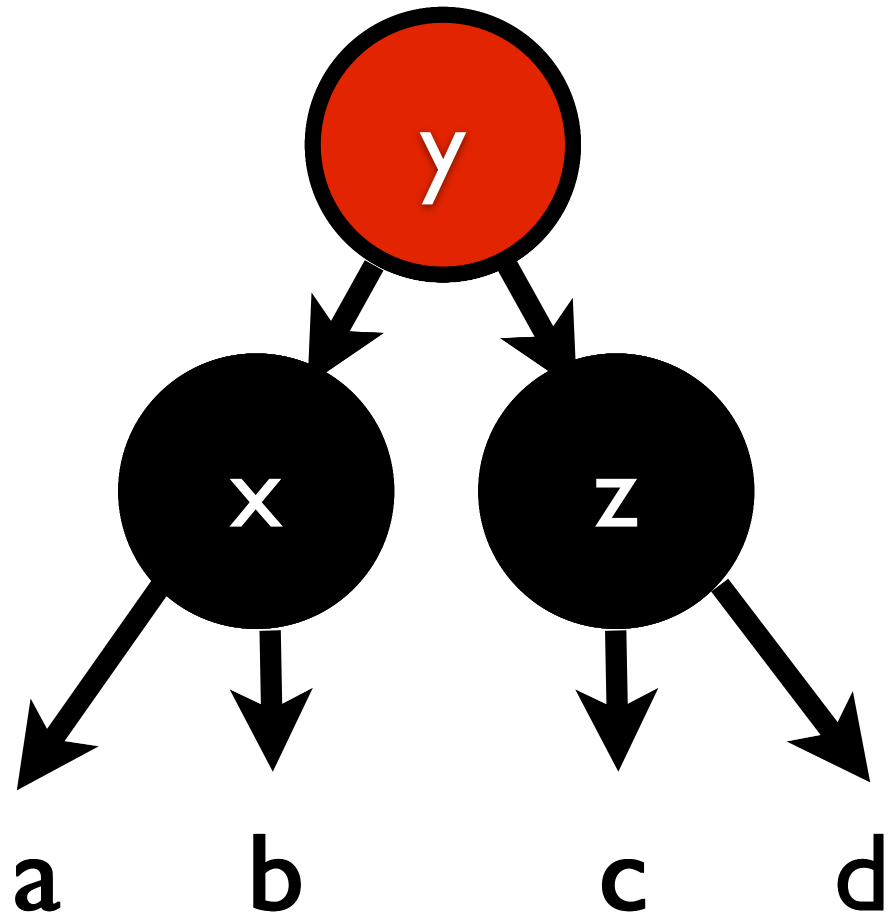

z
x
d
a
y

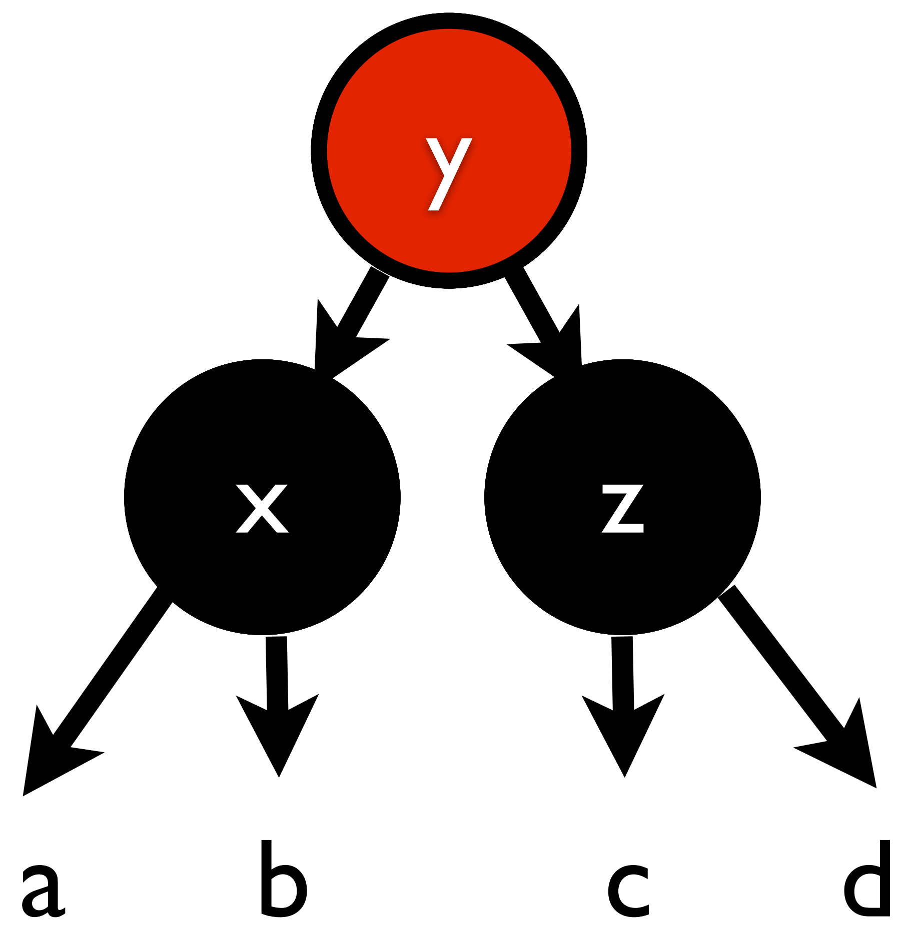

z
x
d
a
y


z
x
d
a
y
b c
z
y
d
x
c
a b
x
a
z
y
d
b c
x
a
y
b
z
c d


z
x
d
a
y
b c
z
y
d
x
c
a b
x
a
z
y
d
b c
x
a
y
b
z
c d


y
x
z
a
b
c
d


y
x
z
a
b
c
d

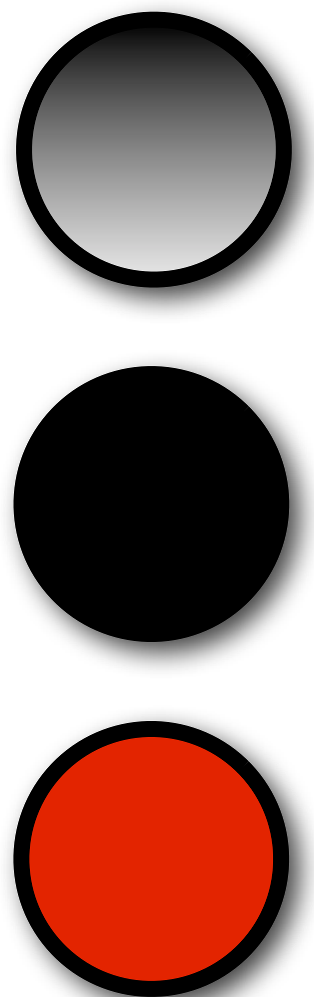

y
x
z
a
b
c
d

```
(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

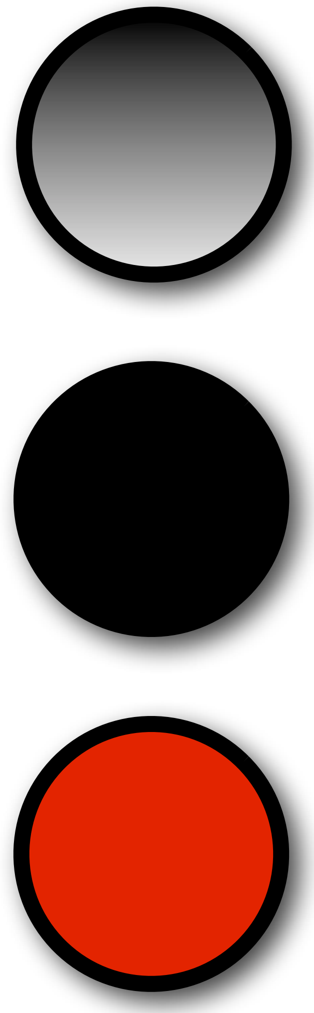
Red



Double black

Black

Red

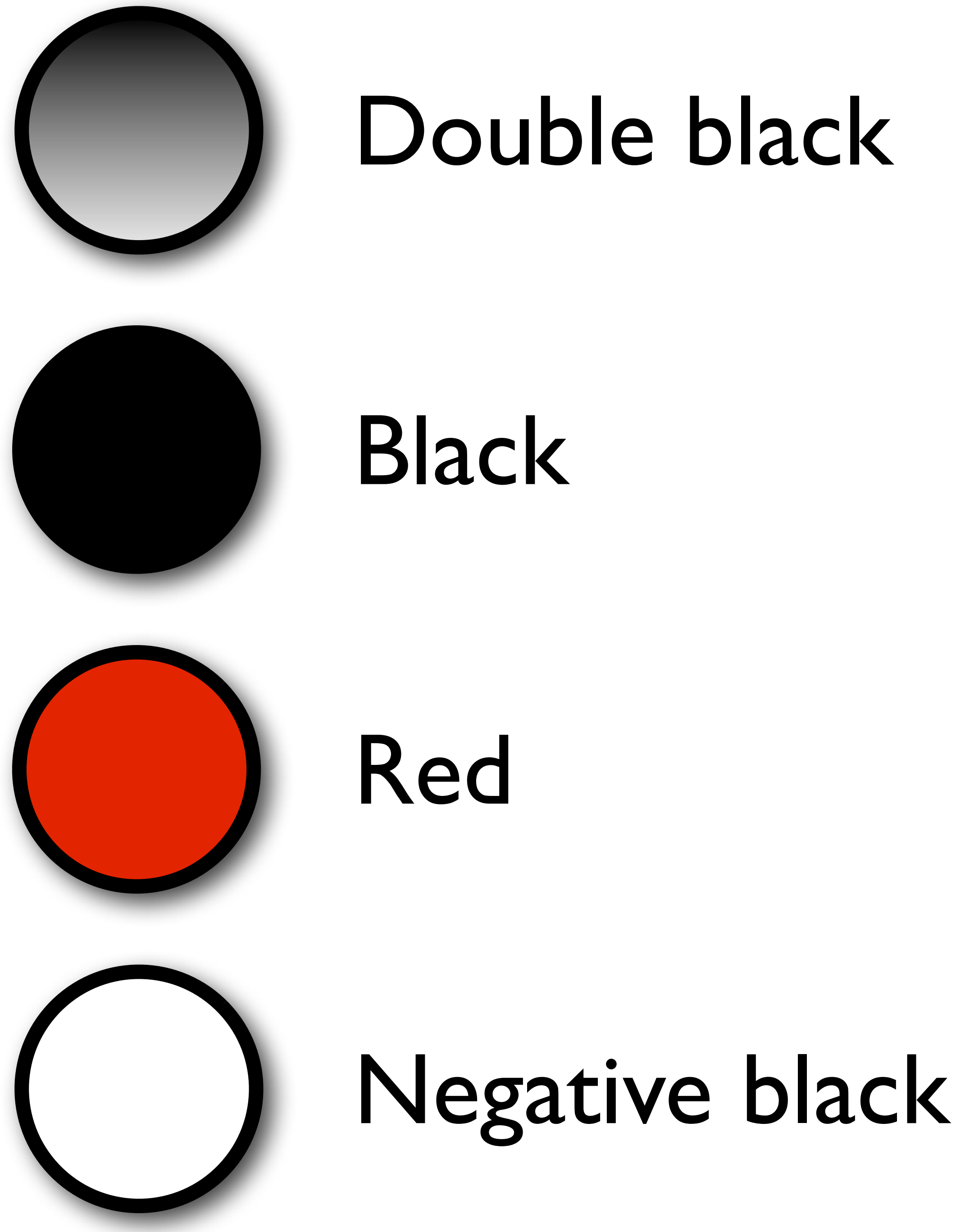


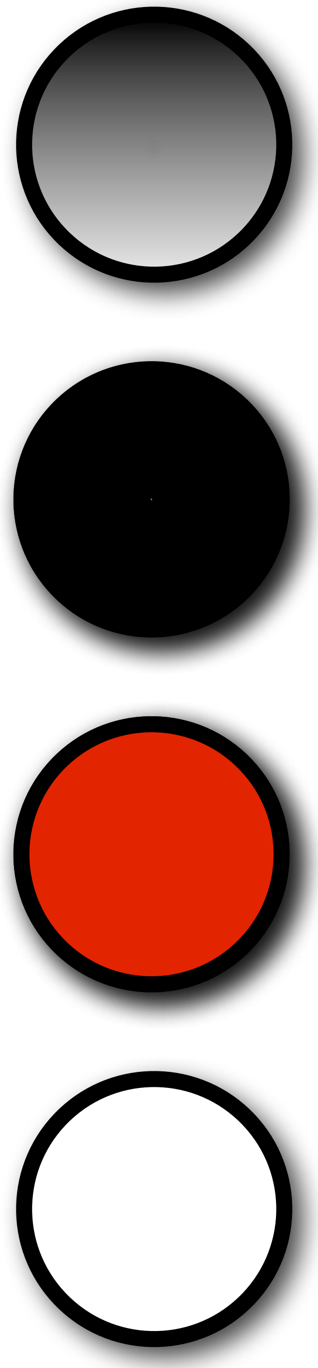
Double black

Black

Red

Negative black


Double black
Black
Red
Negative black

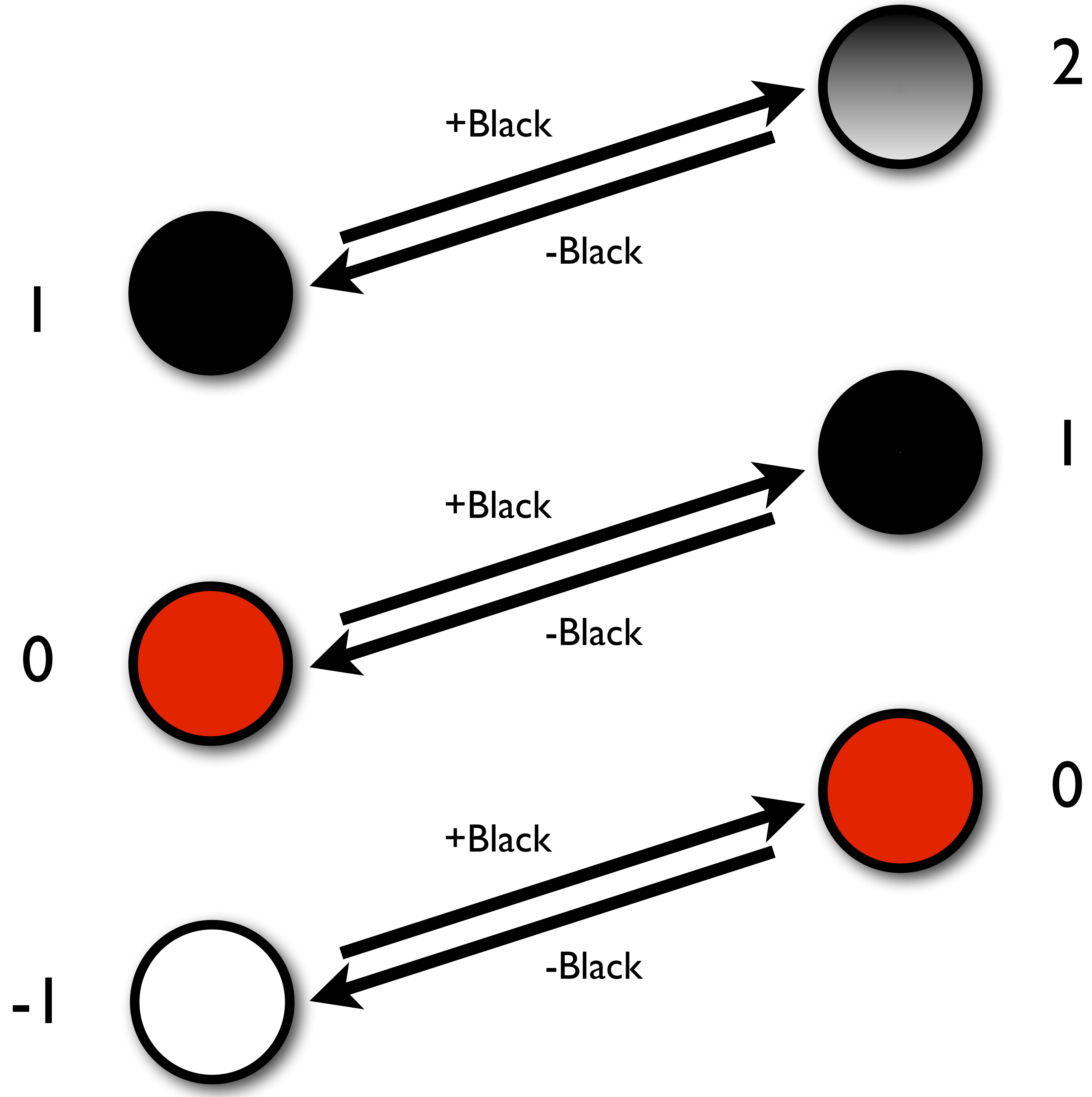


Double black

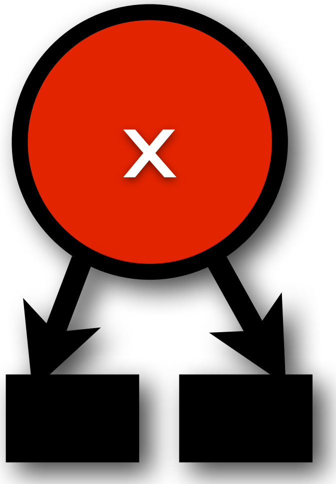
Black

Red

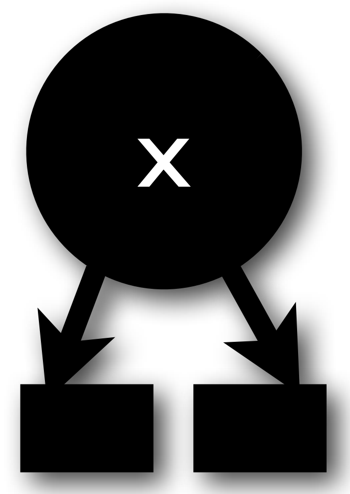
Negative black


2
+Black
-Black
1
1
+Black
-Black
0
0
+Black
-Black
-1

Case 0

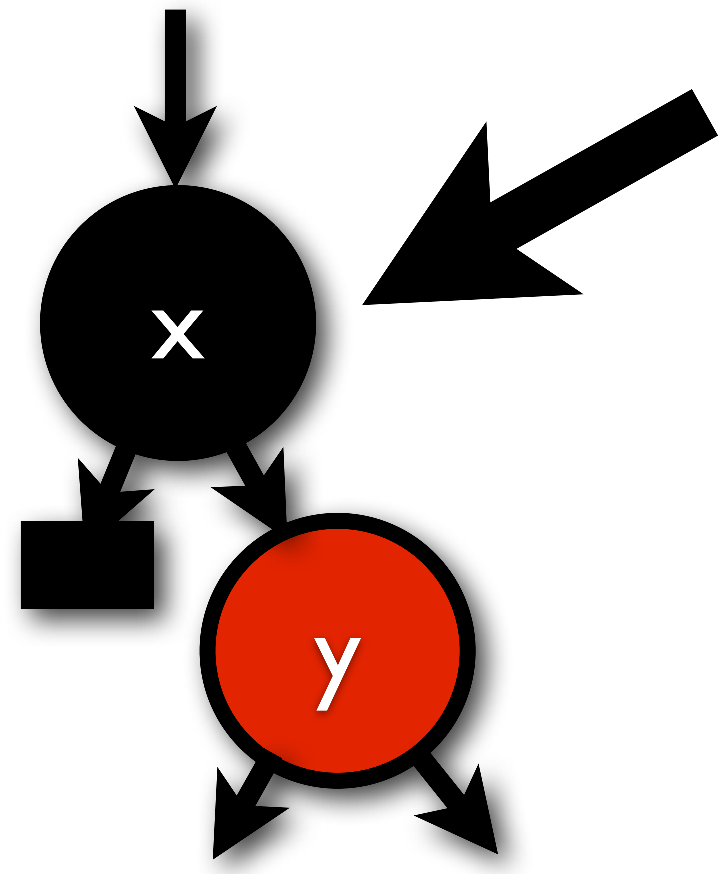

x

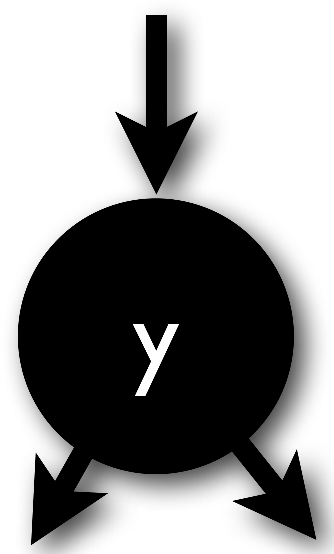
[redacted]

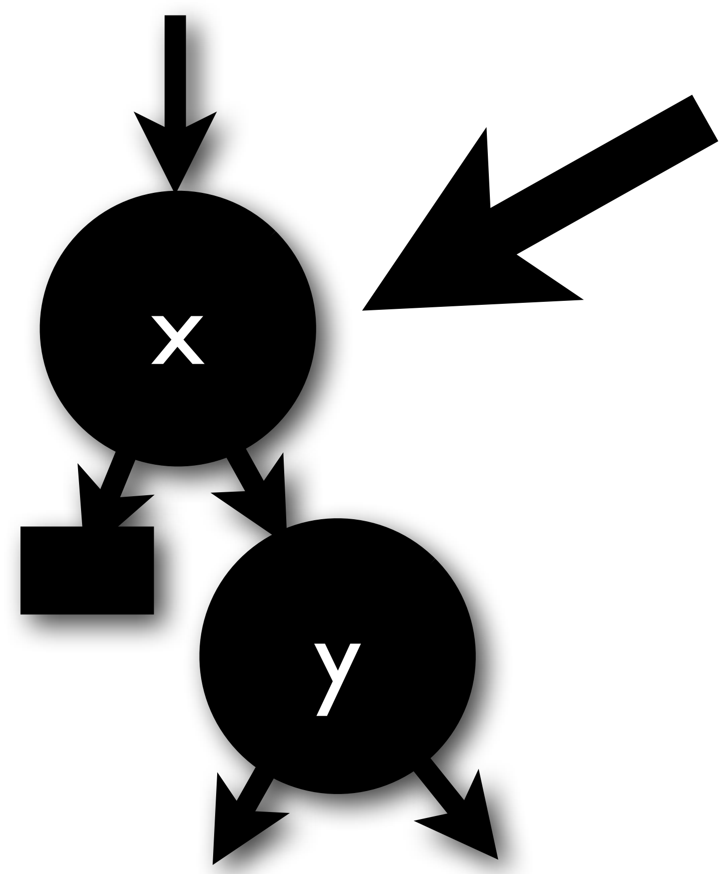

x

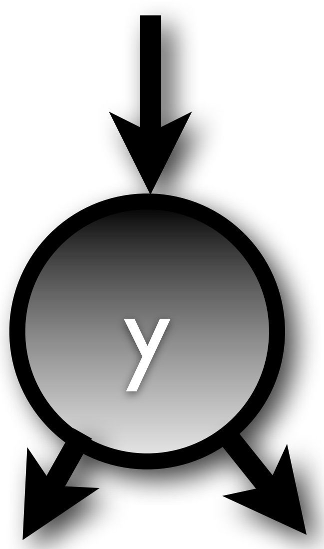
[redacted]

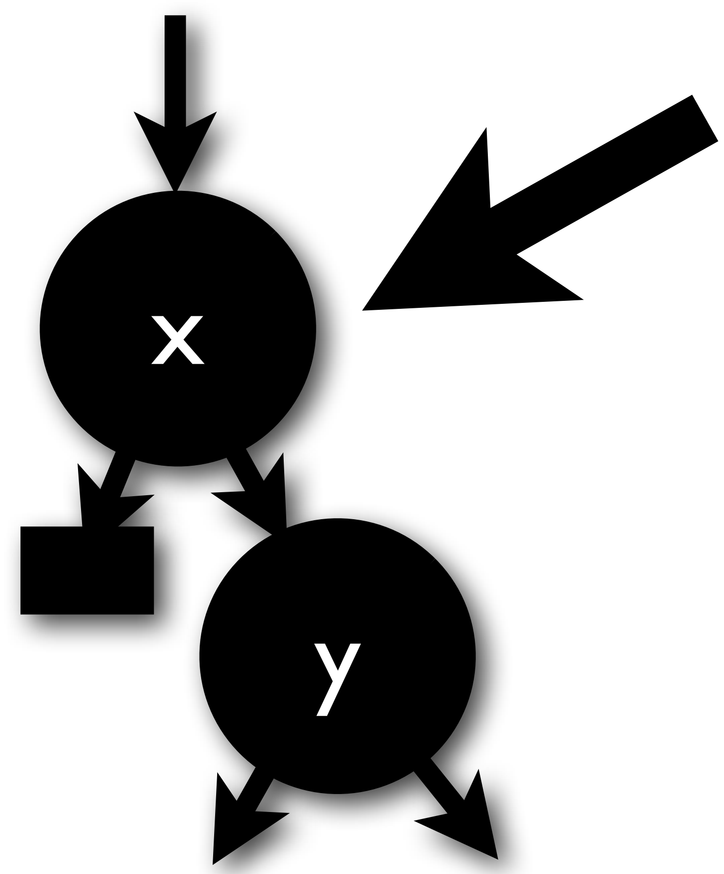
Case 1

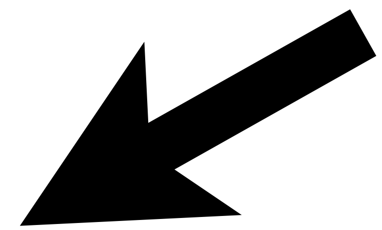

x
y

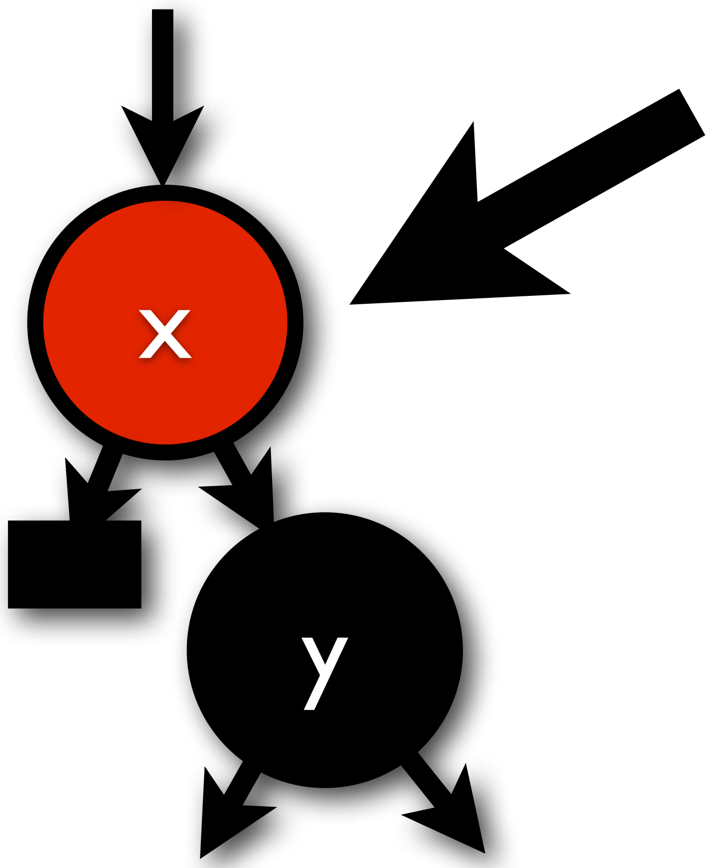

y

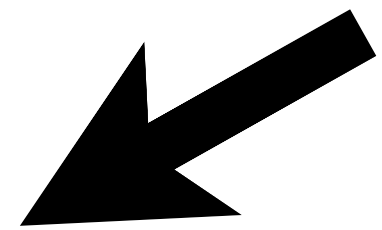

x
y


y


x
y

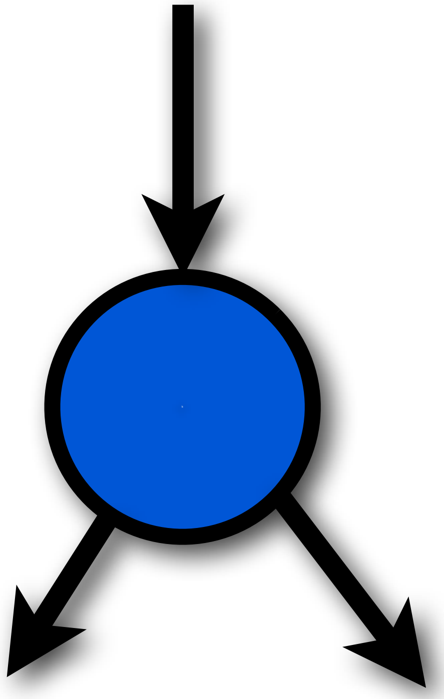


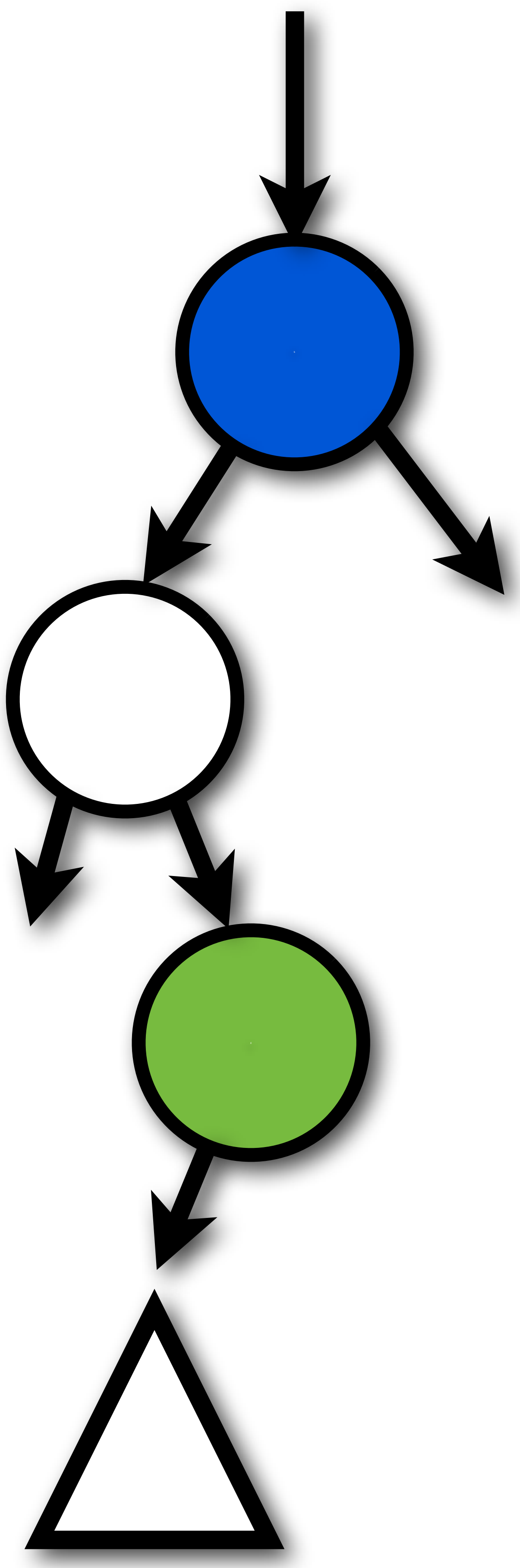

x
y

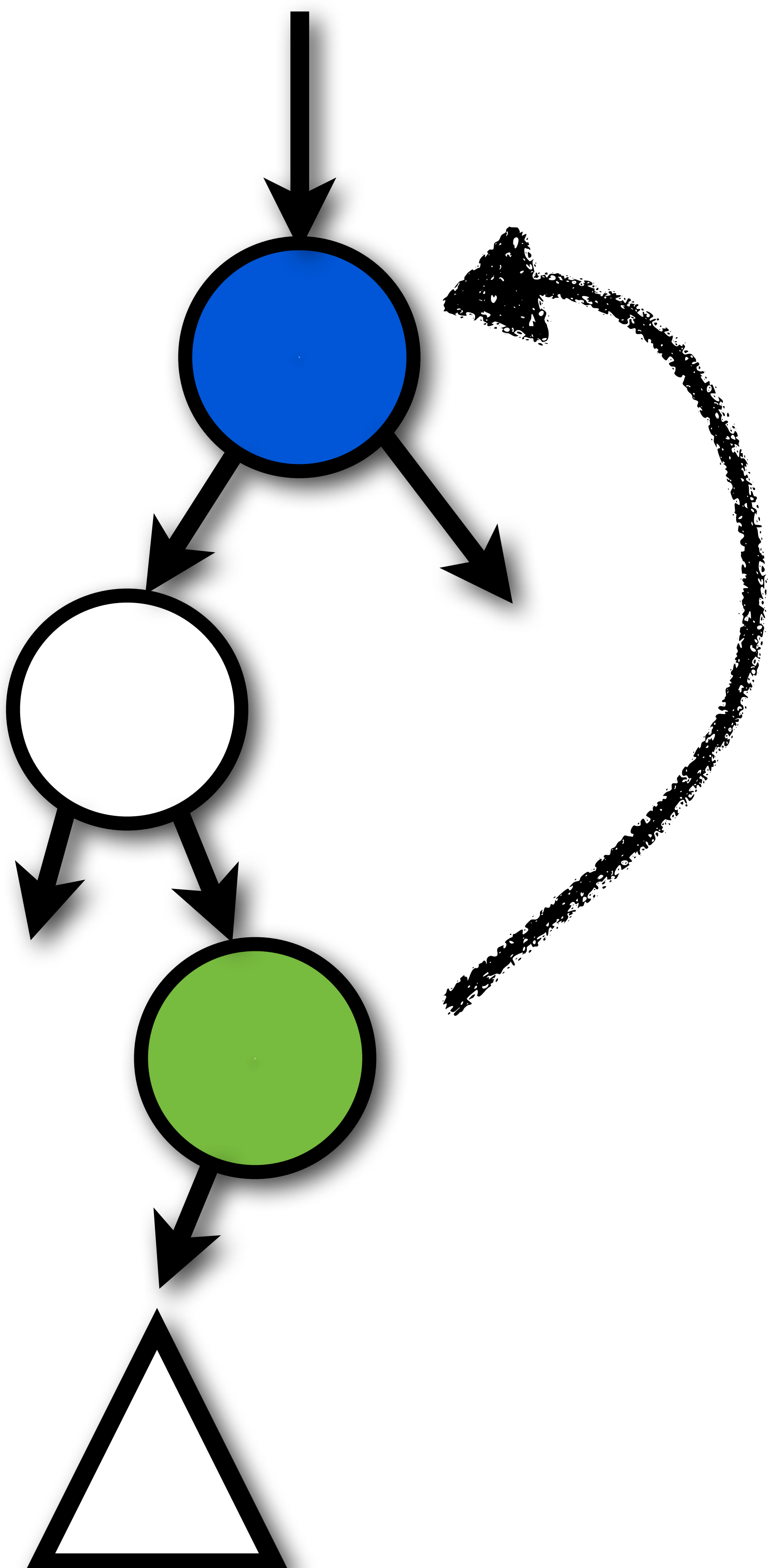


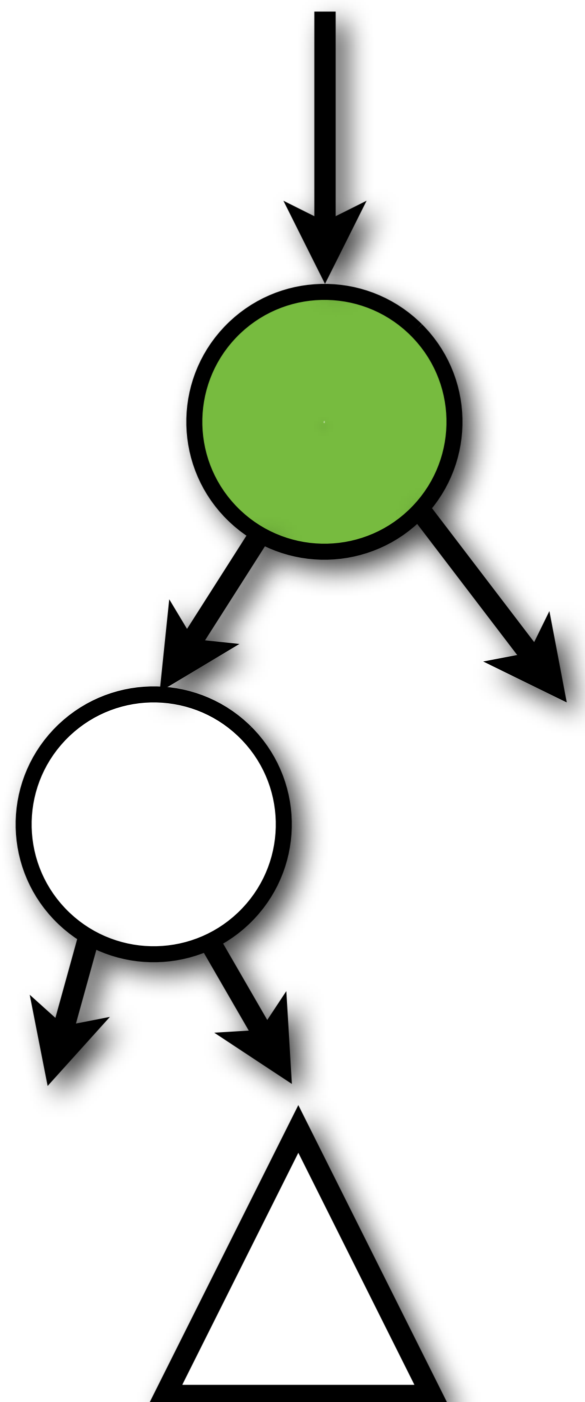
Case 2

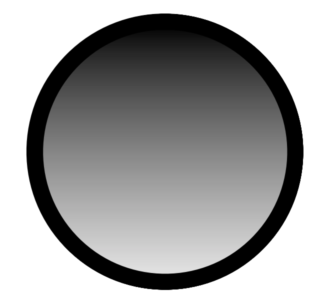
Case 1

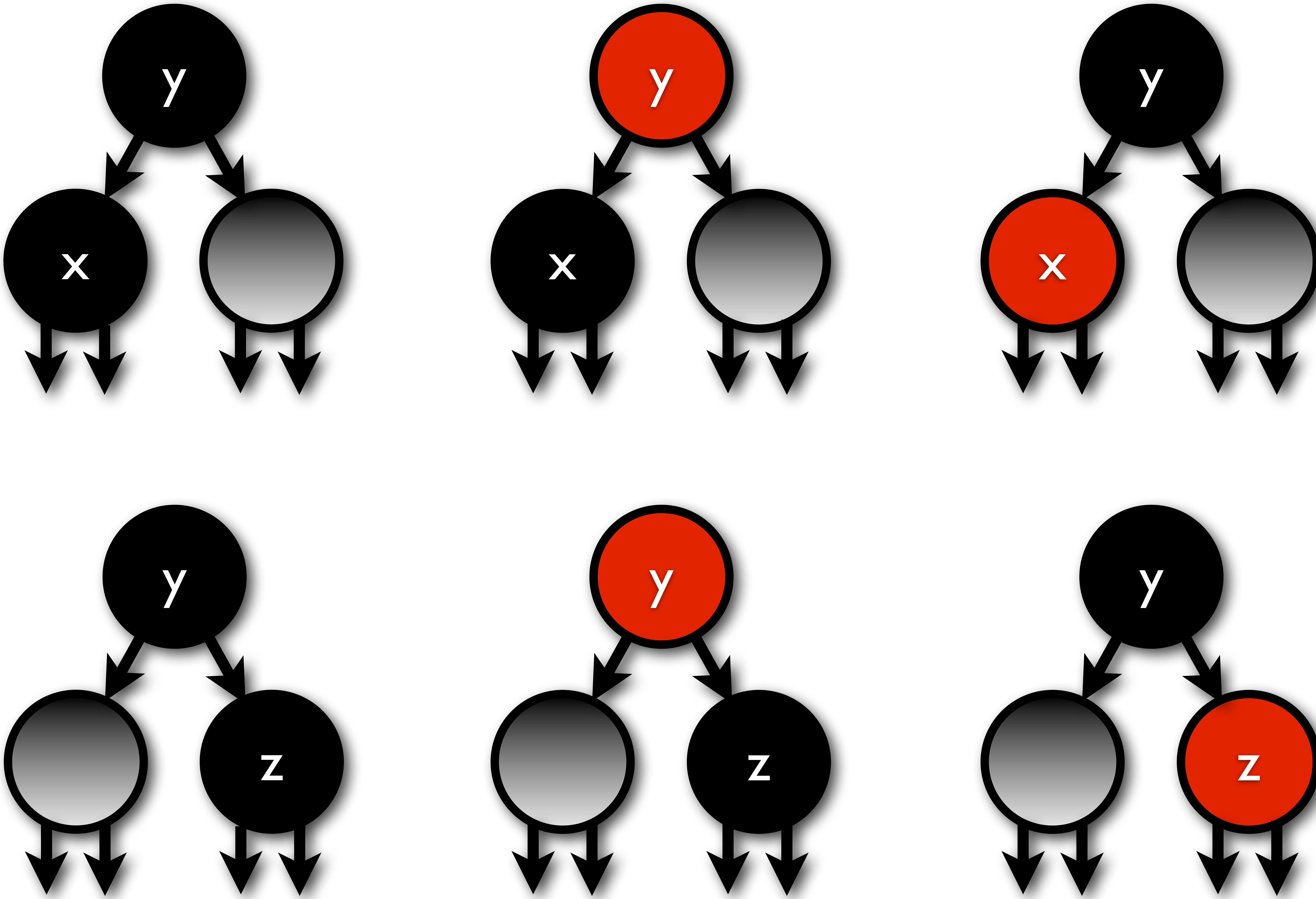


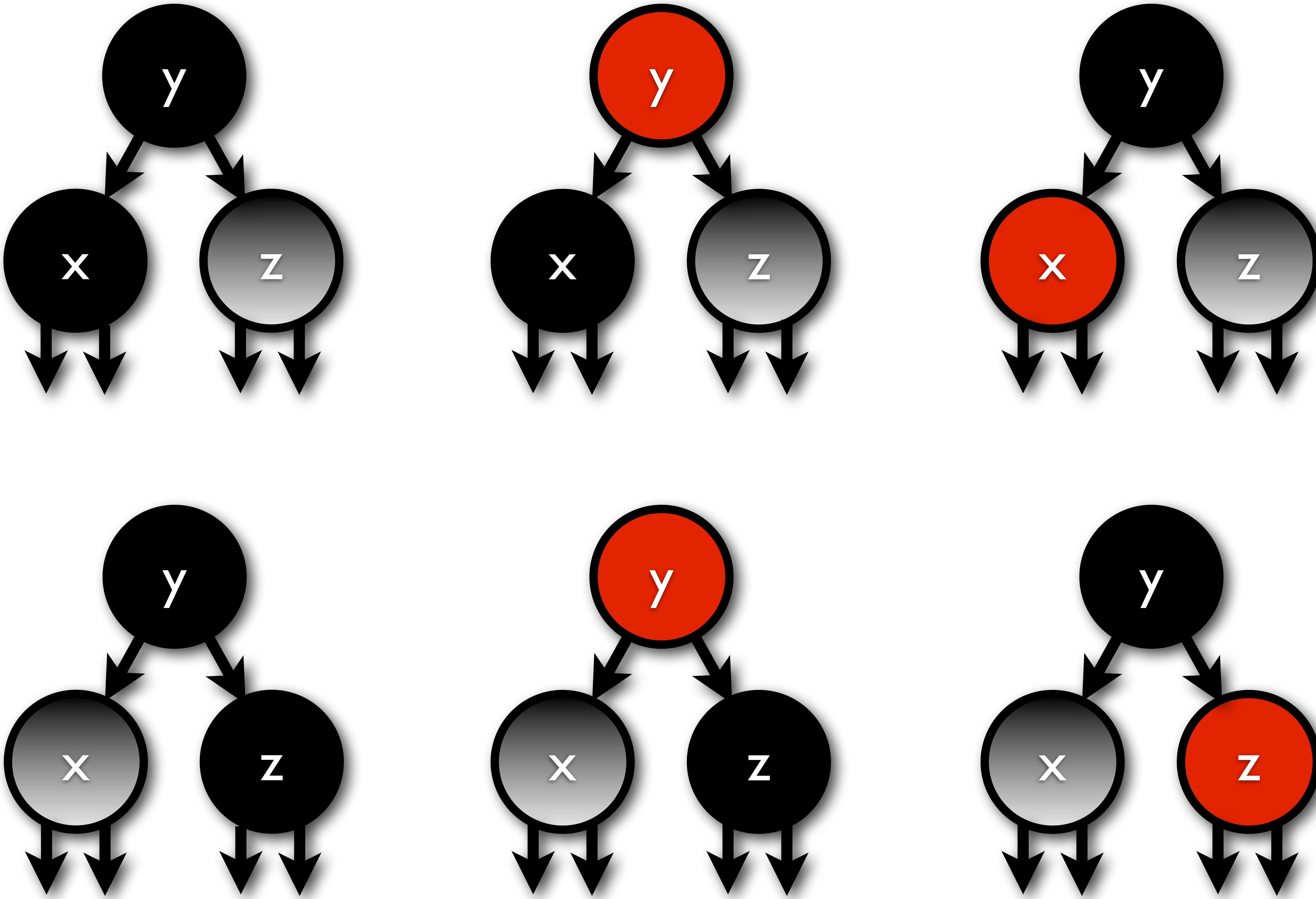




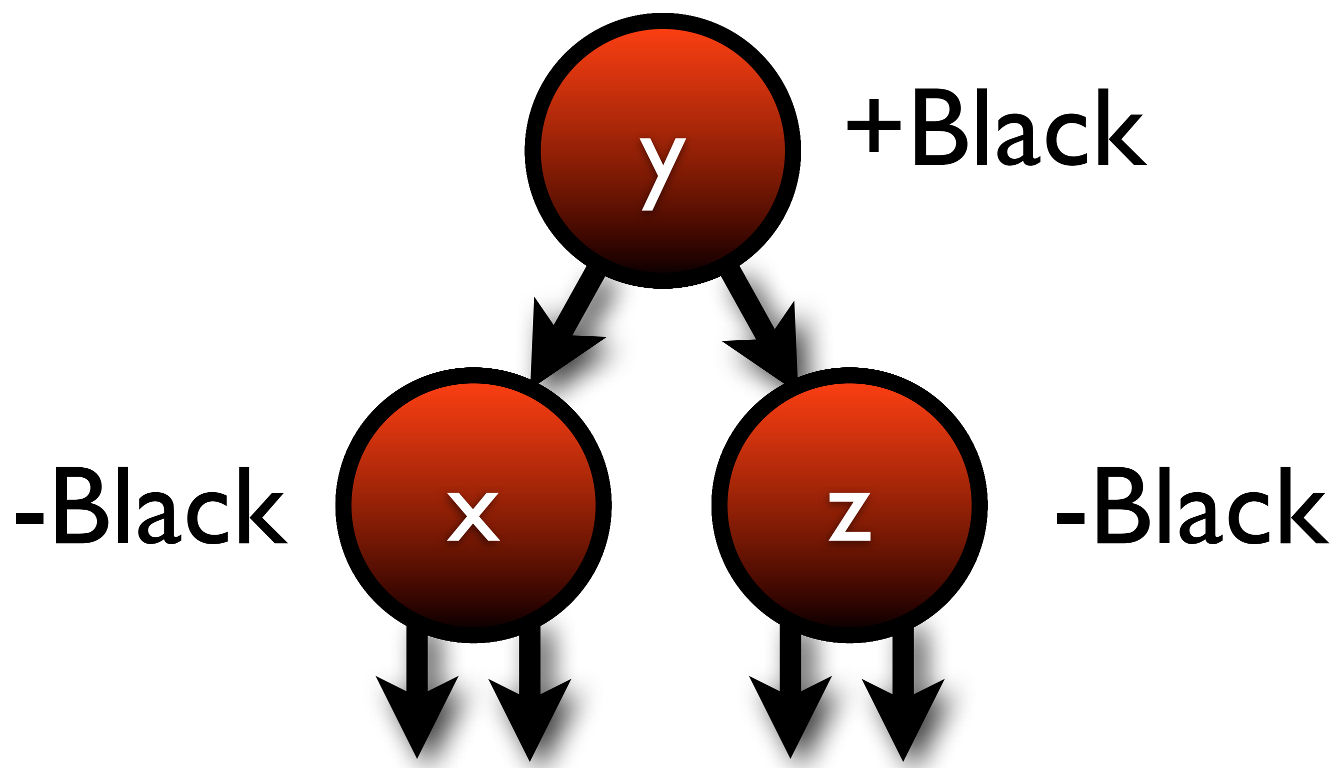


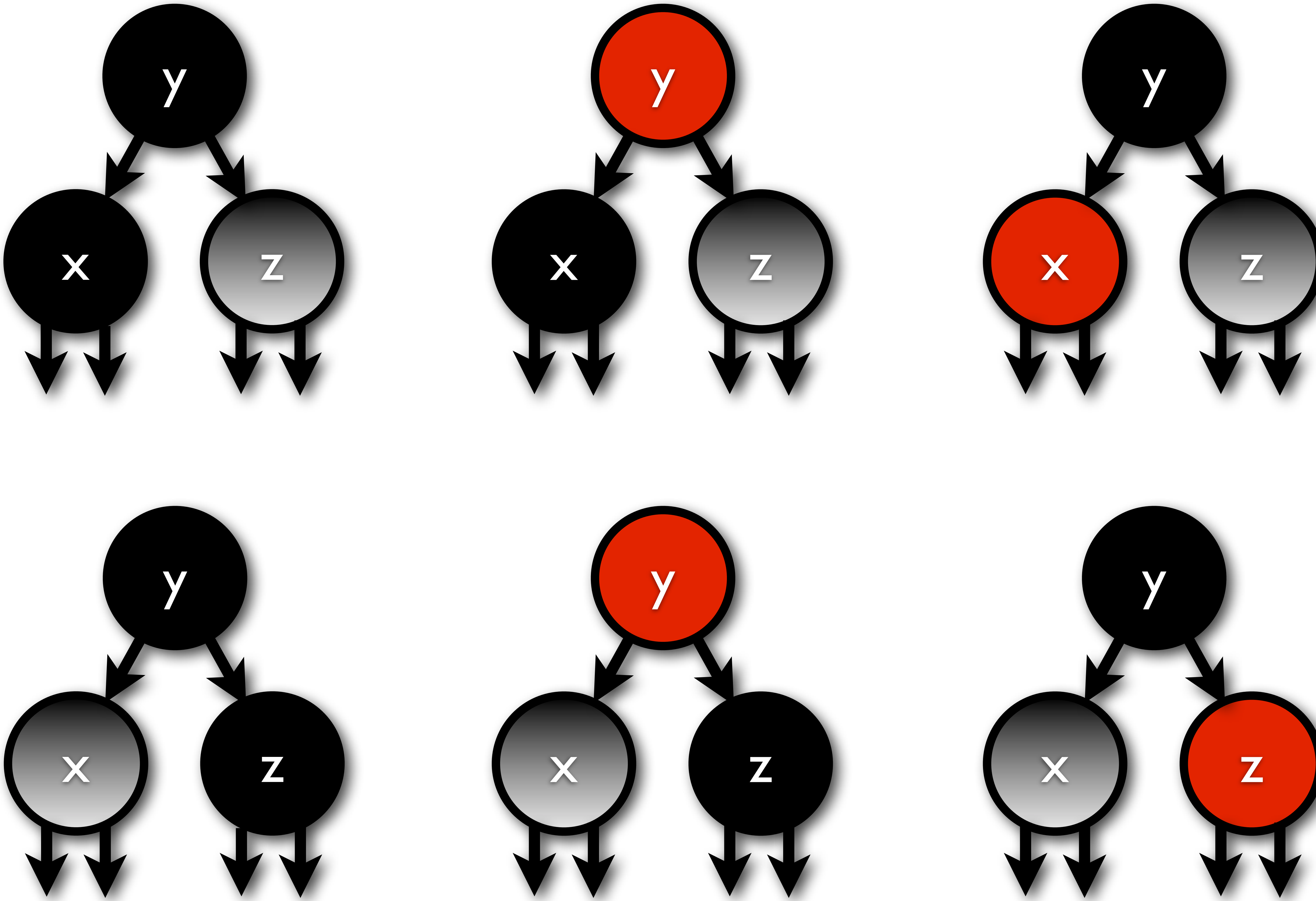
But, what about  ?

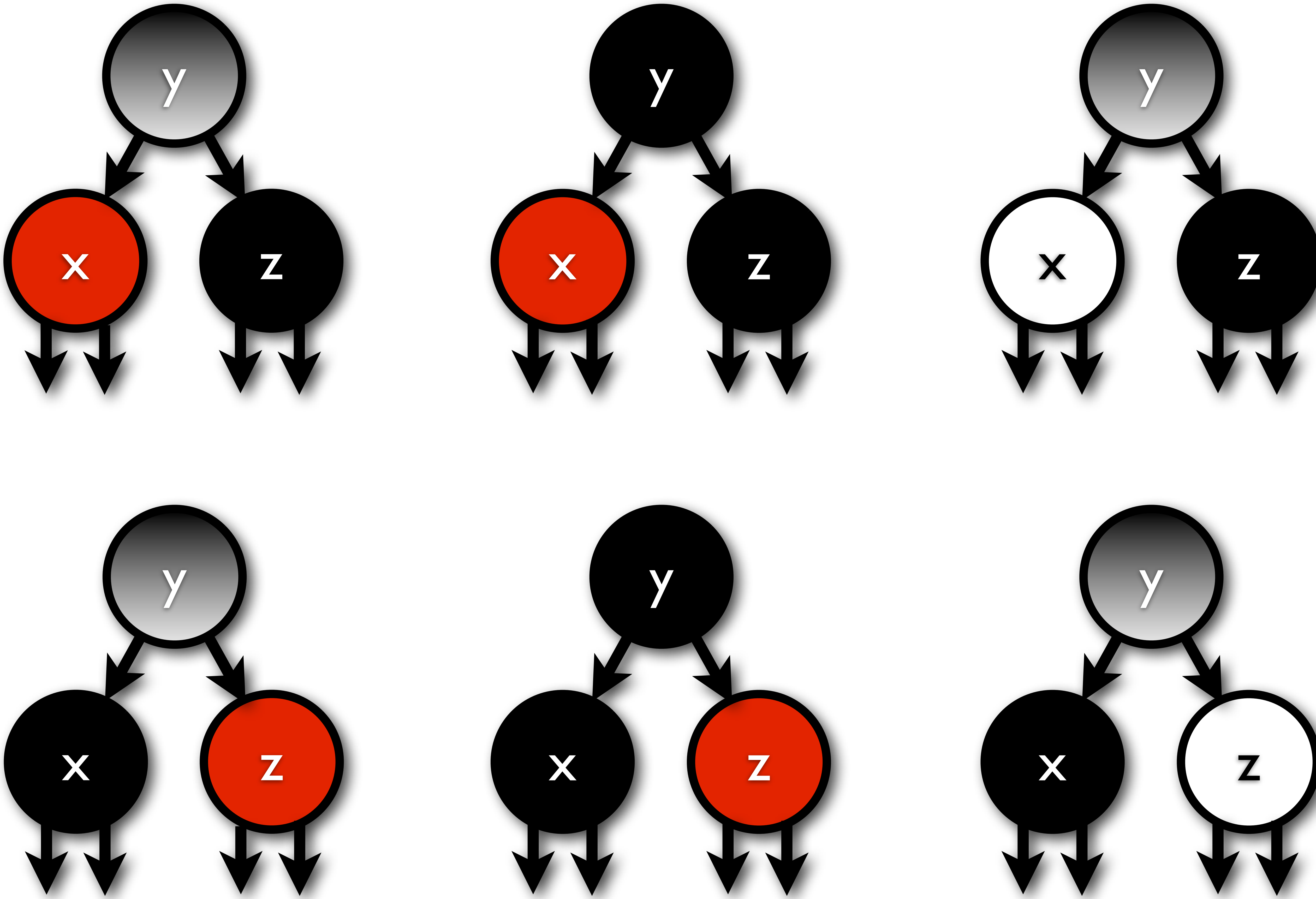

y
x
y
x
y
x
y
z
y
z
y
z

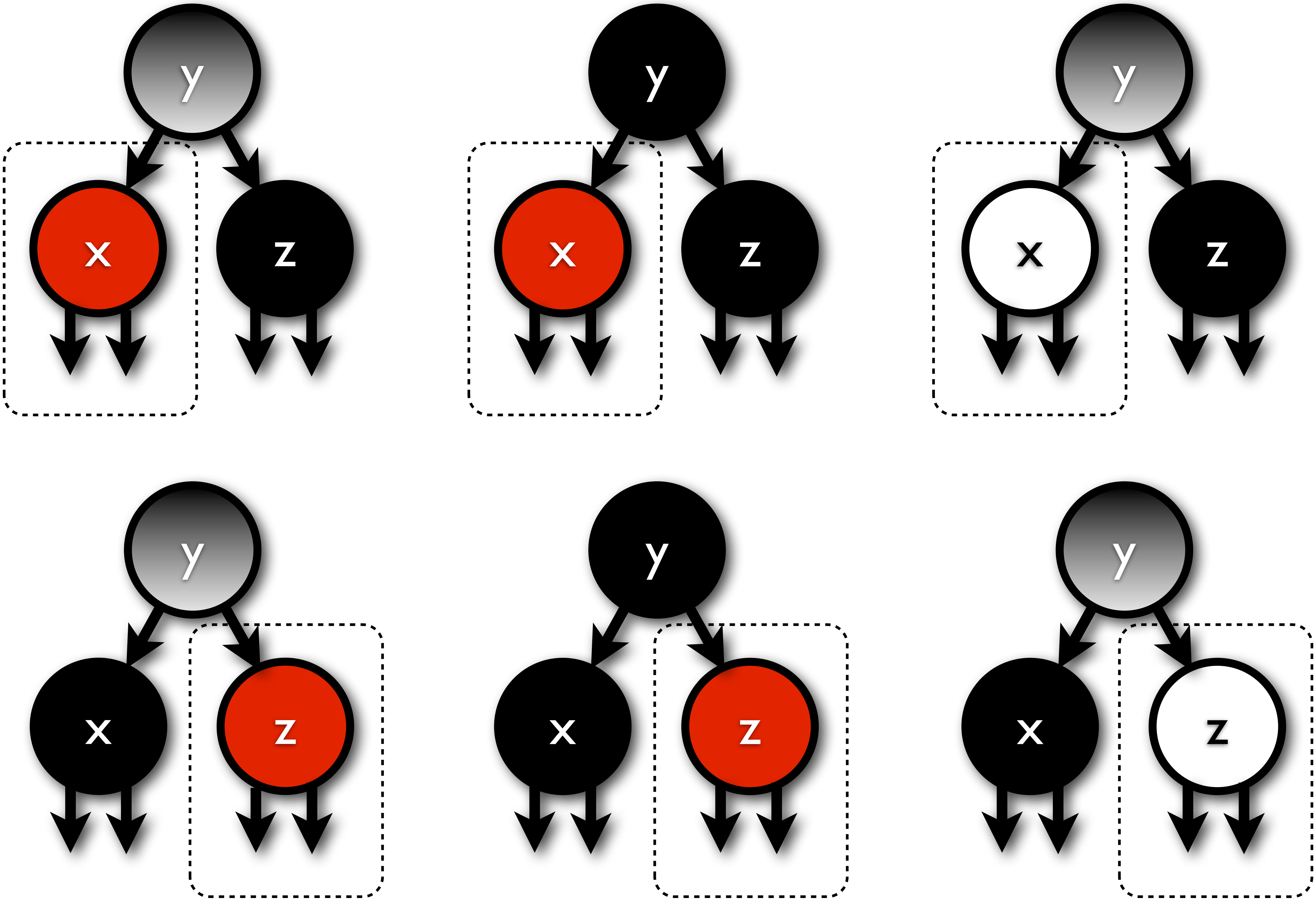

y
x
z
y
x
z
y
x
z
y
x
z
y
x
z
y
x
z

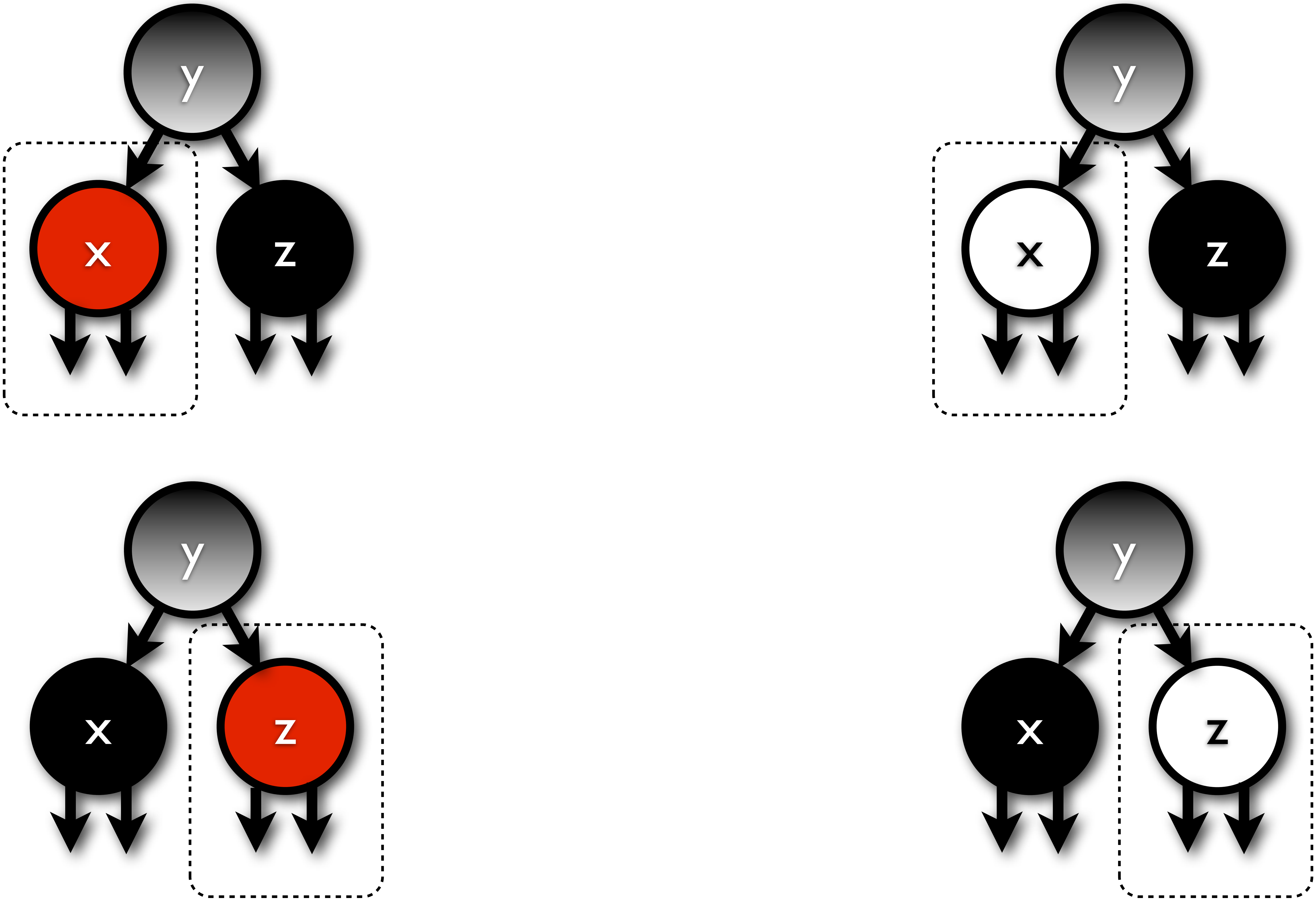
“Bubbling”

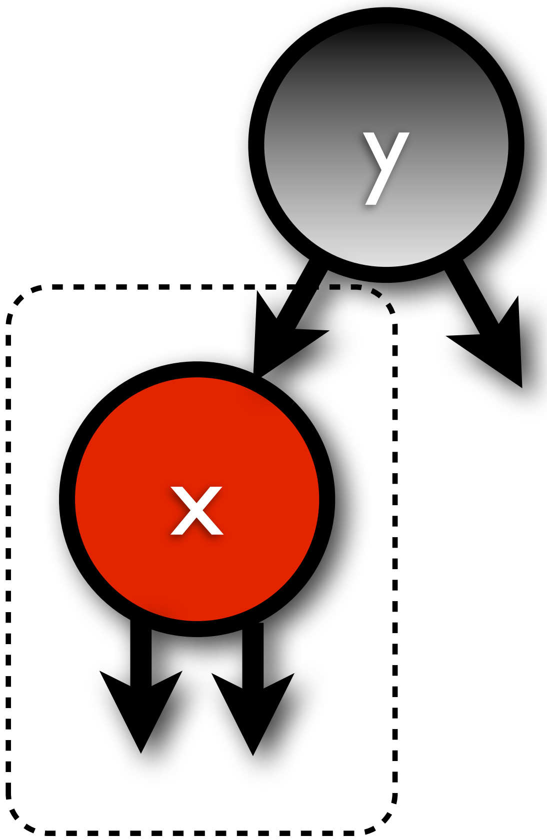


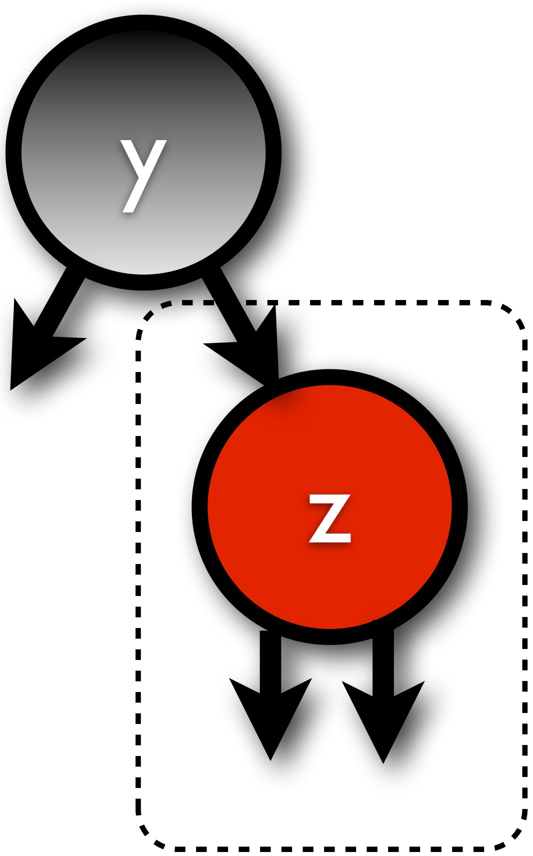

y
x
z
y
x
z
y
x
z
y
x
z
y
x
z
y
x
z

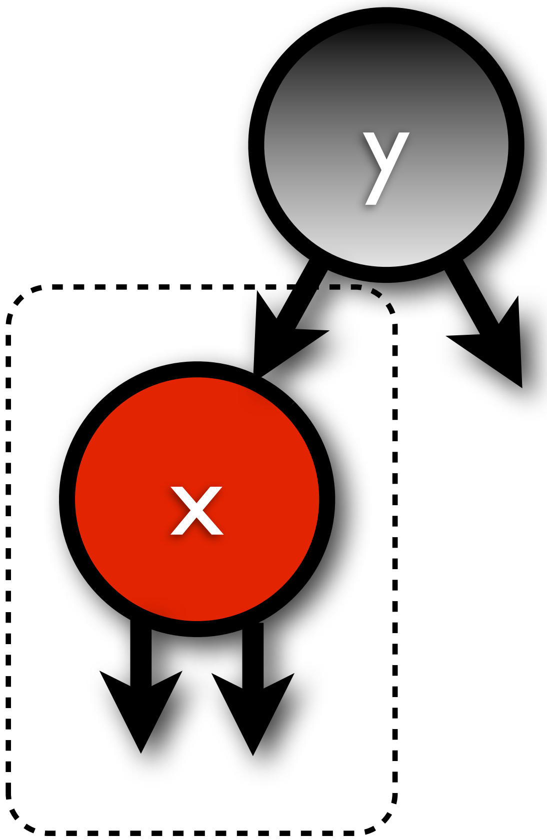

y
x
z
y
x
z
y
x
z
y
x
z
y
x
z
y
x
z

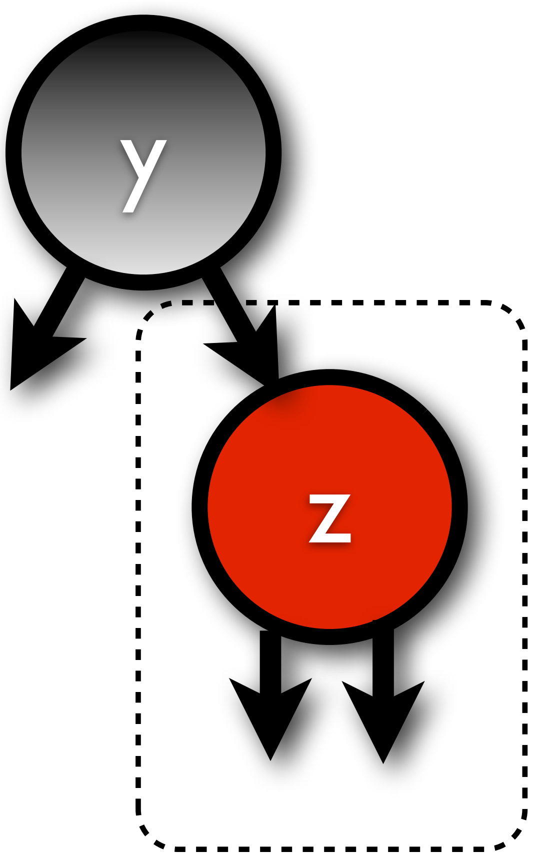

y
x
z
y
x
z
y
x
z
y
x
z
y
x
z
y
x
z

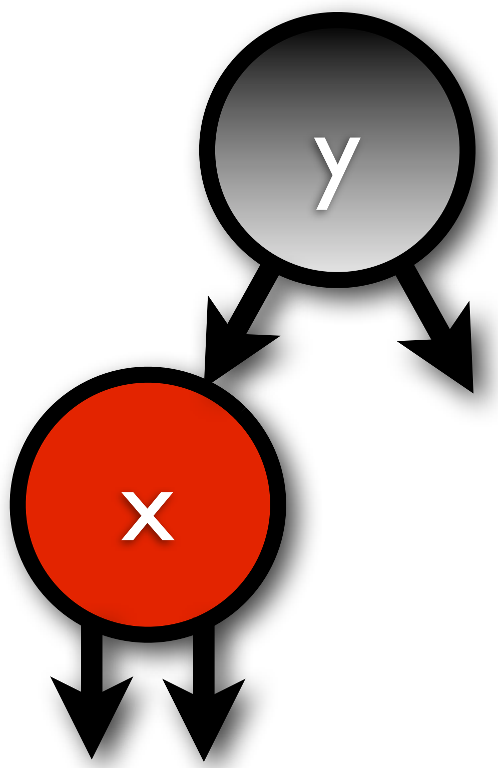

y
x
z
y
x
z
y
x
z
y
x
z

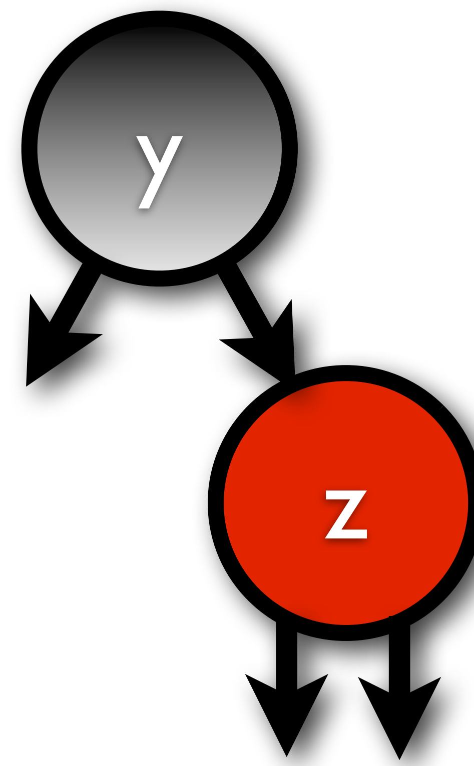

y
x

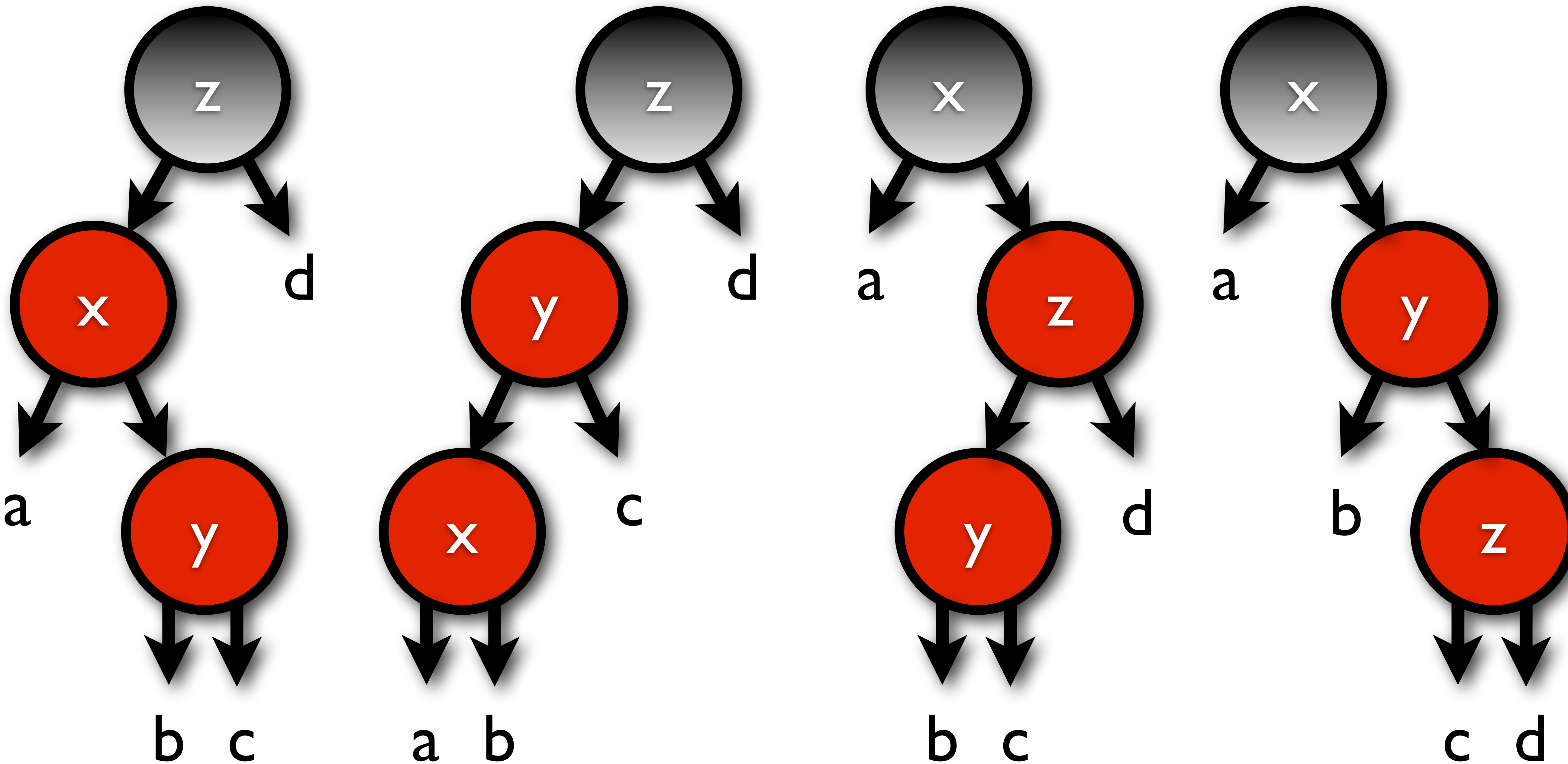

y
z

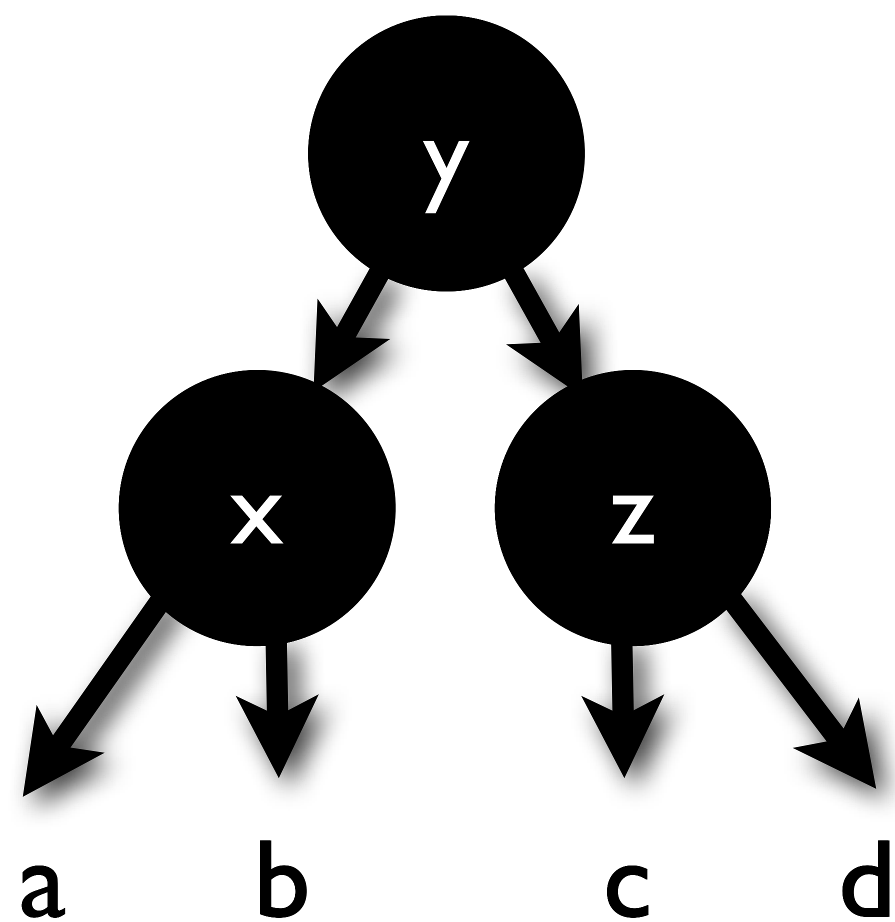

y
x


y
z


y
x

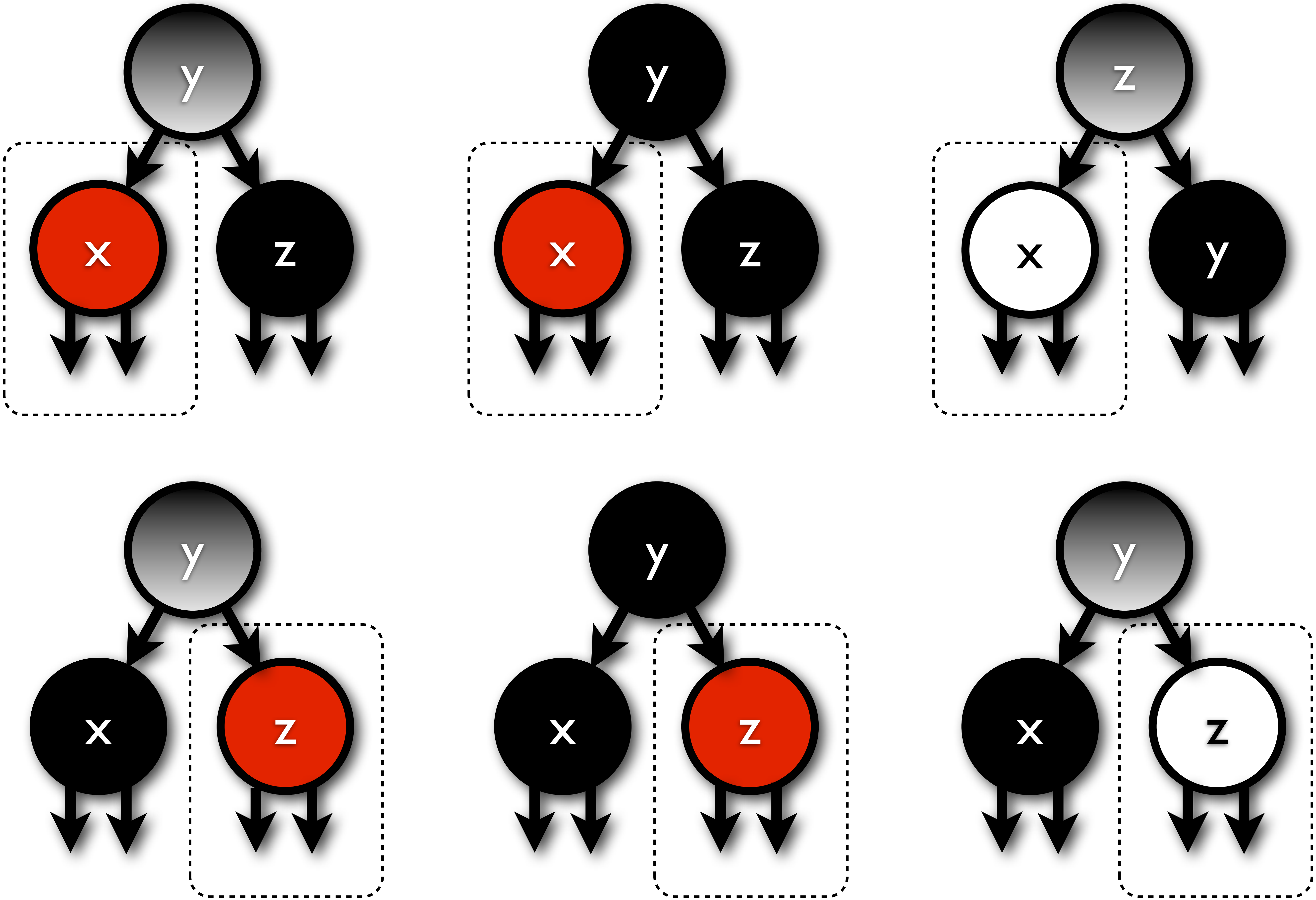

y
z

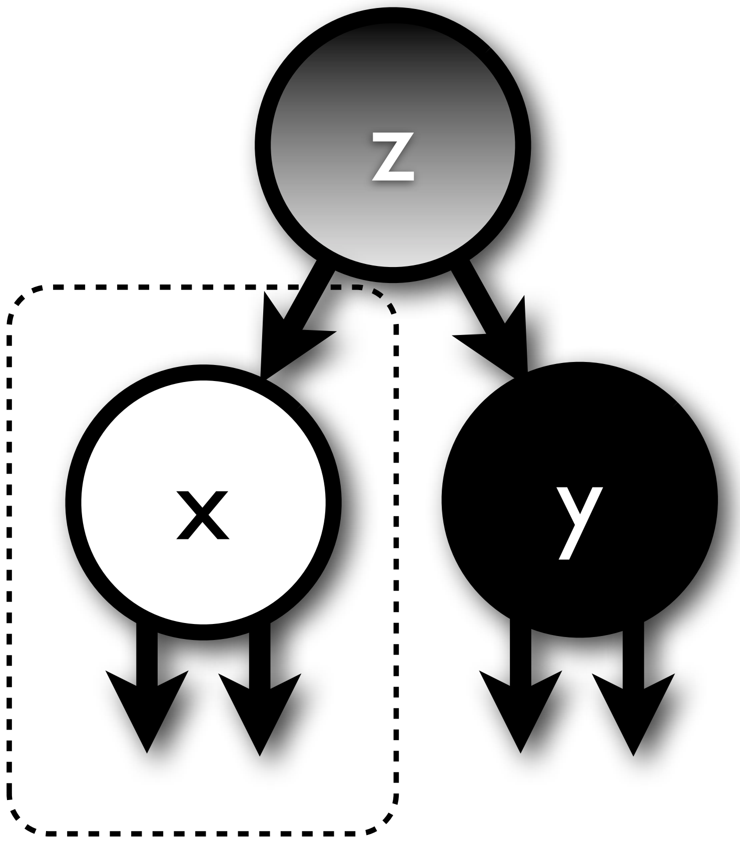

z
d
x
a
y
b c
z
d
y
c
x
a b
x
a
z
d
y
b c
x
a
y
b
z
c d

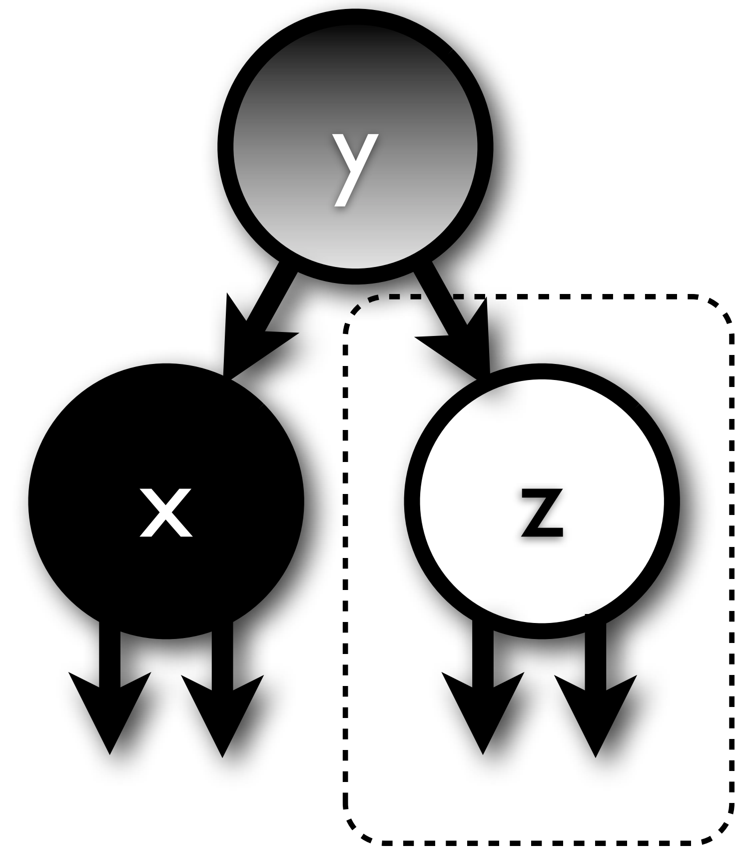

y
x
z
a
b
c
d

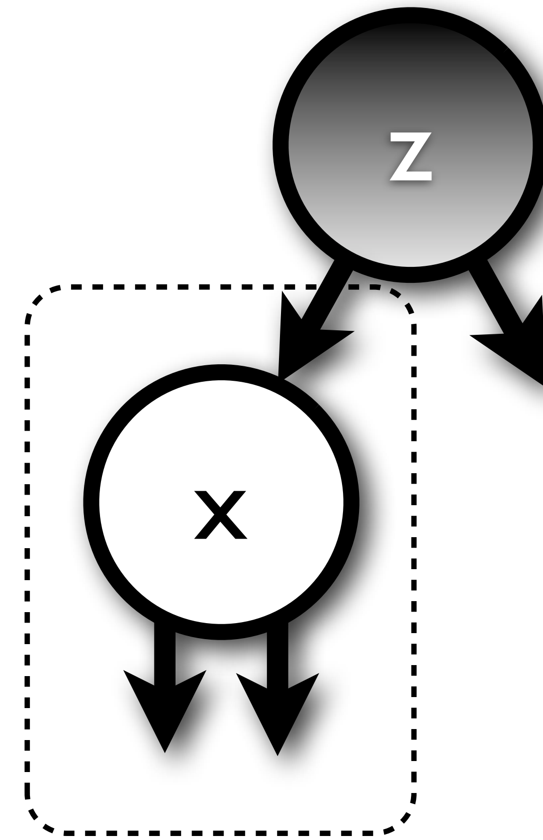
```
(define (balance-node node)
  (match node
    [(or (B    (R (R a x b) y c) z d)
         (B    (R a x (R b y c)) z d)
         (B    a x (R (R b y c) z d))
         (B    a x (R b y (R c z d))))
         ; =>
         (R                (B a x b) y (B c z d))]

    [else     node]))
```

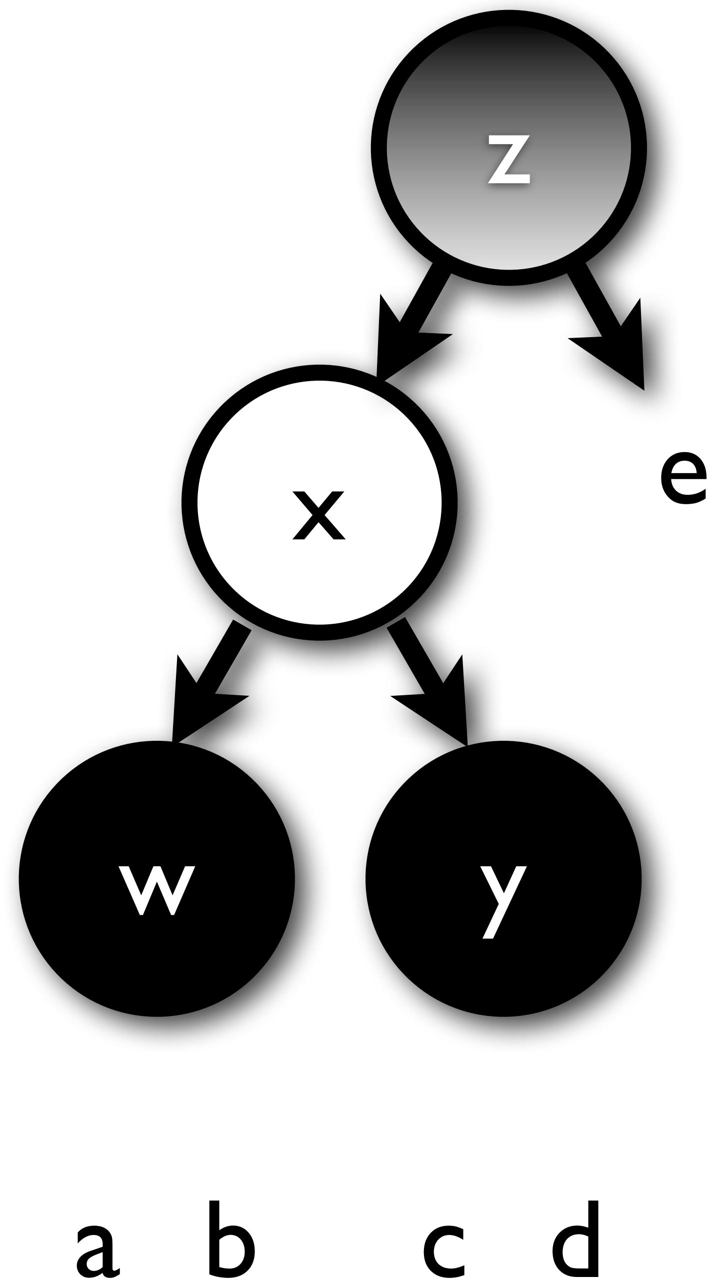
```
(define (balance-node node)
  (match node
    [(or (B/BB (R (R a x b) y c) z d)
         (B/BB (R a x (R b y c)) z d)
         (B/BB a x (R (R b y c) z d))
         (B/BB a x (R b y (R c z d))))
         ; =>
         (T (black-1 node) (B a x b) y (B c z d))]

    [else     node]))
```

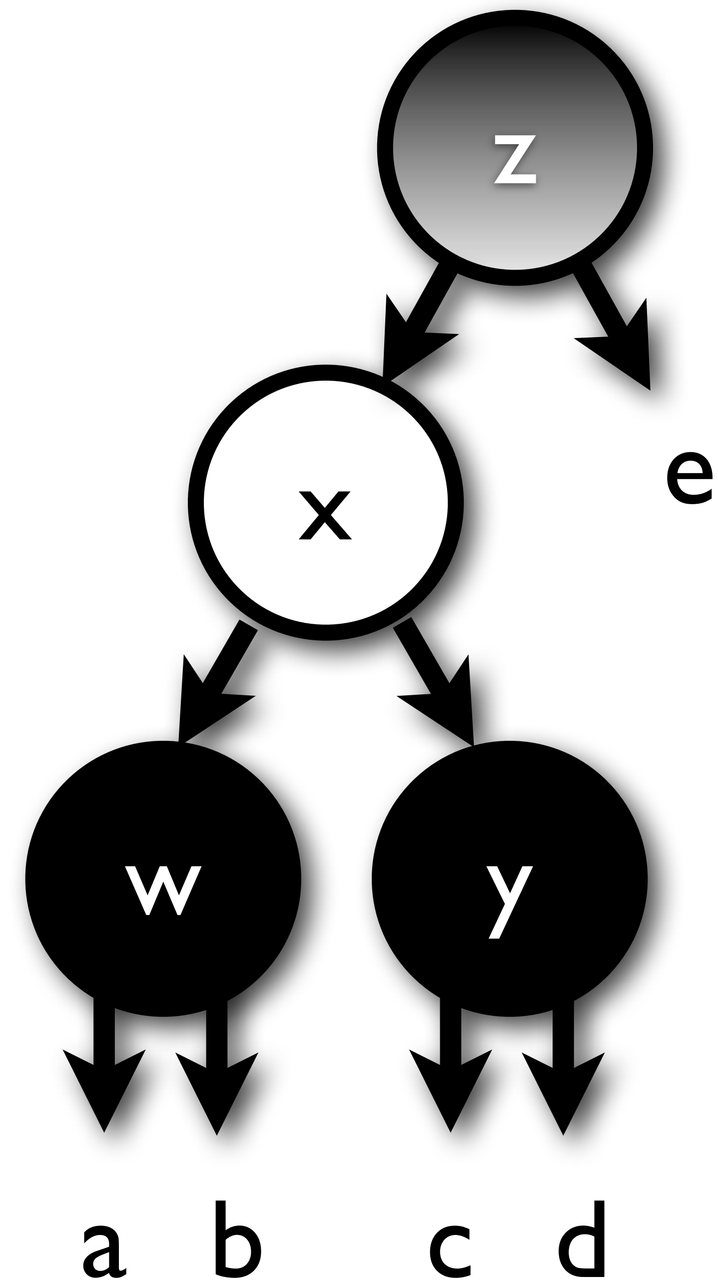

y
x
z
y
x
z
z
x
y
y
x
z
y
x
z
y
x
z

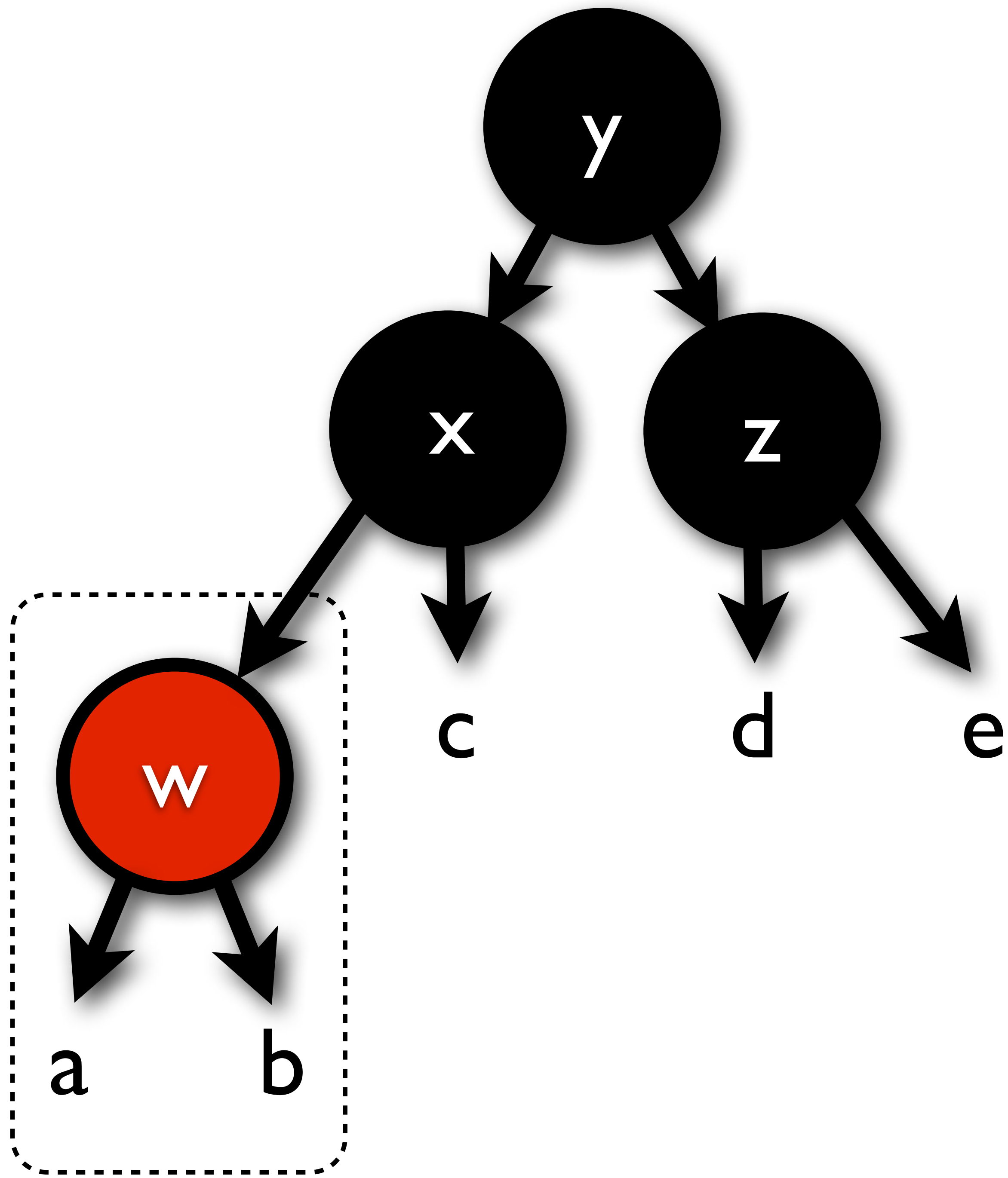

z
x
y


y
x
z


z
x


z
x
e
w
y
a b c d


z
x
e
w
y
a
b
c
d



y
x
z
w
a
b
c
d
e

```
(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?

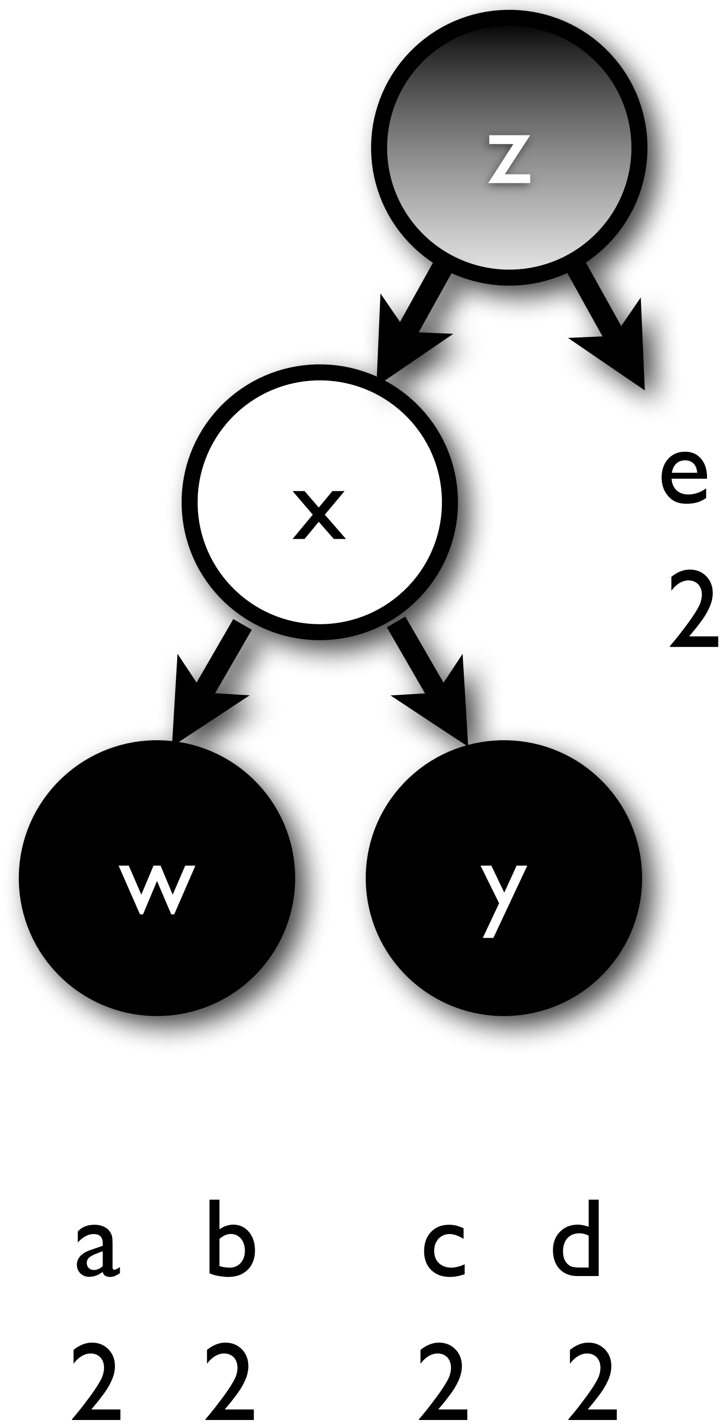
matt.might.net/articles/red-black-delete

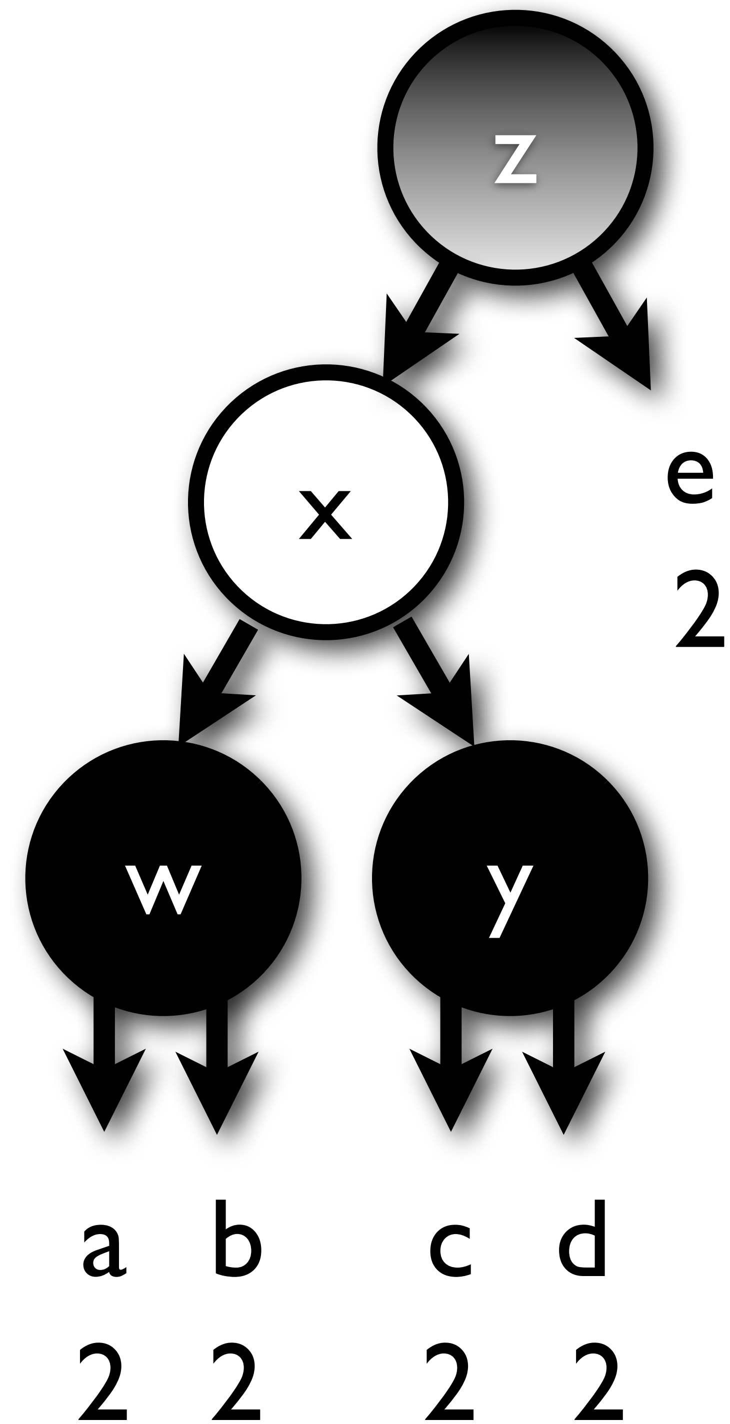
@mattmight

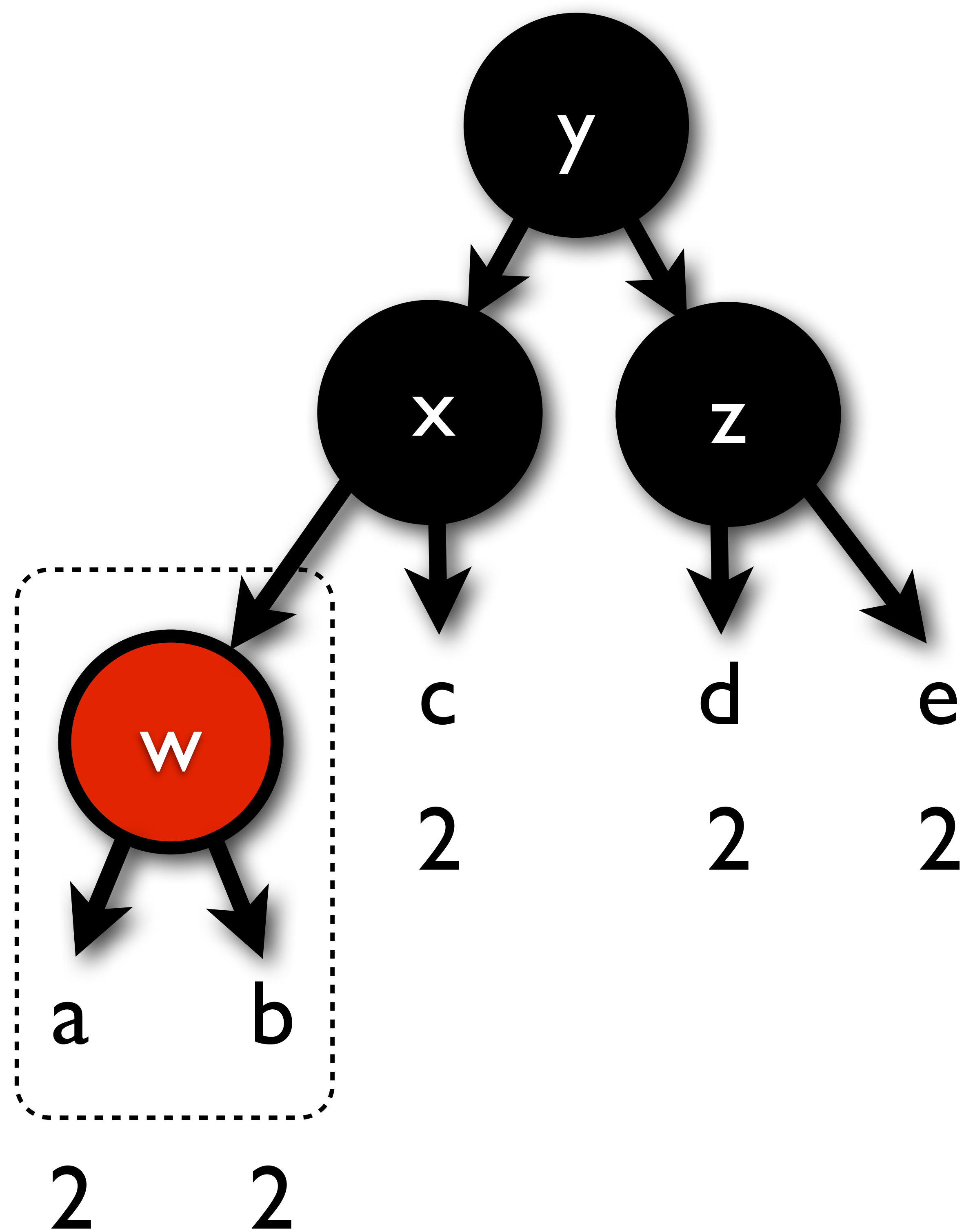
```
(define (sorted-map-delete node key)

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

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

  (         (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


z
x
e
2
w
y
a
b
c
d
2
2
2
2


z
x
e
2
w
y
a b c d
2 2 2 2


y
x
z
w
c
d
e
2
2
2
a
b
2
2