CS3500: Object-Oriented Design Spring 2014

Class 14
2.25.2014

Northeastern University
College *of* Computer and Information Science

Today...

- Assignments
- Red-Black Trees
- Refactoring
- hashCode Improvement

Assignment 7

- Red-Black Tree implementation for MyMap<K,V>
- Due: Friday, February 28, 2014 at 11:59pm

Red-Black Trees (RBT)

Red-Black Trees
[Okasaki]

Red-black trees are binary search trees in which each node has a color (either red or black) and the following balancing invariants are preserved by all operations:

1. No red node has a red child.
2. Every path from the root to an empty tree/node contains the same number of black nodes.

Benefits of RBT

- Balanced BST
- Efficiency

Theorem

In a red-black tree, the longest possible path from the root to an empty tree is no more than twice the length as the shortest possible path from the root to an empty tree.

Functional Red-Black Trees

[Okasaki]

Functional Red-Black Trees

[Okasaki]

```
data Color = R | B

data Tree elt
  = E | T Color (Tree elt) elt (Tree elt)
```

Binary Search Tree (BST)

- t is empty
- t is a node
  - a label
  - the left subtree of t is a BST,
  - the right subtree of t is a BST,
  - every label within the left subtree of t is less than the label of t,
  - every label within the right subtree of t is greater than the label of t

Binary Search Tree (BST)

```
data Tree elt
  = E | T Color (Tree elt) elt (Tree elt)
```

- t is empty
- t is a node
  - a label
  - the left subtree of t is a BST,
  - the right subtree of t is a BST,
  - every label within the left subtree of t is less than the label of t,
  - every label within the right subtree of t is greater than the label of t

Binary Search Tree (BST)

```
data Tree elt
  = E | T Color (Tree elt) elt (Tree elt)

BST.emptyTree(c)     BST.node(c, s, t1, t2)
```

member/contains

```
member x E = False
member x (T _ a y b) | x <  y = member x a
                     | x == y = True
                     | x >  y = member x b
```

```
BST.emptyTree(c).contains(s1)  =  false
BST.node(c, s, t1, t2).contains(s1)
  =  true              if c.compare(s, s1) == 0
BST.node(c, s, t1, t2).contains(s1)
  =  t1.contains(s1)   if c.compare(s, s1) > 0
BST.node(c, s, t1, t2).contains(s1)
  =  t2.contains(s1)   if c.compare(s, s1) < 0
```

Insertions
[Okasaki]

```
insert :: Ord elt => elt -> Set elt ->
          Set elt insert x s = makeBlack (ins s)
  where ins E = T R E x E
        ins(T color a y b)| x < y  = balance color (ins a) y b
                          | x == y = T color a y b
                          | x > y  = balance color a y (ins b)
        makeBlack (T _ a y b) = T B a y b
```

Insertions

```
insert :: Ord elt => elt -> Set elt ->
          Set elt insert x s = makeBlack (ins s)
  where ins E = T R E x E
        ins(T color a y b)| x < y  = balance color (ins a) y b
                          | x == y = T color a y b
                          | x > y  = balance color a y (ins b)
        makeBlack (T _ a y b) = T B a y b
```

```
BST.emptyTree(c).insert(s1)
  =  BST.node(c, s1, BST.emptyTree(c), BST.emptyTree(c))
BST.node(c, s, t1, t2).insert(s1)
  = BST.node(c, s, t1, t2)             if c.compare(s, s1) == 0
BST.node(c, s, t1, t2).insert(s1)
  = BST.node(c, s, t1.insert(s1), t2) if c.compare(s, s1) > 0
BST.node(c, s, t1, t2).insert(s1)
  = BST.node(c, s, t1, t2.insert(s1)) if c.compare(s, s1) < 0
```


Northeastern University
College *of* Computer and Information Science

Balance
[Okasaki]

```
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 color a x b = T color a x b
```

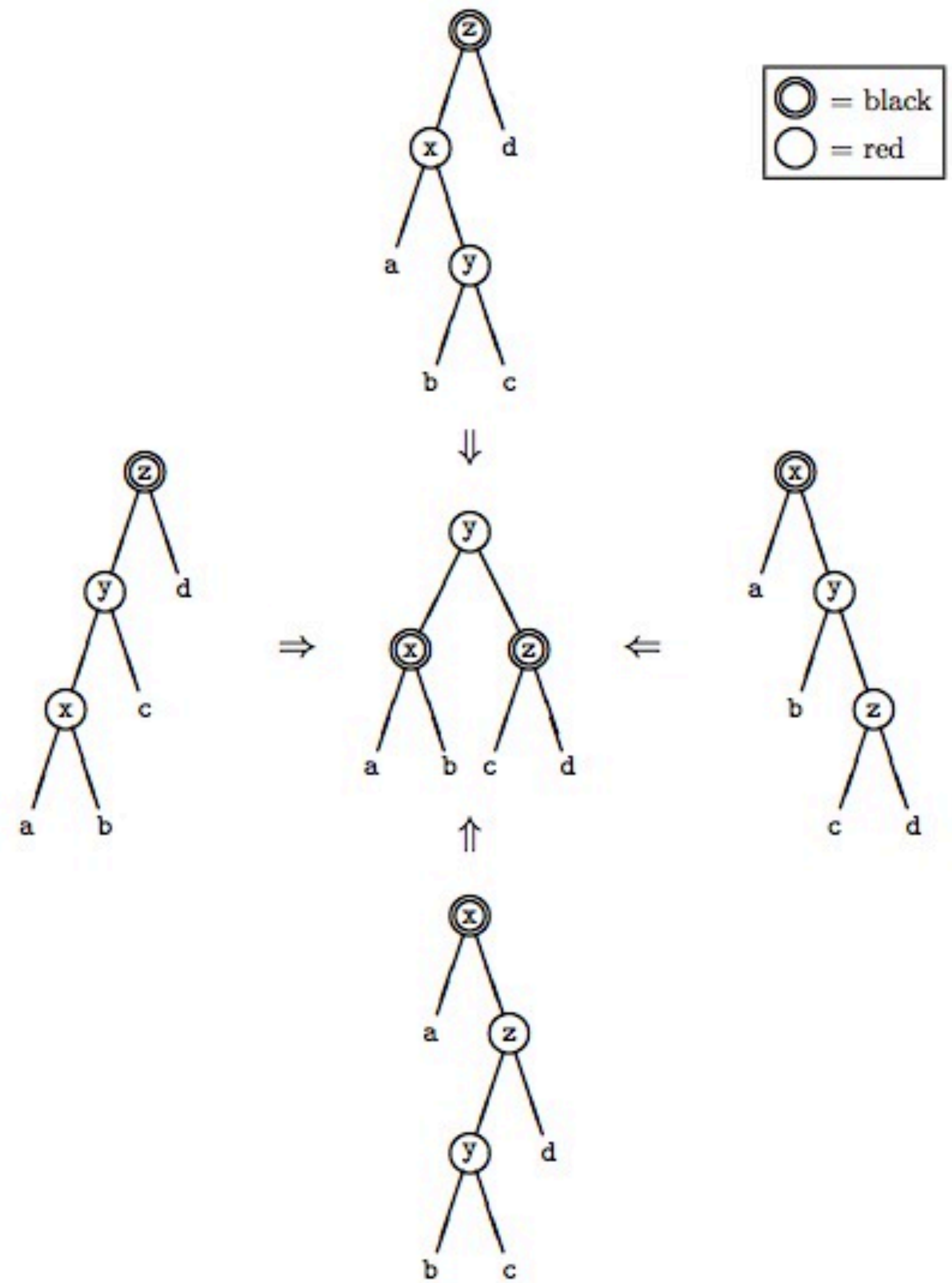
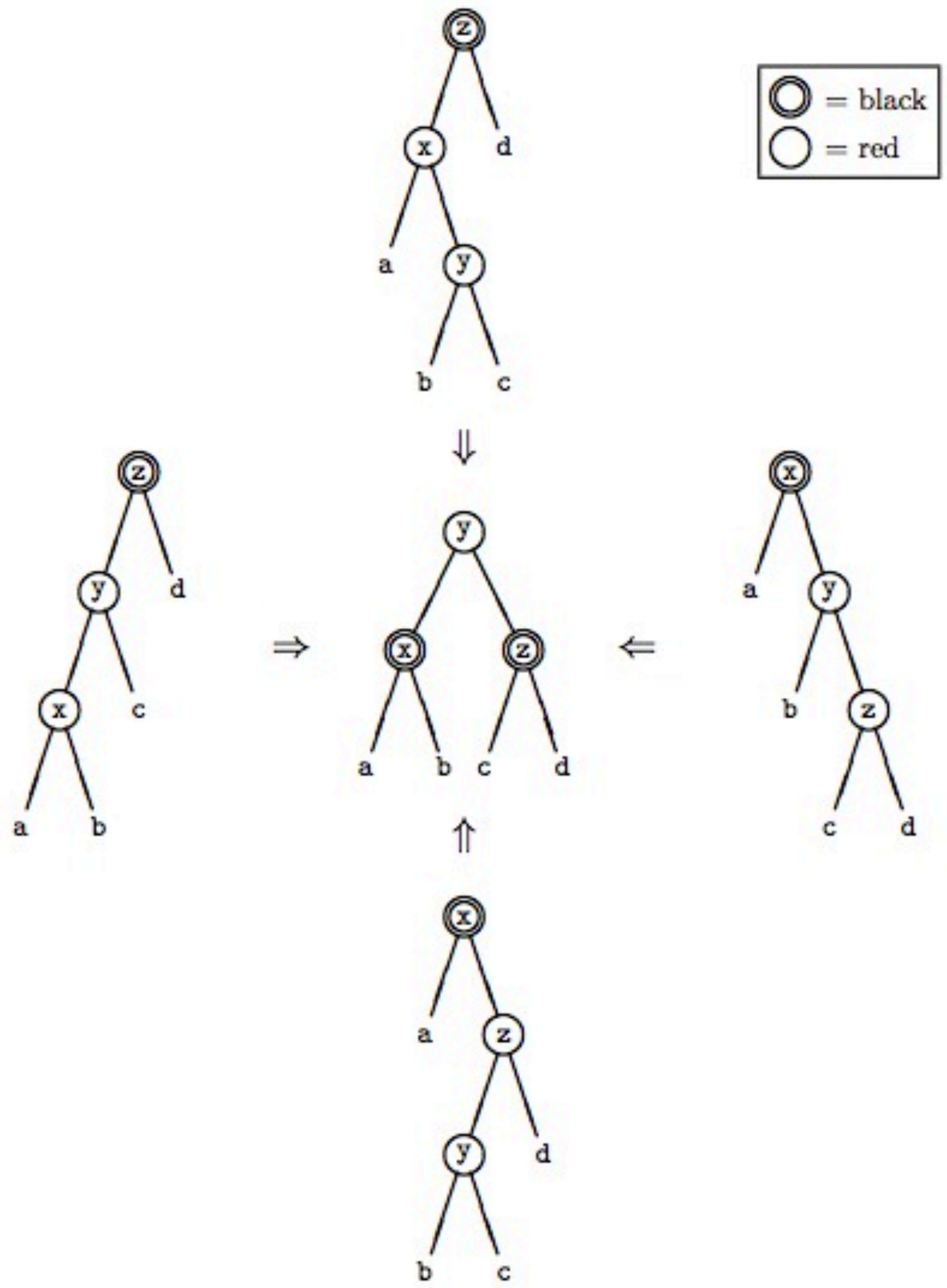
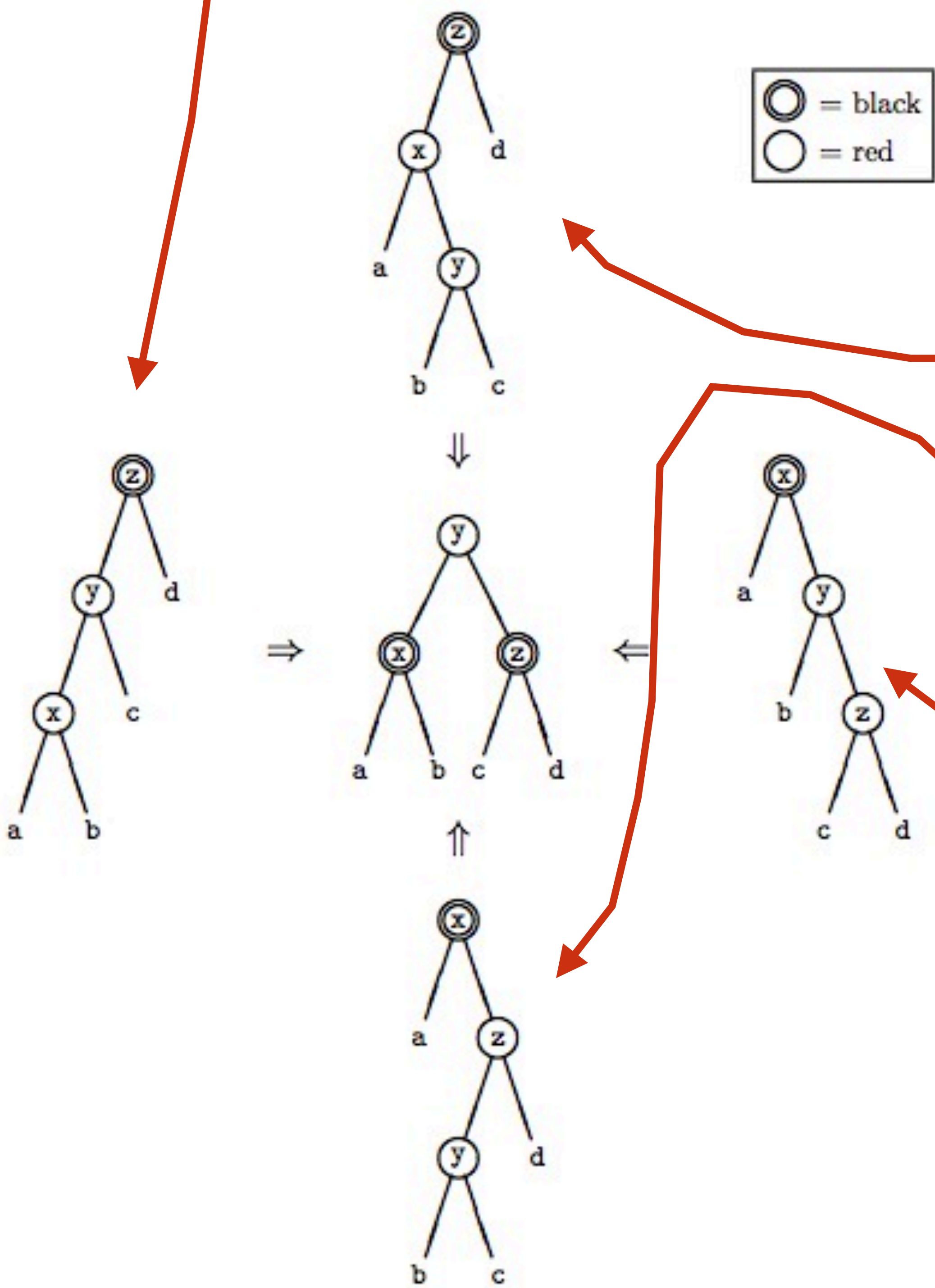


Figure from [Okasaki]




```
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 color a x b = T color a x b
```

from [Okasaki]



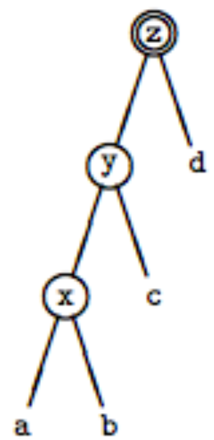

```
balance B (T R (T R a x b) y c) z d
    = T R (T B a x b) y (T B c z d)

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 color a x b = T color a x b
```

from [Okasaki]

ICE

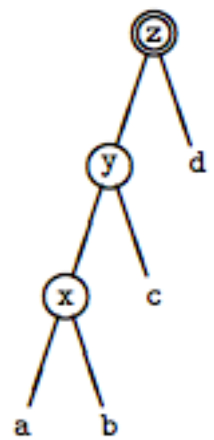
MyMap<String, Integer> with StringByLex

- <Jack, 10>
- <Ellie, 5>
- <Dan, 4>
- <Ben, 2>
- <Carol, 3>
- <Oliver, 15>
- <Tim, 20>
- <Yvette, 25>
- <Victor, 22>




```
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)
```

from [Okasaki]




```
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)
```

```
a x b: left left
y: left data
c: left right
z: data
d: right
```

from [Okasaki]

```
if (isBlack() &&
    !(left.isEmpty()) &&
    !(((Node) left).left.isEmpty()) &&
    (left.color == RED) &&
    (((Node) left).left.color == RED))
```

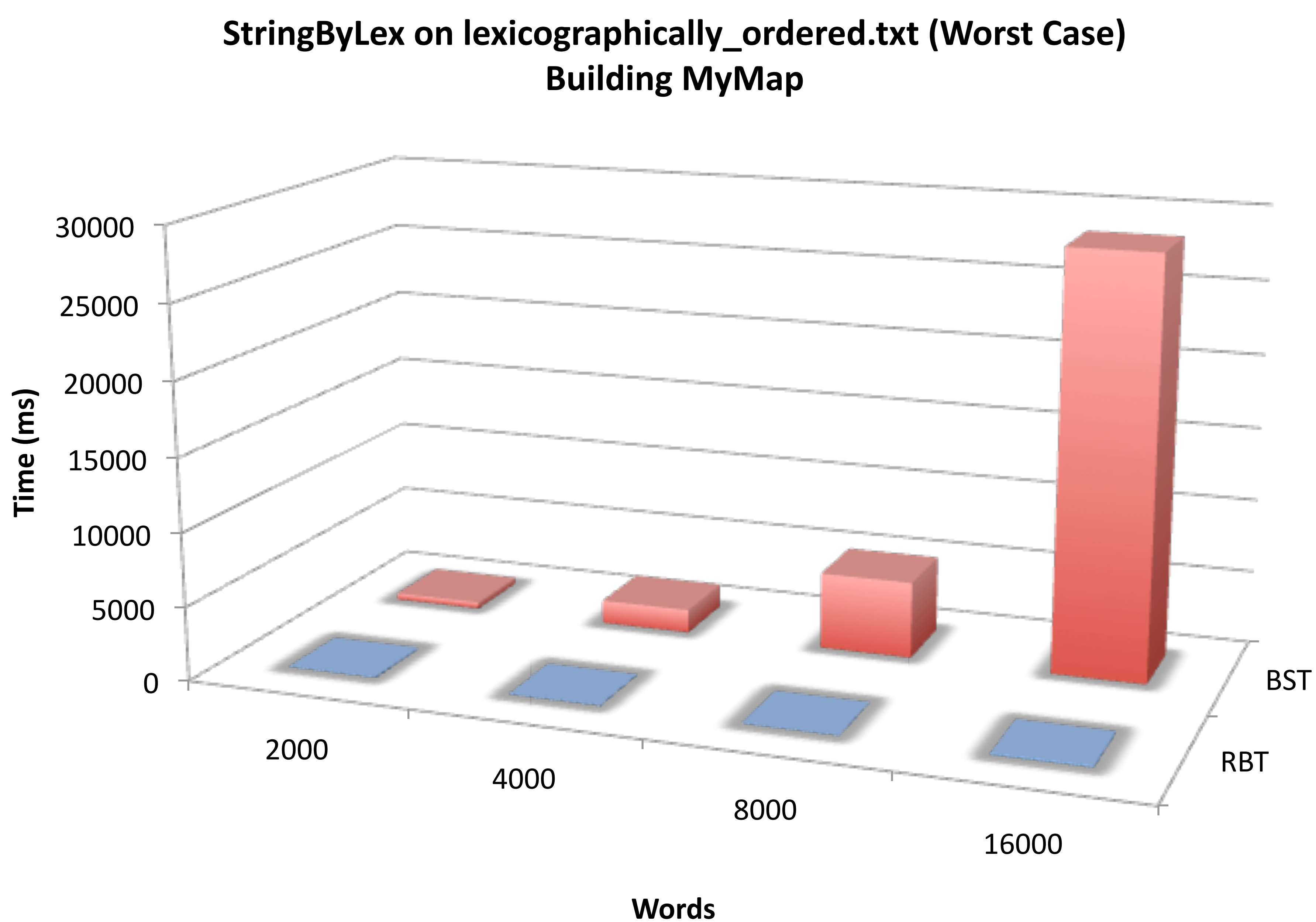
```
if (isBlack() &&
    !(left.isEmpty()) &&
    !(((Node) left).left.isEmpty()) &&
    (left.color == RED) &&
    (((Node) left).left.color == RED))
```

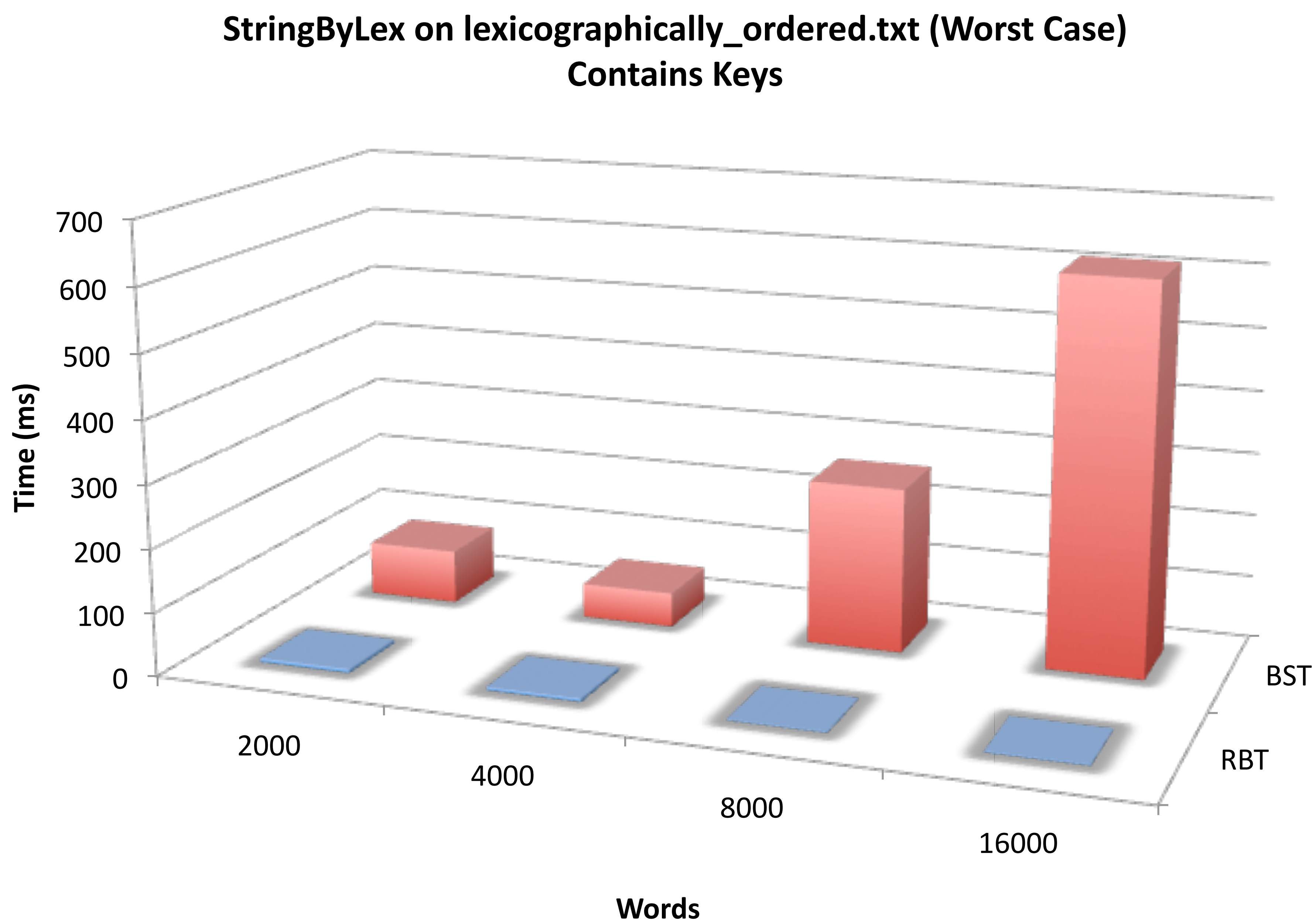
Law of Demeter

Translating into Java

Maintaining Invariants

My Implementations


StringByLex on lexicographically_ordered.txt (Worst Case)
Building MyMap
Time (ms)
30000
25000
20000
15000
10000
5000
0
2000
4000
8000
16000
Words
BST
RBT


StringByLex on lexicographically_ordered.txt (Worst Case)
Contains Keys
Time (ms)
700
600
500
400
300
200
100
0
2000
4000
8000
16000
Words
BST
RBT

Other Approaches to Red-Black Trees

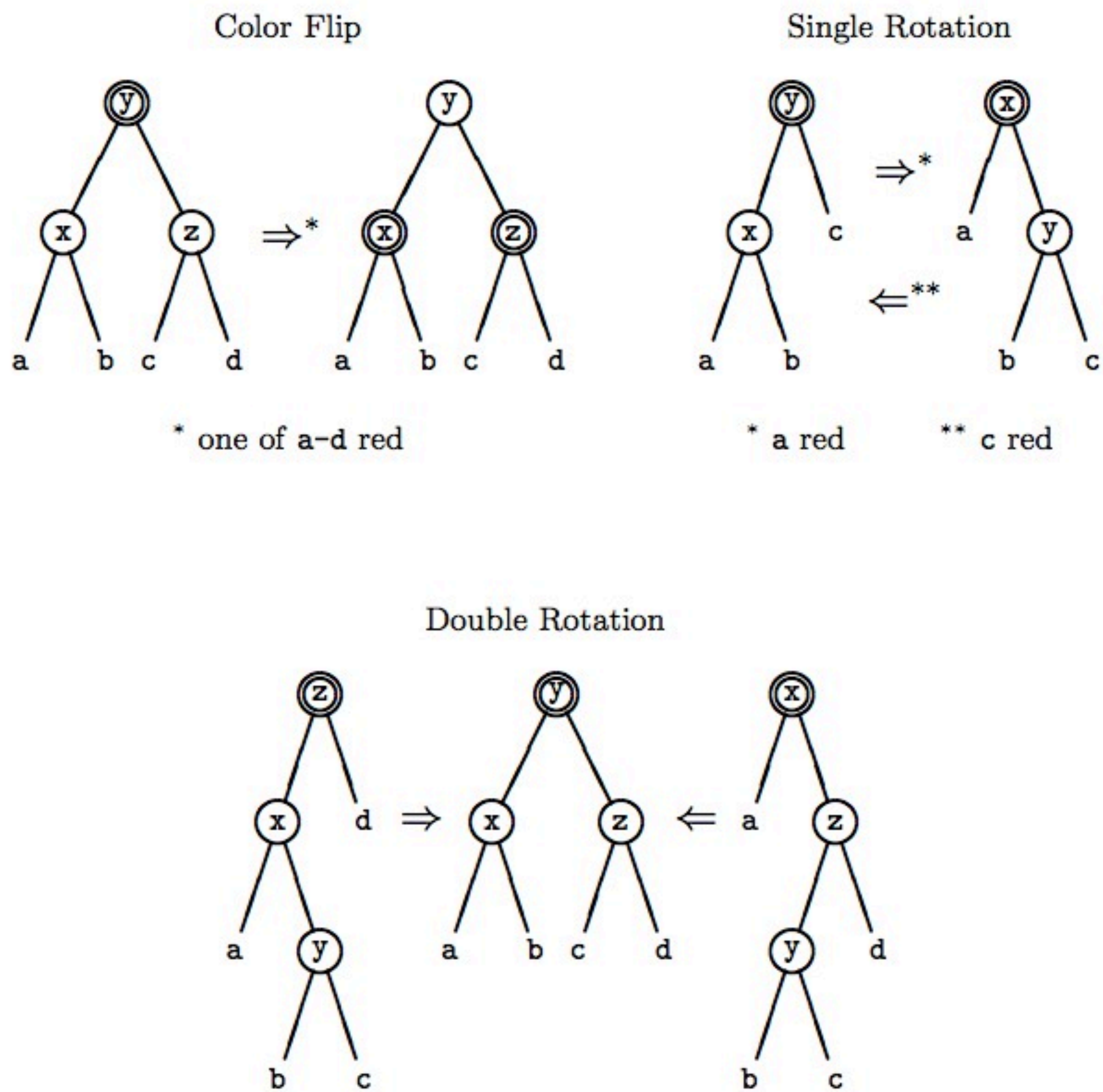


Fig. 2. Alternative balancing transformations. Subtrees a–d all have black roots unless otherwise indicated.

Figure from [Okasaki]

```
-- color flips
balance B (T R a@(T R _ _ _) x b) y (T R c z d)
     || B (T R a x b@(T R _ _ _)) y (T R c z d)
     || B (T R a x b) y (T R c@(T R _ _ _) z d)
     || B (T R a x b) y (T R c z d@(T R _ _ _)) = T R (T B a x b) y (T B c z d
-- single rotations
balance B (T R a@(T R _ _ _) x b) y c = T B a x (T R b y c)
balance B a x (T R b y c@(T R _ _ _)) = T B (T R a x b) y c
-- double rotations
balance B (T R a x (T R b y c)) z d
     || B a x (T R (T R b y c) z d) = T B (T R a x b) y (T R c z d)
-- no balancing necessary
balance color a x b = T color a x b
```

Fig. 3. Alternative implementation of `balance`.

Figure from [Okasaki]

Mutable Binary Search Trees [CLR01]

To insert a new value v into a binary search tree T, we use the procedure TREE-INSERT. The procedure takes a node z for which $z.key = v$, $z.left = \text{NIL}$, and $z.right = \text{NIL}$. It modifies T and some of the attributes of z in such a way that it inserts z into an appropriate position in the tree.

```
TREE-INSERT(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == NIL
      T.root = z          // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
```

[CLR01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. MIT Press and McGraw-Hill, 2nd edition edition, 2001.

Red-Black Tree
[CLR01]

A red-black tree is a binary tree that satisfies the following ***red-black properties***:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Rotation
[CLR01]

```
LEFT-ROTATE(T, x)
y = x.right               // set y
x.right = y.left          // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                 // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                // put x on y's left
x.p = y
```

Insert into RBT
[CLR01]

We can insert a node into an n-node red-black tree in $O(\lg n)$ time. To do so, we use a slightly modified version of the TREE-INSERT procedure (Section 12.3) to insert node z into the tree T as if it were an ordinary binary search tree, and then we color z red. (Exercise 13.3-1 asks you to explain why we choose to make node z red rather than black.) To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations. The call RB-INSERT(T, z) inserts node z, whose *key* is assumed to have already been filled in, into the red-black tree T.

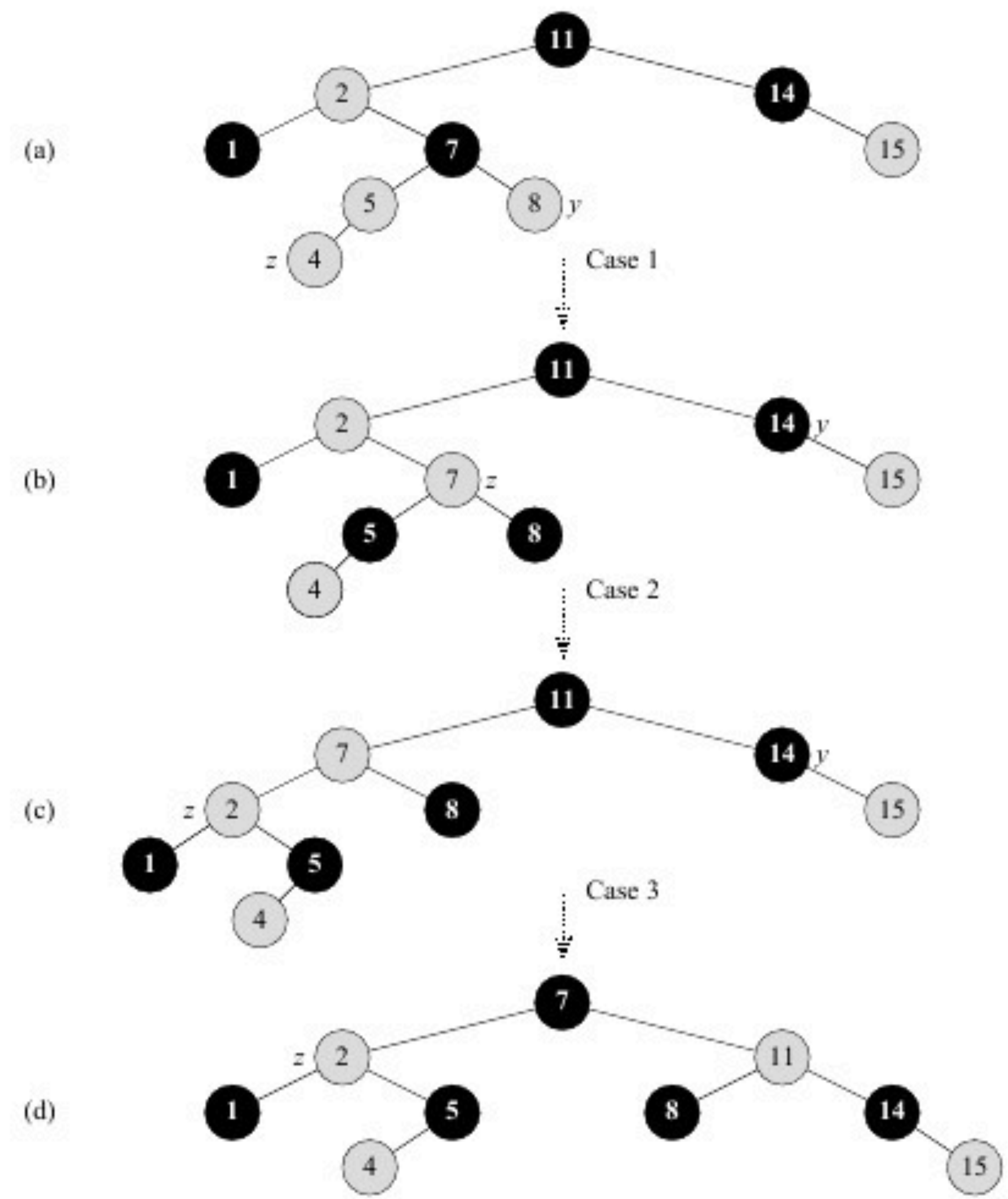
```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == T.nil
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

The procedures TREE-INSERT and RB-INSERT differ in four ways. First, all instances of NIL in TREE-INSERT are replaced by *T.nil*. Second, we set *z.left* and *z.right* to *T.nil* in lines 14–15 of RB-INSERT, in order to maintain the proper tree structure. Third, we color z red in line 16. Fourth, because coloring z red may cause a violation of one of the red-black properties, we call RB-INSERT-FIXUP(T, z) in line 17 of RB-INSERT to restore the red-black properties.

When are rotations needed? [CLR01]

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                       // case 1
            y.color = BLACK                         // case 1
            z.p.p.color = RED                       // case 1
            z = z.p.p                               // case 1
        else if z == z.p.right
                z = z.p                             // case 2
                LEFT-ROTATE(T, z)                   // case 2
            z.p.color = BLACK                       // case 3
            z.p.p.color = RED                       // case 3
            RIGHT-ROTATE(T, z.p.p)                  // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

[CLR01]


(a)
11
2
14
1
7
15
5
8
y
z
4
Case 1
(b)
11
2
14
y
1
7
z
15
5
8
4
Case 2
(c)
11
7
14
y
z
2
8
15
1
5
4
Case 3
(d)
7
z
2
11
1
5
8
14
4
15