Red-Black Trees

Red-Black Tree

Wholeness of the Lesson

- Red-black trees provide a solution to the problem of unacceptably slow worst case performance of binary search trees. This is accomplished by introducing a new element: nodes of the tree are colored red or black, adhering to the balance condition for red-black trees. The balance condition is maintained

Red-Black Tree

during insertions and deletions and doing so introduces only slight overhead.

- **Science of Consciousness: Red-black trees, as an example of BSTs** with a balance condition, exhibit the Principle of the Second Element for solving the problem of skewed BSTs.

Review: Binary Search Trees

- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, eleements have:
  - *key*: an identifying field inducing a total ordering
  - *left*: pointer to a left child (may be NULL)
  - *right*: pointer to a right child (may be NULL)
  - *p*: pointer to a parent node (NULL for root)

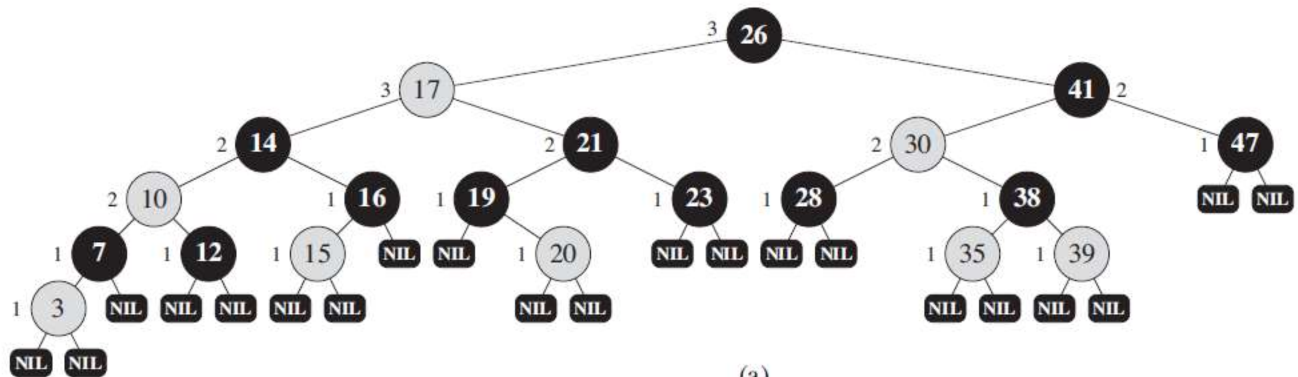
Red-Black Trees

- *Red-black trees*:
  - Binary search trees augmented with node color
  - Operations designed to guarantee that the height $h = O(\lg n)$
- First: describe the properties of red-black trees
- Then: prove that these guarantee $h = O(\lg n)$
- Finally: describe operations on red-black trees

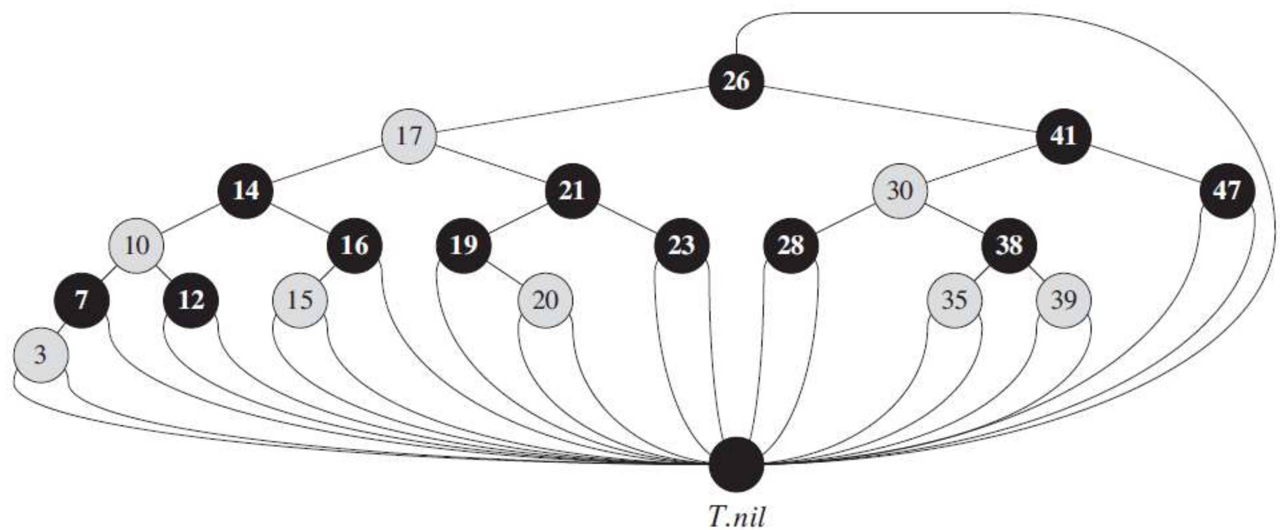
Red-Black Properties

- The *red-black properties*:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
     - Note: this means every “real” node has 2 children
  3. If a node is red, both children are black
     - Note: can’t have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

Red-Black Tree Example

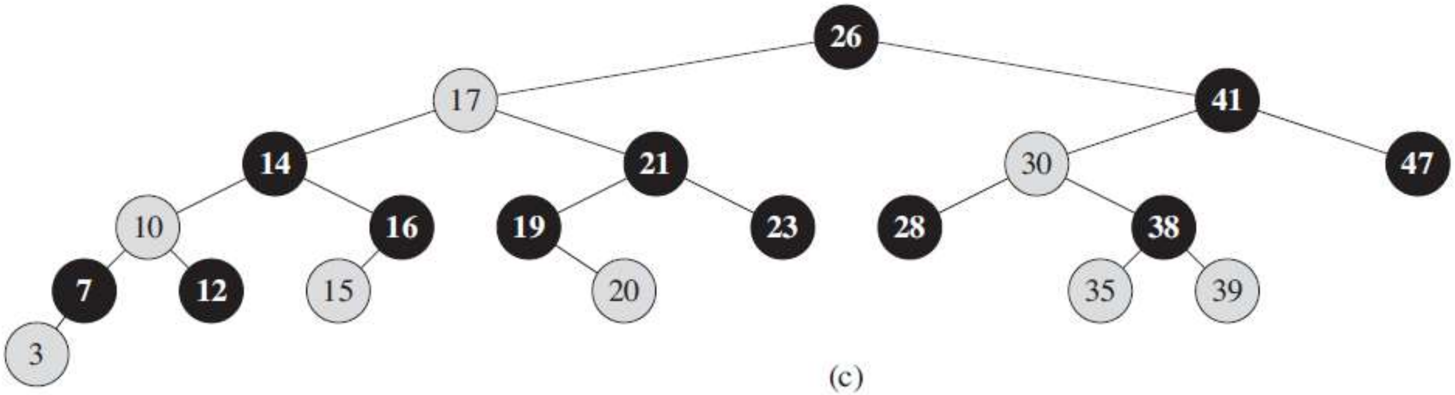


(a)



(b)

Red-Black Tree Example



(c)

Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *T.nil*, which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. We shall use this drawing style in the remainder of this chapter.

Height of Red-Black Trees

- *What is the minimum black-height of a node with height h?*
- A: a height-h node has black-height $\geq h/2$
- Theorem: A red-black tree with n internal nodes has height $h \leq 2 \lg(n + 1)$
- *How do you suppose we'll prove this?*

Red-Black Properties: Some Proof

A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

Proof We start by showing that the subtree rooted at any node x contains at least $2^{\mathrm{bh}(x)} - 1$ internal nodes. We prove this claim by induction on the height of x. If the height of x is 0, then x must be a leaf ($T.nil$), and the subtree rooted at x indeed contains at least $2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$ internal nodes. For the inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black-height of either $\mathrm{bh}(x)$ or $\mathrm{bh}(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{\mathrm{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{\mathrm{bh}(x)-1} - 1) + (2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ internal nodes, which proves the claim.

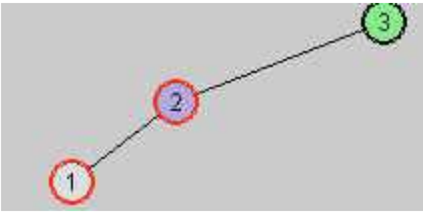
Red-Black Properties

- The *Black Height must be at least h/2. Hence,*

$$n \geq 2^{h/2} - 1 .$$

Moving the 1 to the left-hand side and taking logarithms on both sides yields $\lg(n+1) \geq h/2$, or $h \leq 2\lg(n+1)$. ∎

Red-Black Tree: Some Key Cases (Rotation – for balance)



→

More on Rotation

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
```

More on Rotation

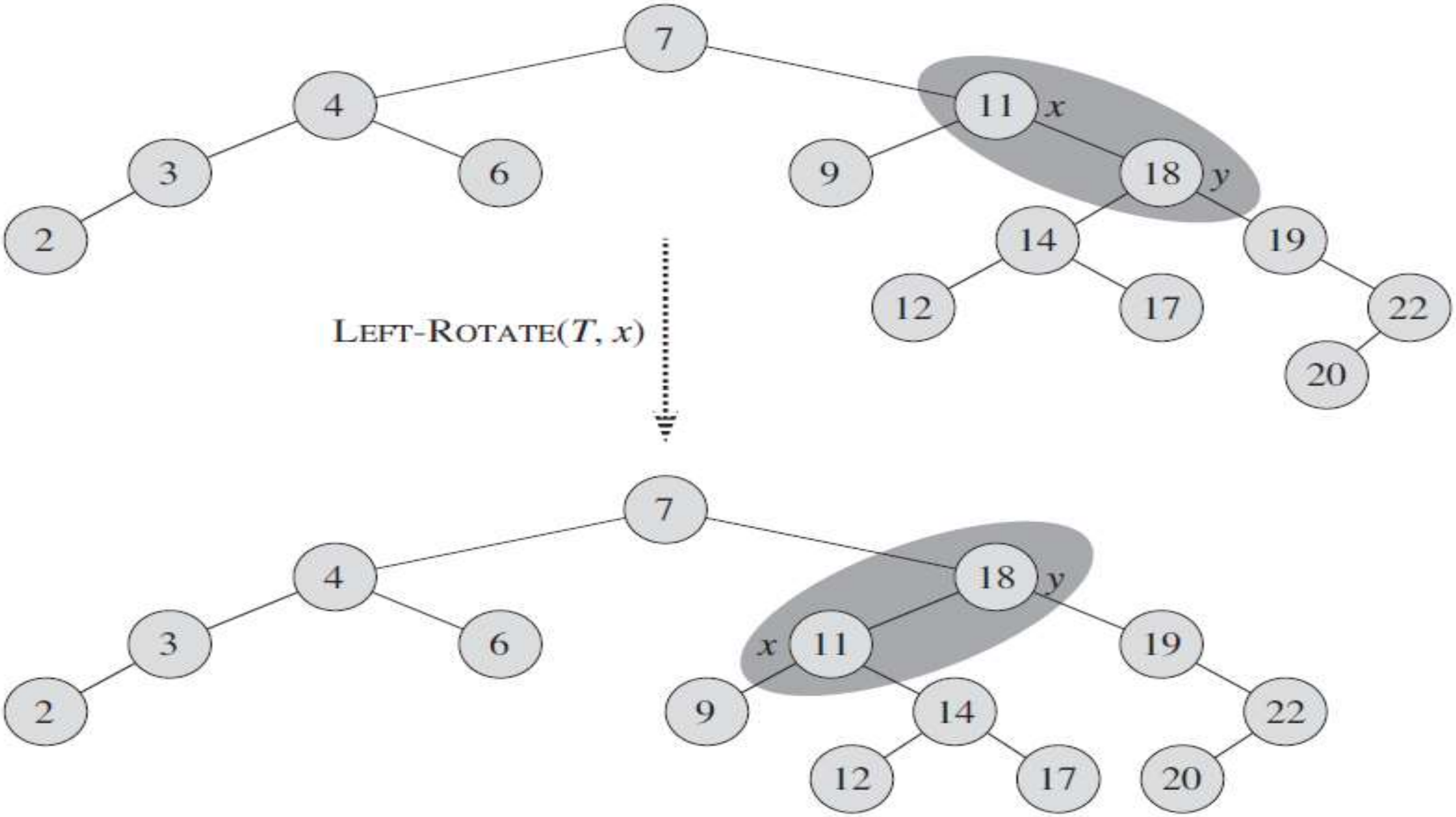


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

RB Insert

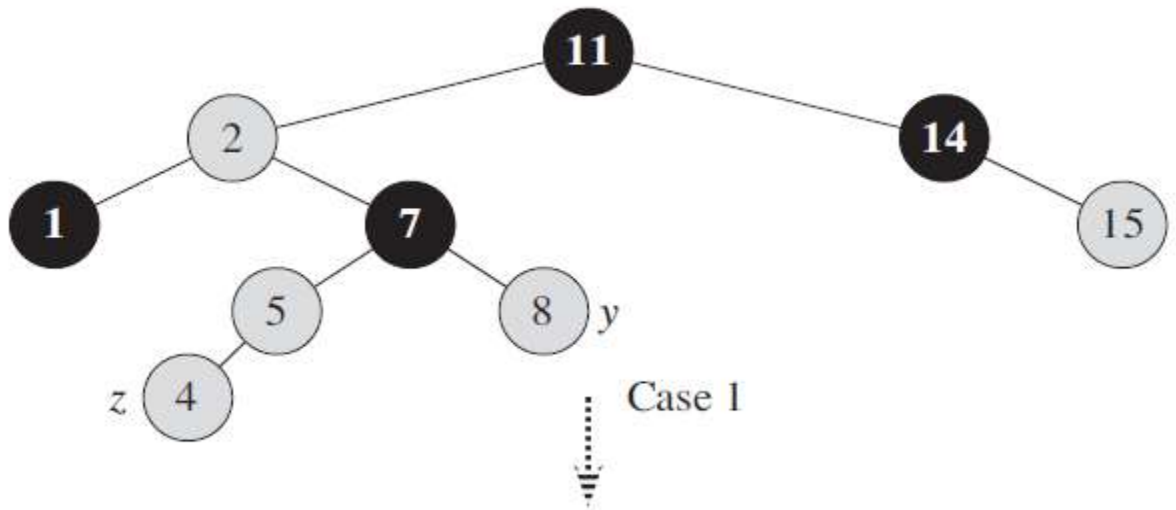
```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
        y = x
        if z.key < x.key
                x = x.left
        else x = x.right
z.p = y
if y == T.nil
        T.root = z
elseif z.key < y.key
        y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

RB Insert

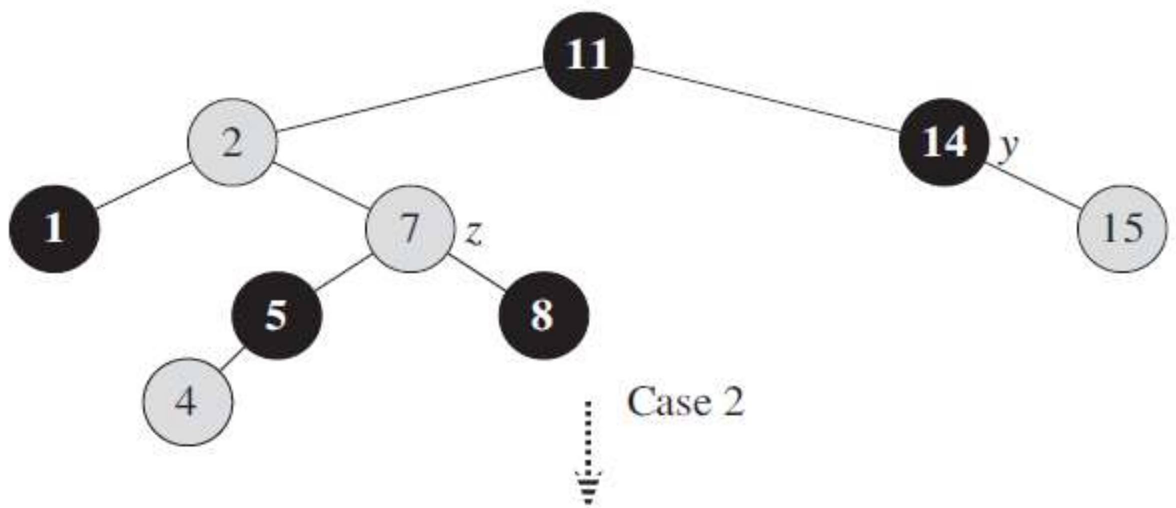
```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                    // case 1
            y.color = BLACK                      // case 1
            z.p.p.color = RED                    // case 1
            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
```

RB Insert

(a)

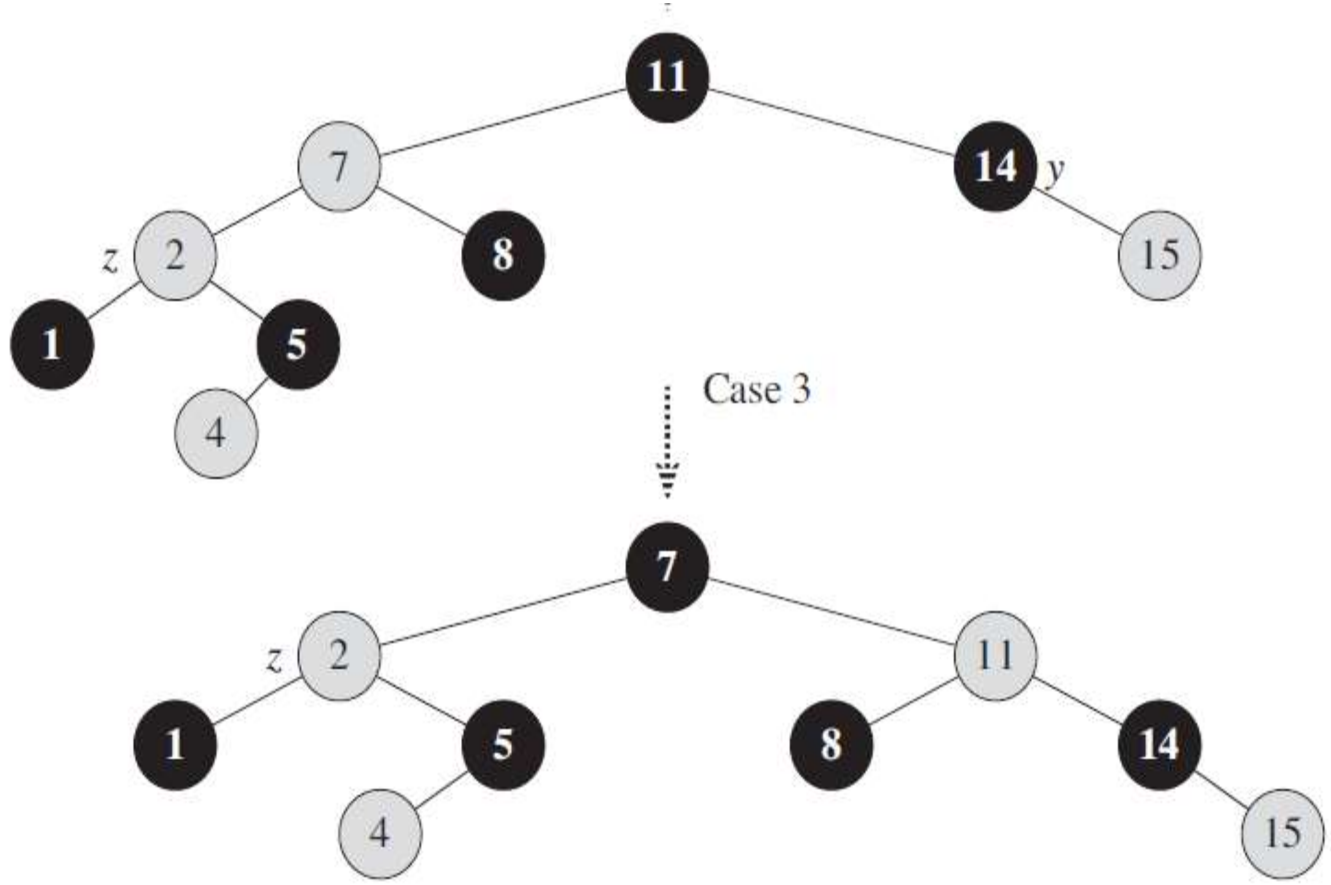


(b)



RB Insert

(c)



(d)

RB Insert

(d)

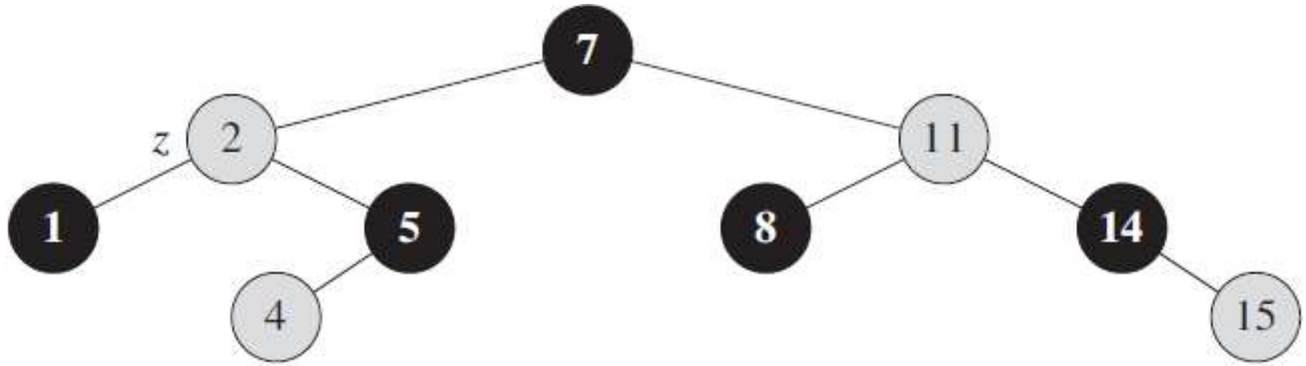


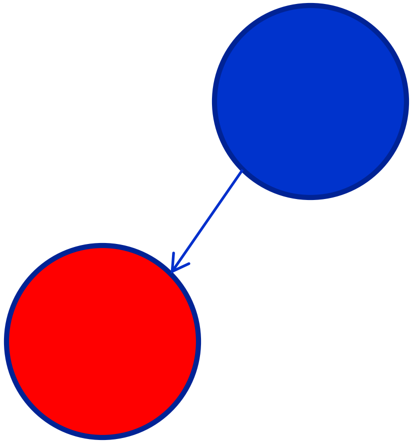
Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. We recolor nodes and move the pointer z up the tree, resulting in the tree shown in **(b)**. Once again, z and its parent are both red, but z's uncle y is black. Since z is the right child of $z.p$, case 2 applies. We perform a left rotation, and the tree that results is shown in **(c)**. Now, z is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in **(d)**, which is a legal red-black tree.

RB Insert (Repeated for Conveni.)

```
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
```

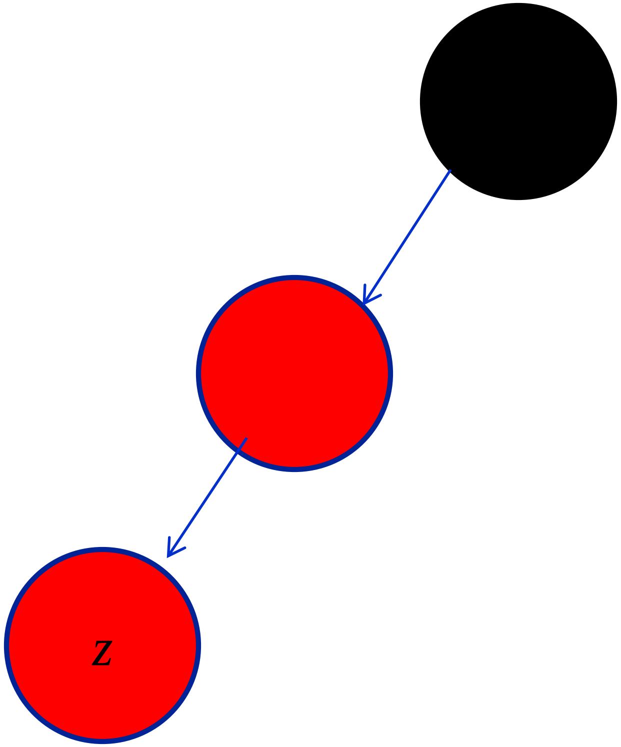
Red-Black Insertion

Code – No Case



Red-Black Insertion

Code – Case 3



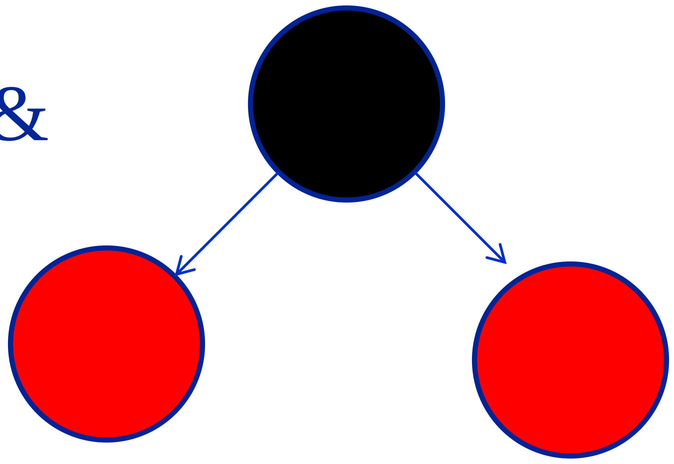
[Right Rotate]

Red-Black Insertion

Code – Case 3

-> Middle node becomes root & black

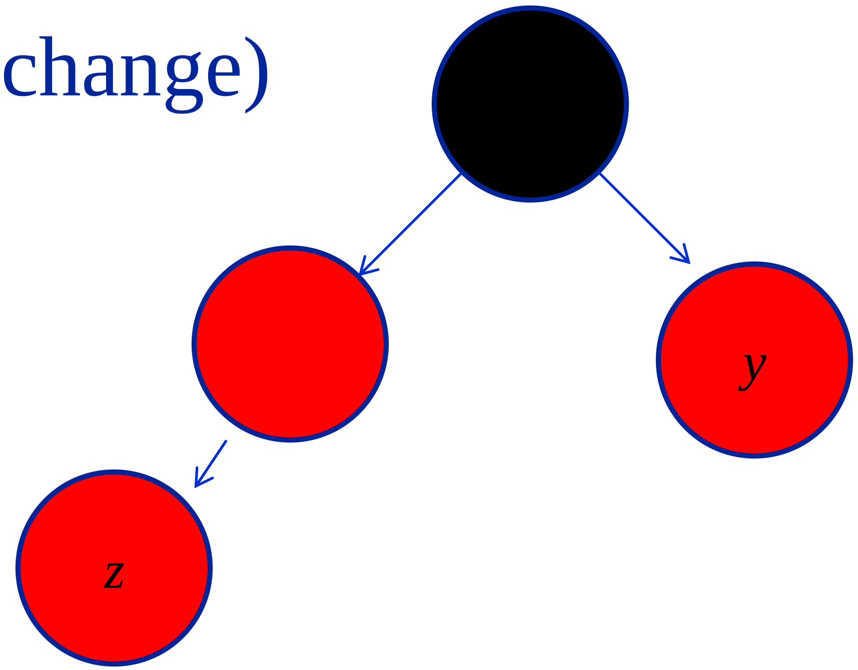
-> Root node becomes right node & red



Red-Black Insertion

Code – Case 1

(Just color change)



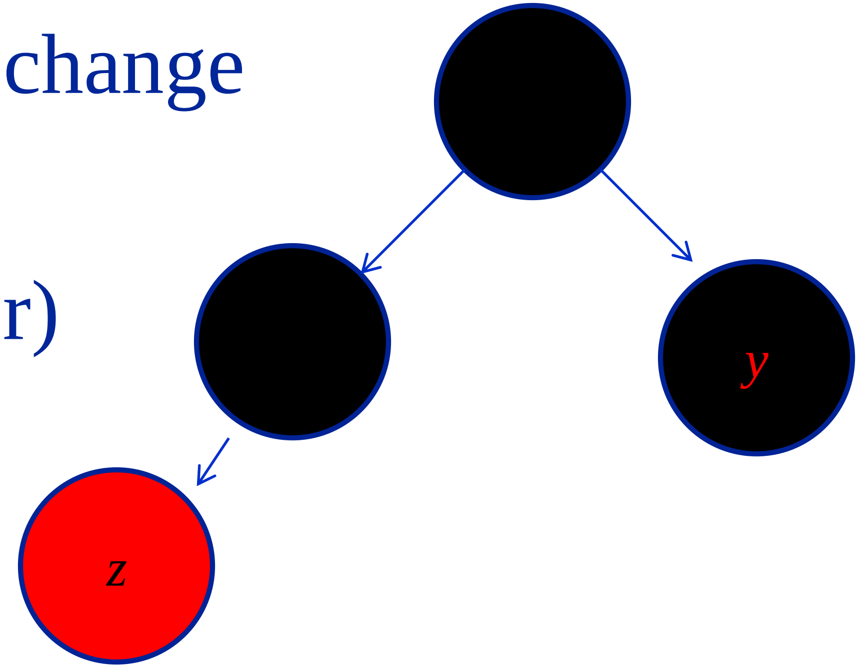
Red-Black Insertion

Code – Case 1

(Just color change

Including

T.root.color)



Red-Black Trees: Key Comments

- *So, the trick is to use Red and Back labels to nodes and use appropriate rules to manipulate nodes in such a way that makes the tree balanced after inserting a new value.*

- *The same is true for Deletion (we will cover it in lab problems).*

RB Trees: Worst-Case Time

- We've proved that a red-black tree has O(lg *n*) height
- Corollary: These operations take O(lg *n*) time:
  - Minimum(), Maximum()
  - Successor(), Predecessor()
  - Search()
- Insert() and Delete():
  - Will also take O(lg *n*) time
  - But will need special care since they modify tree