Red-Black Trees

Prem Nair

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.

Binary Search Trees

- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, elements 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)

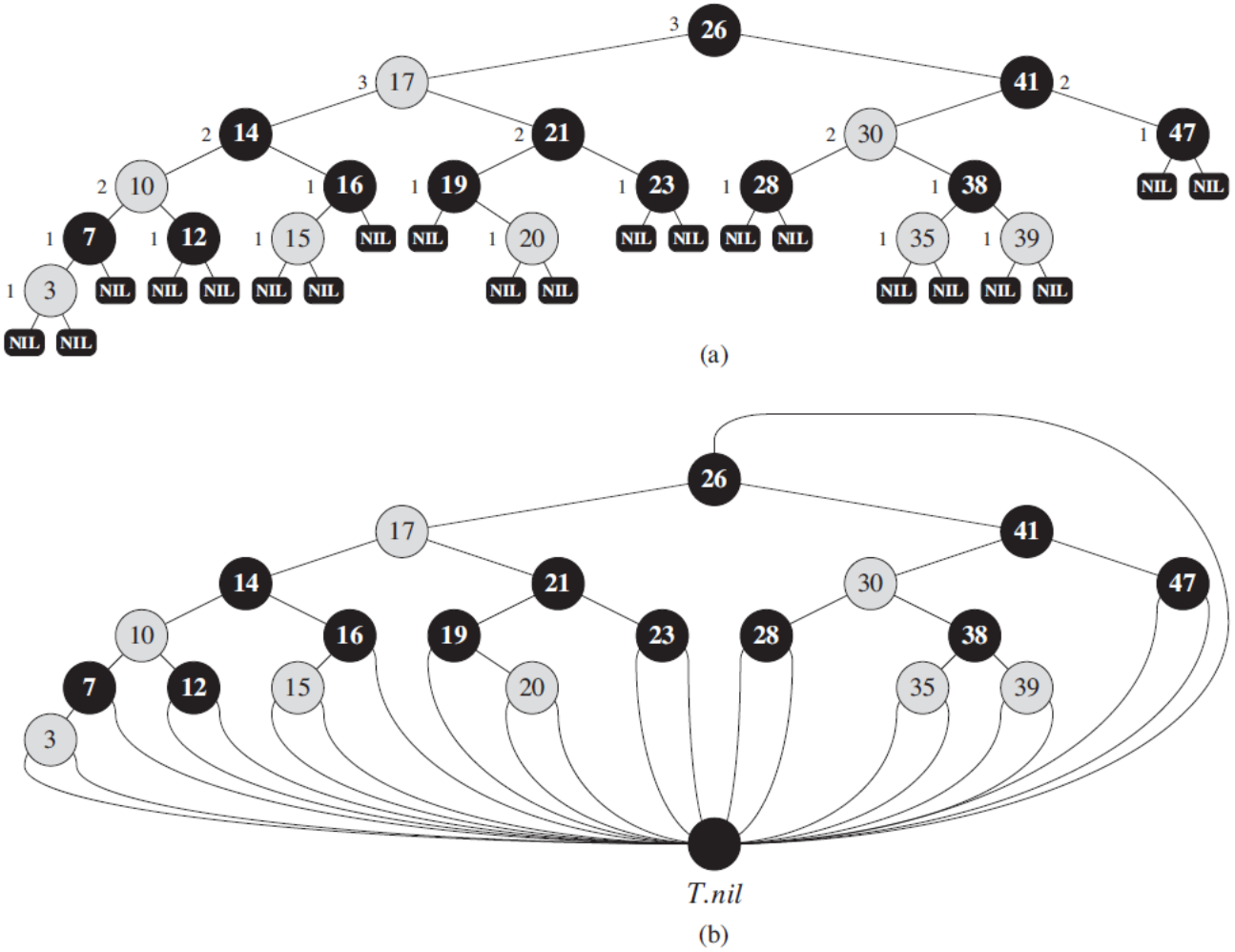
Red-Black Trees

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

5 Properties of a Red-Black Tree

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 a "real" node to descendent leaf contains the same number of black nodes.
   - Note: The number of black nodes on the path **excluding the "real" node** **but** **including the leaf** is the **black-height**. **Black-height of leaf is 0.**
5. The root is always black

Red-Black Tree Example



Red-Black Tree Example

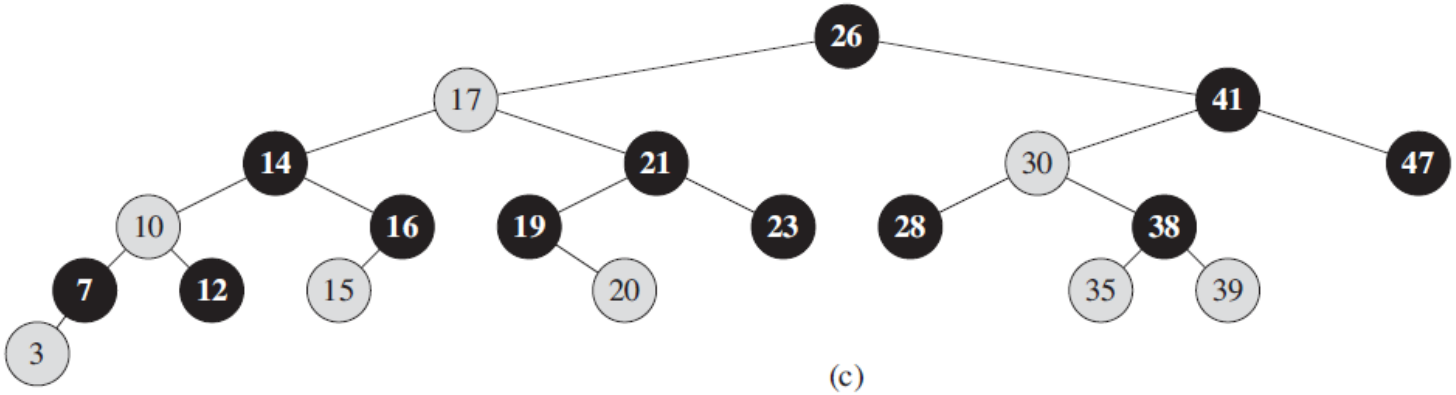


Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a 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 $h \leq 2\lg(n+1)$. ■

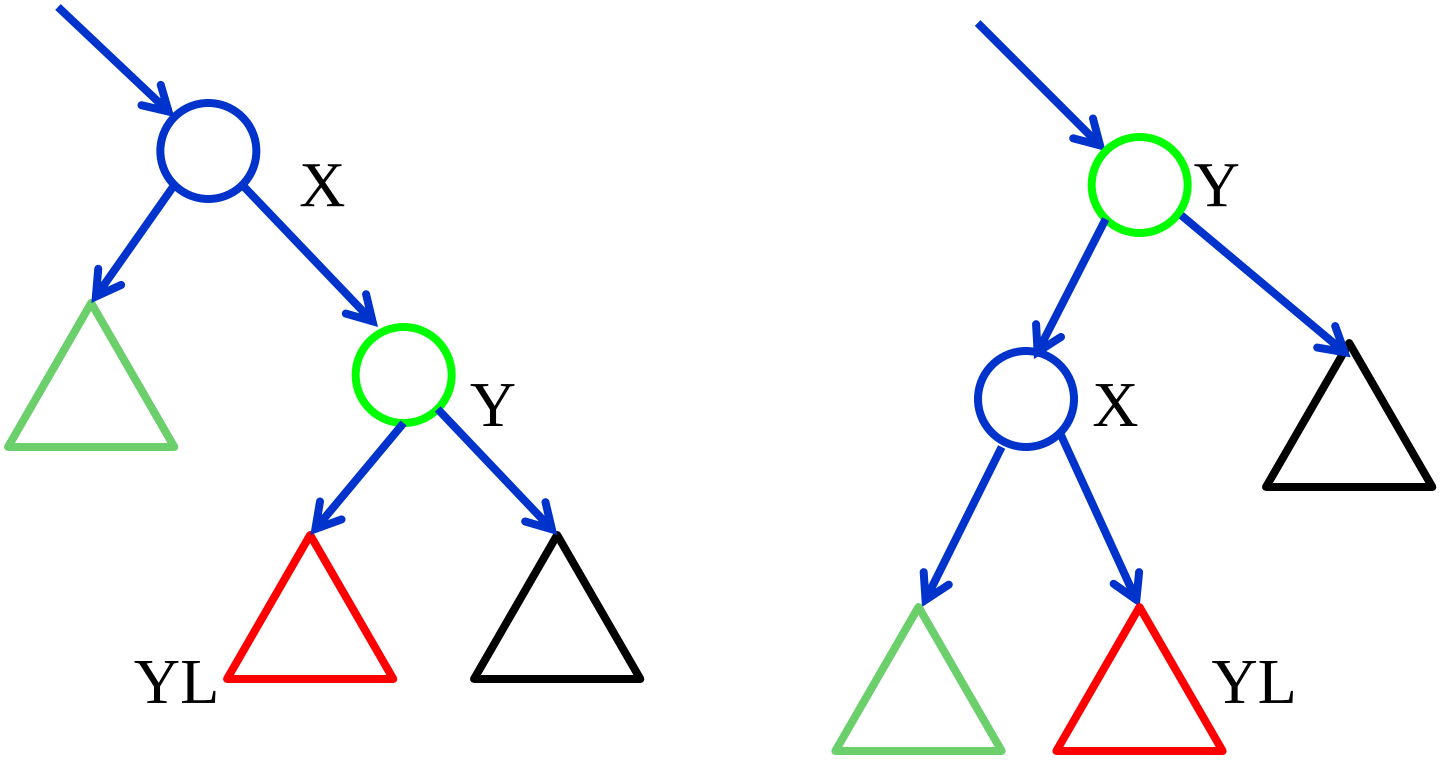
Left Rotation

Let X be the parent and Y be the right child.

TO DO:

1. Make X Left Child of Y
2. Make “previous Left Child of Y,
   as the Right Child of X.

Left Rotation
LEFT-ROTATE(T, x)



More on Rotation

```
LEFT-ROTATE(T, x)
 1   y = x.right                  // set y
 2   x.right = y.left             // turn y's left subtree into x's right subtree
 3   if y.left ≠ T.nil
 4       y.left.p = x
 5   y.p = x.p                    // link x's parent to y
 6   if x.p == T.nil
 7       T.root = y
 8   elseif x == x.p.left
 9       x.p.left = y
10   else x.p.right = y
11   y.left = x                   // put x on y's left
12   x.p = y
```

More on Rotation

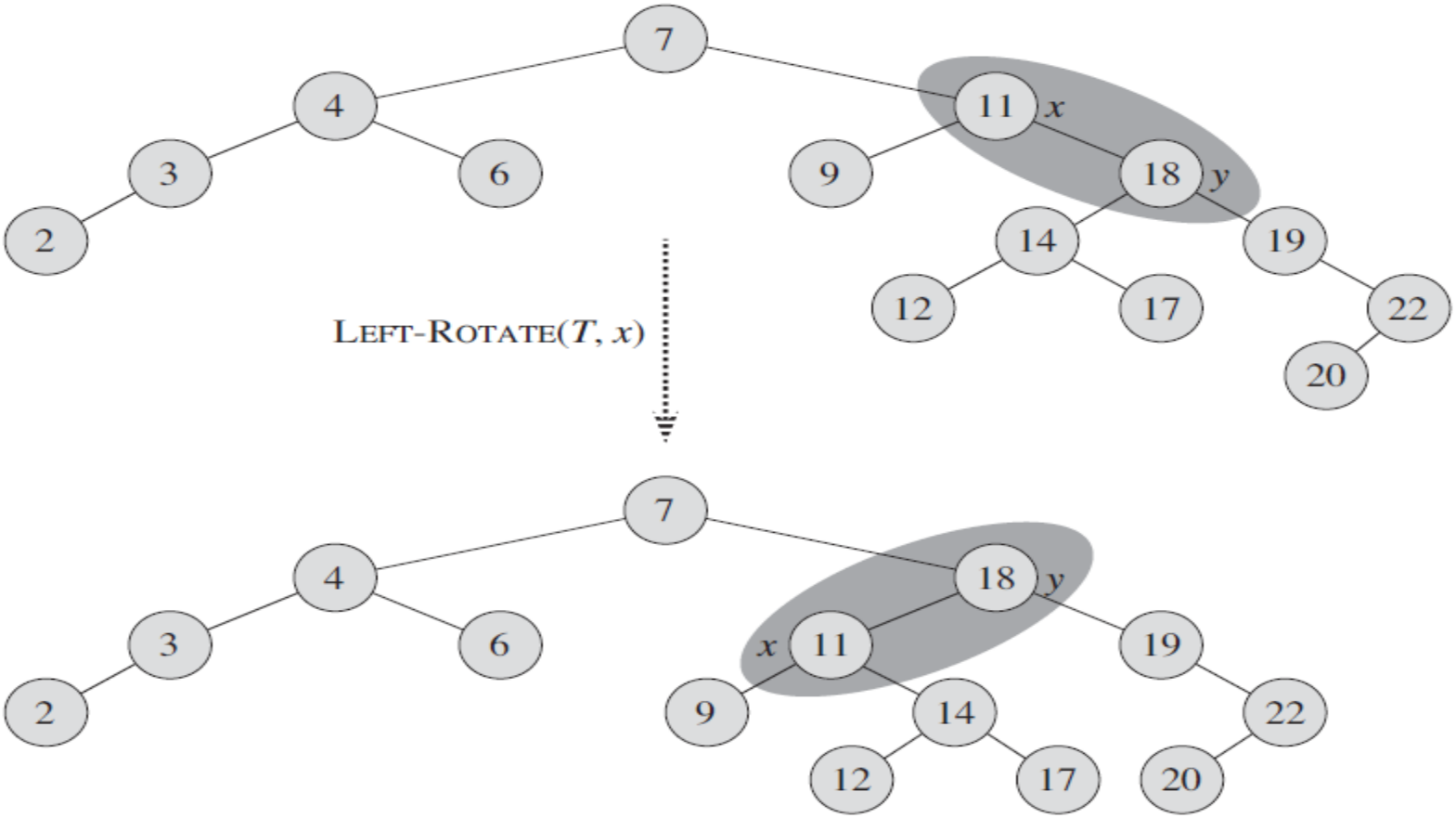


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

RB Insert

```
RB-INSERT(T, z)
 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)
```

RB Insert

Note: There should be an "else" after Case 2. The book has it wrong

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                    // case 1
 6              y.color = BLACK                      // case 1
 7              z.p.p.color = RED                    // case 1
 8              z = z.p.p                            // case 1
 9          else if z == z.p.right
10                  z = z.p                          // case 2
11                  LEFT-ROTATE(T, z)                // case 2
12              z.p.color = BLACK                    // case 3
13              z.p.p.color = RED                    // case 3
14              RIGHT-ROTATE(T, z.p.p)               // case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

Inserting a new node

- Create a node Z and insert as in any BST
- Color Z Red
- **Restore the Red-Black tree properties**
  - **This involves many different steps.**
  - **That is illustrated next.**

Five cases

Z is the node that is RED, Z's Parent is also RED.

Case 1: Z has a RED uncle.

Change Parent, Uncle to BLACK

Change Grand Parent to RED

Z <- Z.Parent.Parent

Cases 2 and 3: parent is a LEFT child of the grand parent.

Case 2:(LR) Z has a no RED uncle and Z is RIGHT child.

Z <- Z.Parent

LEFT ROTATION(T, Z)

Case 3:(LL) Z has a no RED uncle and Z is LEFT child.

Change Color of Parent, Grand Parent

RIGHT ROTATION(T, Z.Parent.Parent)

At the end, assign BLACK color to Root.

Cases 4 and 5: when parent is a RIGHT child of the grand parent.

Case 4: (RL) Z has a no RED uncle and Z is LEFT child.

Z <- Z.Parent

RIGHT ROTATION(T, Z)

Case 3:(RR) Z has a no RED uncle and Z is RIGHT child.

Change Color of Parent, Grand Parent

LEFT ROTATION(T, Z.Parent.Parent)

At the end, assign BLACK color to Root.

Summary

Case 1: RED uncle. U + P BLACK, GP RED, Move UP, UP

NO RED UNCLE

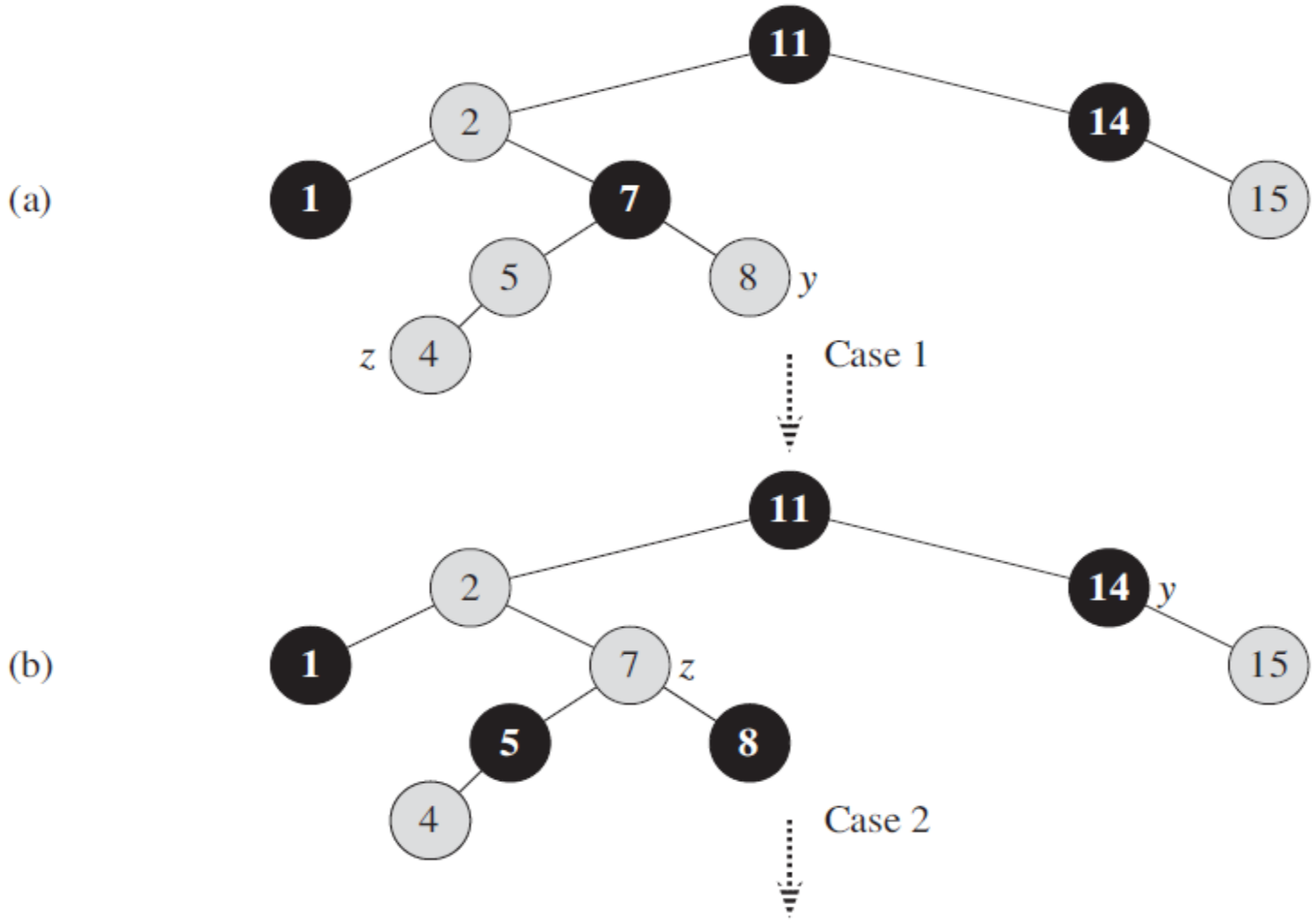
Case 2: (LR): Move UP, Rotate(L)

Case 3: (LL): Case 1 + Rotate(R)

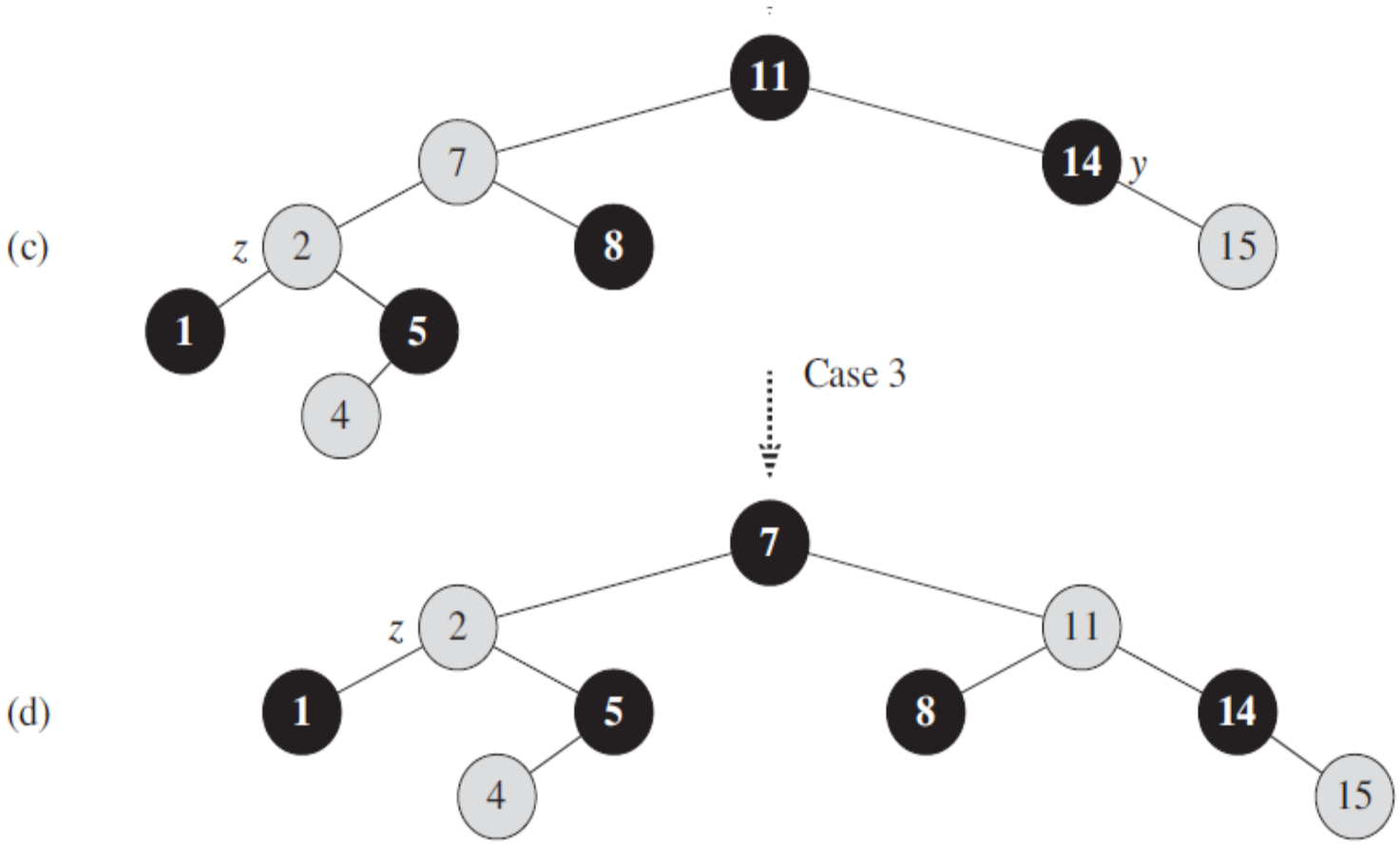
Case 4: (RL): Move UP, Rotate(R)

Case 5: (RR): Case 1 + Rotate(L)

RB Insert



RB Insert



RB Insert

(d)

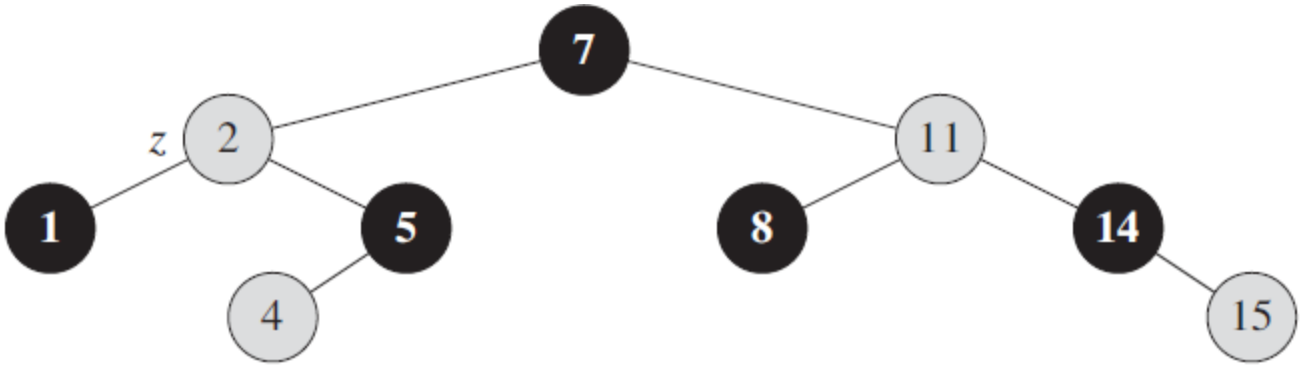
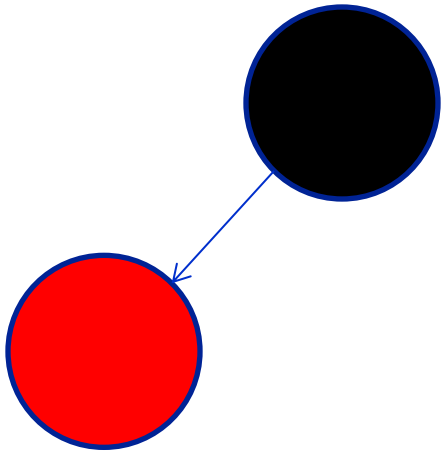


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. We recolor nodes and move the pointer z up the tree, resulting in the tree shown in **(b)**. 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.

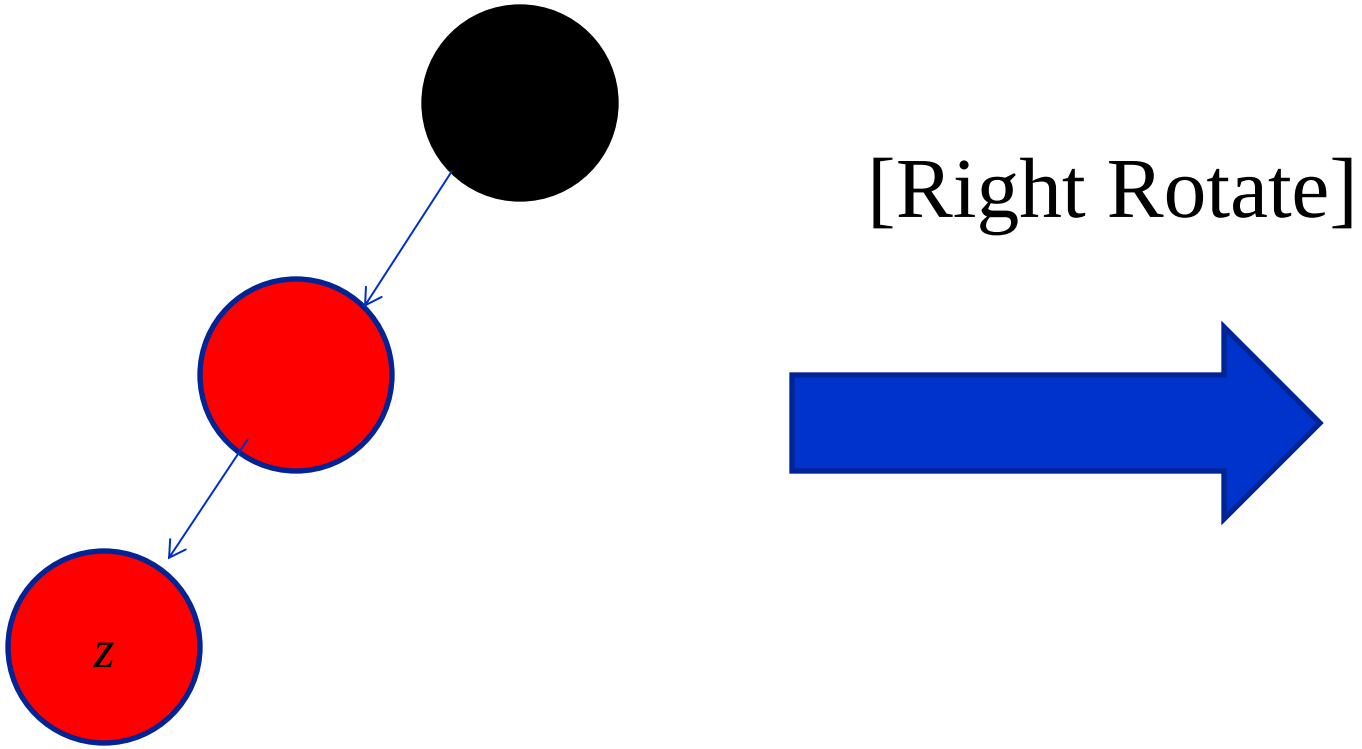
Red-Black Insertion

Code – No Case



Red-Black Insertion

Code – Case 3



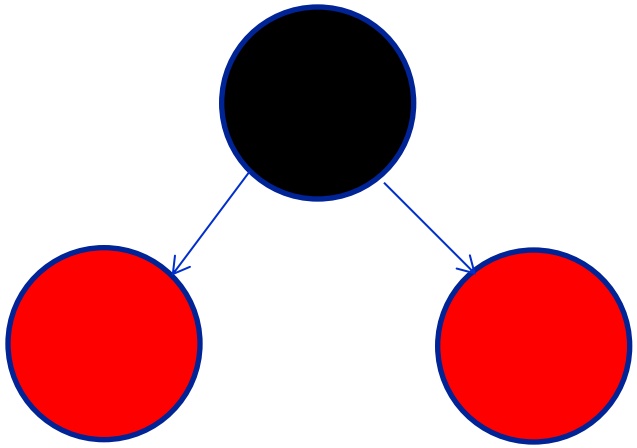
Red-Black Insertion

Code – Case 3

Change color:

Patent, GreandParent

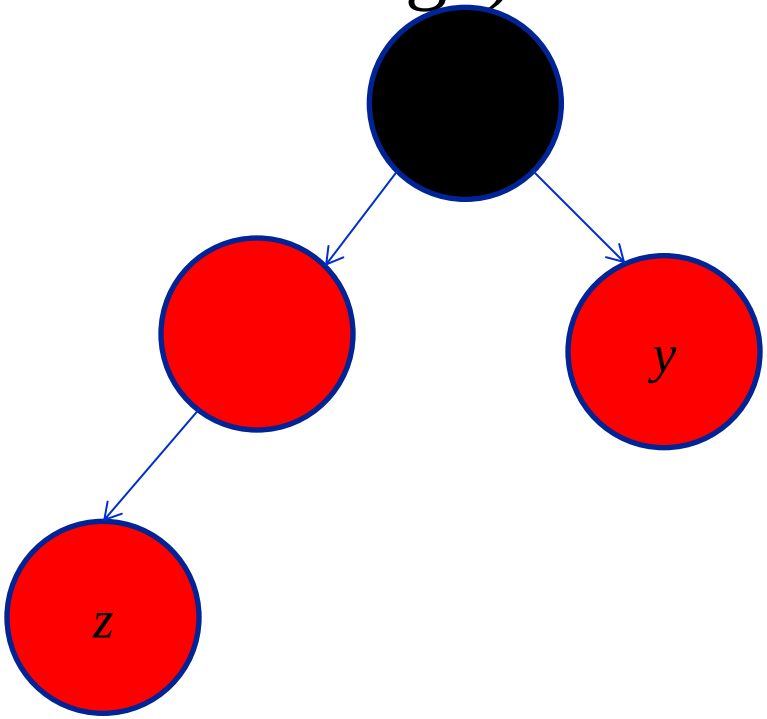
Right-Rotate(T, z.p.p)



Red-Black Insertion

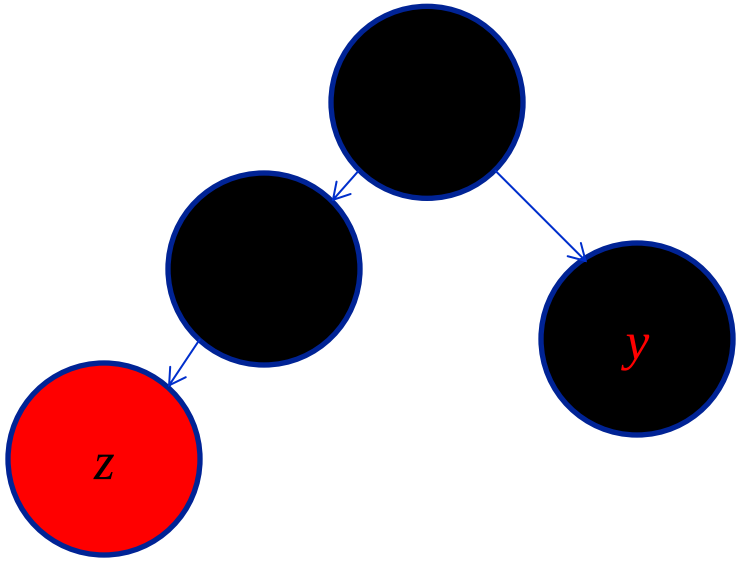
Code – Case 1 : Has a red uncle

(Just color change)



Red-Black Insertion

Case 1 P : Black, Uncle: Black Grand Parent: Red)



Make root Black

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.*

RB Trees: Worst-Case Time

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

Using a Red-Black Tree as a Dictionary

1. A dictionary is a collection of pairs (k, e), which we consider to be key/value pairs. The purpose of a dictionary is to make it possible to look up values by key.
2. The Dictionary ADT supports these operations:
    1. get(k) returns value (or e).
    2. put(k, e) insert the pair into dictionary.
    3. remove(k) remove any key/value pair whose key is k.
    4. containsKey(k) returns true if k occurs as a key in the dictionary.

Dictionary Implementations

1. List. Key/value pairs can be inserted into any kind of list. Insertion can be done by adding to the end of the list in O(1) time. However, lookups and deletes require O(n) time.
2. Hashtable. O(1) average case time for insert, lookup and delete operations. However, hashtables do not store keys in any particular order.

Dictionary Implementations

3. Red-Black Trees. When keys have a natural ordering (like integers or strings), key/value pairs can be stored in the nodes of a red-black tree and ordered by keys. Red-black tree has O(log n) worst case time for insert, lookup and delete.

Ordered Dictionary

- An ordered dictionary is a dictionary that maintains its data in sorted order.
- Red-black tree is an example of ordered dictionary
- The price paid for maintaining the order is
  - O(log n) vs. O(1) (Hashtable).
- Java's implementation of an ordered dictionary is TreeMap.

Main Point 3

The integrity of red-black trees is preserved after tree operations (insertions and deletions) are performed by maintaining the balance condition after execution of each operation. This maintenance does not increase the cost of operations because it requires only constant time, involving local color changes, color flips, and rotations.

Main Point 3

Science of Consciousness: The ability to maintain its fundamental character in the face of change is the expression of the *invincible* quality of pure consciousness. Pure consciousness, in giving rise to diversity, maintains its unbounded and immortal status. In society, this invincible quality is seen when a small percentage of a population engages in group practice of the TM and TM-Sidhi Programs – the inherent harmony of the society is enlivened to the extent that it “averts the birth of an enemy.”

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Red-Black Trees

1. In order to keep the benefits of a BST but eliminate the possibility of a worst case running time of $\Omega(n)$, different types of balance conditions have emerged, resulting in AVL trees and others.
2. Among enhanced BSTs with a balance condition, red-black trees are the most efficient. They outperform AVL trees by eliminating the need, in the worst case, to traverse the full height of the tree more than once.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

3. *Transcendental Consciousness* is the home of all the laws of nature. Enlivenment of this field brings support of nature's almighty power to the field of action.
4. *Impulses Within The Transcendental Field*. Within its unmanifest nature, pure consciousness, as a field of all possibilities, is capable of accomplishing anything, and sets forth a blueprint for manifesting the entire universe.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

5. *Wholeness Moving Within Itself.* In Unity Consciousness, the transformational dynamics of consciousness, at the basis of all of creation, are appreciated as the lively impulses of one's own consciousness, one's own being.

Visualalization

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html