*CHAPTER*
*10*
# Red-black Trees

**Objectives**

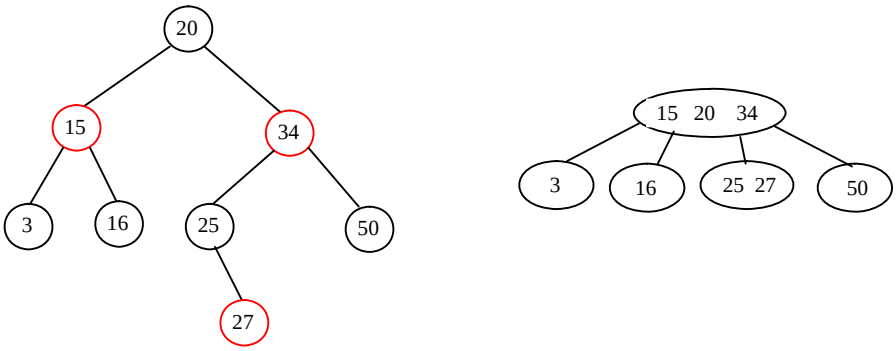
- To know what a red-black tree is (§10.1).
- To convert a red-black tree to a 2-4 tree and vice versa (§10.2).
- To design the RBTree class that extends the BinaryTree class (§10.3).
- To insert an element in a red-black tree and resolve the double red problem if necessary (§10.4).
- To insert an element from a red-black tree and resolve the double black problem if necessary (§10.5).
- To implement and test the RBTree class (§§10.6-10.7).
- To compare the performance of AVL trees, 2-4 trees, and RBTree (§10.8).

## 10.1 Introduction

***<Side Remark: derived from 2-4>***
***<Side Remark: color attribute>***
***<Side Remark: external>***
***<Side Remark: black depth>***

A red-black tree is a binary search tree, which is derived from a *2-4 tree*. A red-black tree corresponds to a 2-4 tree. Each node in a red-black tree has a *color attribute* red or black, as shown in Figure 10.1(a). A node is called *external* if its left or right subtree is empty. Note that a leaf node is external, but an external node is not necessarily a leaf node. For example, node 25 is external, but it is not a leaf. The *black depth* of a node is defined as the number of black nodes in a path from the node to the root. For example, the black depth of node 25 is 2 and the black depth of node 27 is 2.

******PD: The color of the nodes is important in this chapter. Use the second color of this book to substitute the red color in the manuscript. AU***




(a) A red-black tree (b) A 2-4 tree

**Figure 10.1**
*A red-black tree can be represented using a 2-4 tree, and vice versa.*

A red-black tree has the following properties:

1. The root is black.
2. Two adjacent nodes cannot be both red.
3. All external nodes have the same black depth.

The red-black tree in Figure 10.1(a) satisfies all three properties. A red-black tree can be converted to a 2-4 tree, and vice versa. Figure 10.1(b) shows an equivalent 2-4 tree for the red-black tree in Figure 10.1(a).

## 10.2 Conversion between Red-Black Trees and 2-4 Trees

You can design insertion and deletion algorithms for red-black trees without the knowledge of 2-4 trees. However, the correspondence between red-black trees and 2-4 trees provides useful intuition for understanding the structure of red-black trees and operations. For this reason, this section discusses the correspondence between these two types of trees.

***<Side Remark: red-black to 2-4>***

To convert a red-black tree to a 2-4 tree, simply merge any red nodes with its parent to create a 3-node or a 4-node. For example, the red nodes 15 and 34 are merged to its parent to create a 4-node, and the red node 27 is merged to its parent to create a 3-node, as shown in Figure 10.1(b).

***<Side Remark: 2-4 to red-black>***

To convert a 2-4 tree to a red-black tree, perform the following transformations for each node $u$:

***<Side Remark: converting 2-node>***

1. If $u$ is a 2-node, color it black, as shown in Figure 10.2(a).

***<Side Remark: converting 3-node>***

2. If $u$ is a 3-node with element values $e_0$ and $e_1$, there are two ways to convert it. One is to make $e_0$ the parent of $e_1$ and the other is to make $e_1$ and $e_0$. In any case, color the parent black and the child red, as shown in Figure 10.2(b).

***<Side Remark: converting 4-node>***

3. If $u$ is a 4-node with element values $e_0$, $e_1$, and $e_2$, make $e_1$ the parent of $e_0$ and $e_2$. Color $e_1$ black and $e_0$ and $e_2$ red, as shown in Figure 10.2(c).

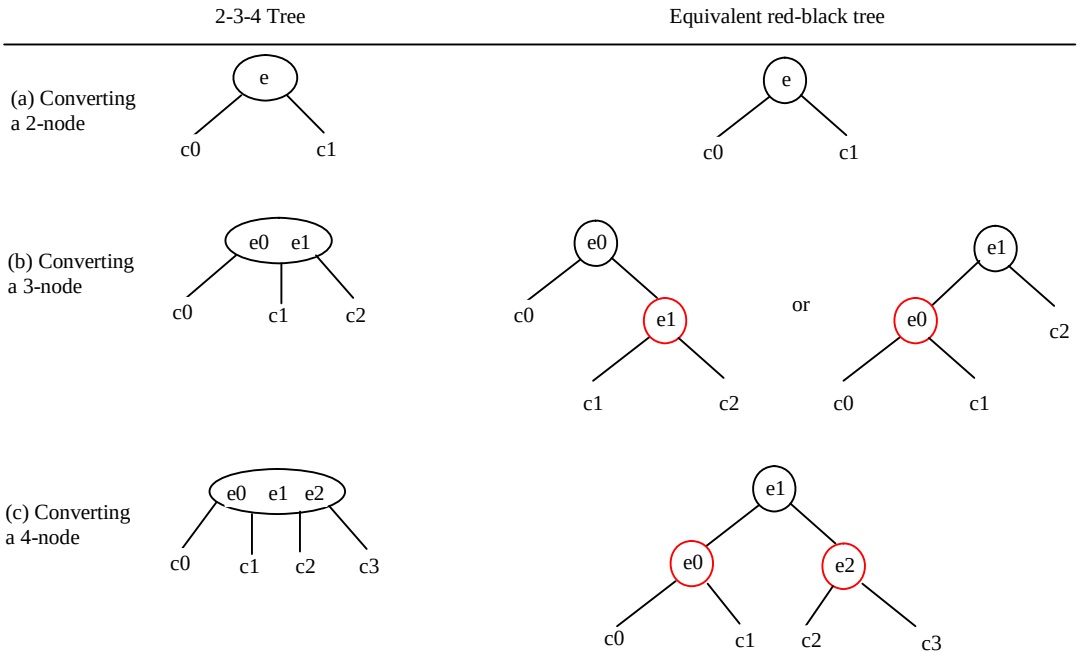



**Figure 10.2**

*A node in a 2-4 tree can be transformed to nodes in a red-black tree.*

***<Side Remark: not unique>***

Let us apply the transformation for the 2-4 tree in Figure 10.1(b). After transforming the 4-node, the tree is shown in Figure 10.3(a). After transforming the 3-node, the tree is shown in Figure 10.3(b). Note that the transformation for a 3-node is not unique. Therefore, the

conversion from a 2-4 tree to a red-black tree is *not unique*. After transforming the 3-node, the tree could be shown in Figure 10.4.

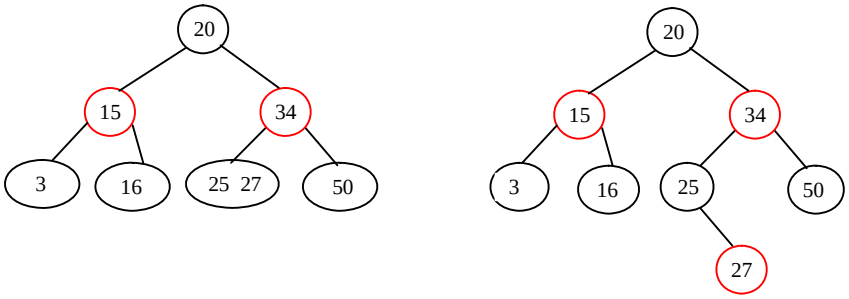



**Figure 10.3**
*A node in a 2-4 tree can be transformed to nodes in a red-black tree.*

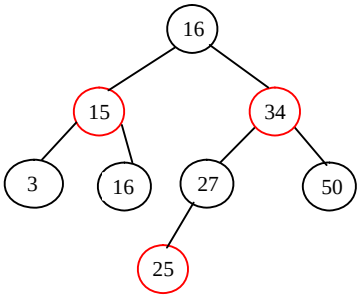



**Figure 10.4**
*The conversion from a 2-4 tree to a red-black tree is not unique.*

You can prove that the conversion results in a red-black tree that satisfies all three properties.

***<Side Remark: Property 1 proof>***
**Property 1**. The root is black.
Proof: If the root of a 2-4 tree is a 2-node, the root of the red-black tree is black. If the root of a 2-4 tree is a 3-node or 4-node, the transformation produces a black parent at the root.

***<Side Remark: Property 2 proof>***
**Property 2**. Two adjacent nodes cannot be both red.
Proof: Since the parent of a red node is always black, no two adjacent nodes can be both red.

***<Side Remark: Property 3 proof>***
**Property 3**. All external nodes have the same black depth.
Proof: When you covert a node in a 2-4 tree to red-black tree nodes, it produces one black node, zero, one, or two red nodes as its children, depending on whether the original node is a 2-node, 3-node, or 4-node. Only a leaf 2-4 node may produce external red-black nodes. Since a 2-4 tree is perfectly balanced, the number of black nodes in any path from the root to an external node is the same.

## 10.3 Designing Classes for Red-Black Trees

A red-black tree is a binary tree. So you can define the RBTree class to extend the BinaryTree class, as shown in Figure 10.5.

***New Figure
<PD: UML Class Diagram>

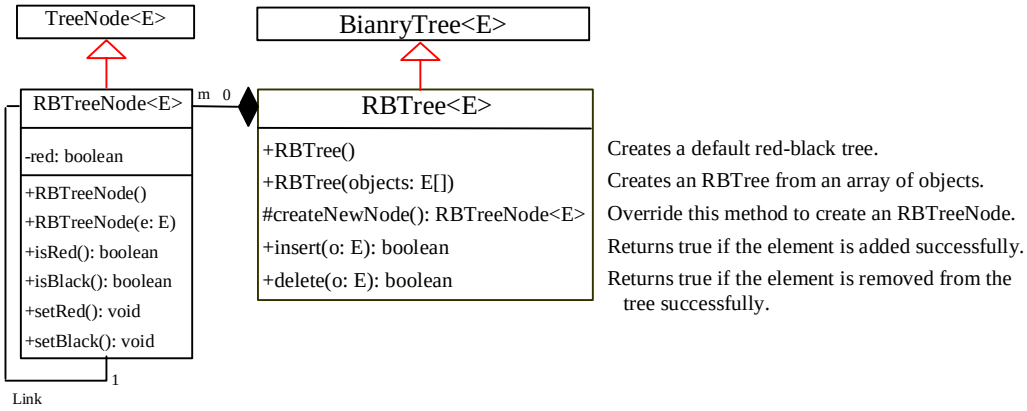



**Figure 10.5**
*The RBTree class extends BinaryTree with new implementations for the remove and delete methods.*

<Side Remark: RBTreeNode>
Each node in a red-black tree has a color property. Because the color is either red or black, it is efficient to use the boolean type to denote the color. The RBTreeNode class can be defined to extend BianryTree.TreeNode with the color property. For convenience, we also provide the methods for checking the color and setting a new color. Note that TreeNode is defined as a static inner class in BinaryTree. RBTreeNode will be defined as a static inner class in RBTree. Note that BinaryTreeNode contains the data fields element, left, and right, which are inherited in RBTreeNode. So, RBTreeNode contains four data fields, as pictured in Figure 10.6.

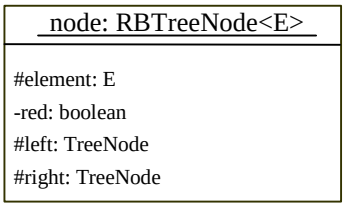



**Figure 10.6**
*An RBTreeNode contains data fields element, red, left, and right.*

<Side Remark: createNewNode()>
In the BinaryTree class, the createNewNode() method creates a TreeNode object. This method is overridden in the RBTree class to create an RBTreeNode. Note that the return type of the createNewNode() method in the BianryTree class is TreeNode, but the return type of the createNewNode() method in RBTree class is RBTreeNode. This is fine, since RBTreeNode is a subtype of TreeNode.

Searching an element in a red-black tree is the same as searching in a regular binary tree. So, the search method defined in the BinaryTree class also works for RBTree.

The insert and delete methods are overridden to insert and delete an element and perform operations for coloring and restructuring if necessary to ensure that the three properties of the red-black tree is satisfied.

Pedagogical NOTE

***<side remark: Red-Black tree animation>***

Run from www.cs.armstrong.edu/liang/jds/exercisejds/Exercise11_3.html to see how a red-black tree works, as shown in Figure 10.7.

**Figure 10.7**

*The animation tool enables you to insert, delete, and search elements in a red-black tree visually.*

******End NOTE***

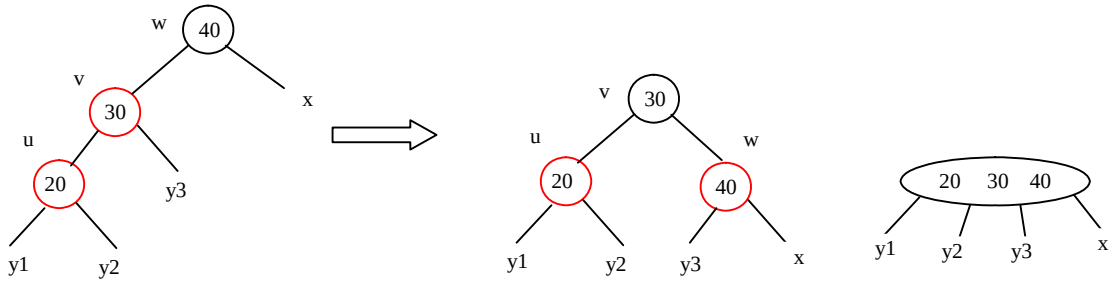
## 10.4 Overriding the insert Method

***<Side Remark: double red>***

A new element is always inserted as a leaf node. If the new node is the root, color it black. Otherwise, color it red. If the parent of the new node is red, it violates Property 2 of the red-black tree. We call this the *double red* violation.

Let $u$ denote the new node inserted, $v$ denote the parent of $u$, and $w$ denote the parent of $v$, and $x$ denote the sibling of $v$. To fix the double red violation, consider two cases:

Case 1: $x$ is black or $x$ is null. There are four possible configurations for $u$, $v$, $w$, and $x$, as shown in Figures 10.7(a), 10.8(a), 10.9(a), and 10.10(a). Note that $x$, $y_1$, $y_2$, and $y_3$ may be null. In this case, $u$, $v$, and $w$ form a 4-node in the corresponding 2-4 tree, as shown in Figure 10.8(c), 10.9(c), 10.10(c), and 10.11(c), but are represented incorrectly in the red-black tree. To correct it, restructure and recolor three nodes $u$, $v$, and $w$, as shown in Figure 10.8(b), 10.9(b), 10.10(b), and 10.11(b).




(a) (b) (c)

**Figure 10.8**

*Case 1.1: u < v < w.*

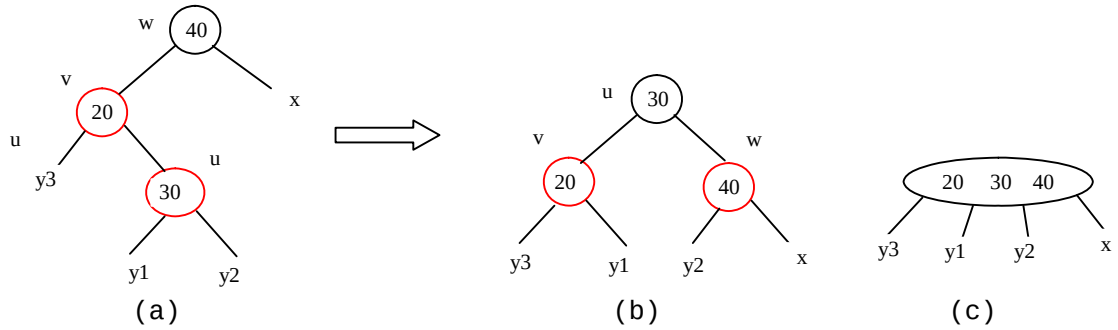



**Figure 10.9**
*Case 1.2: v < u < w*

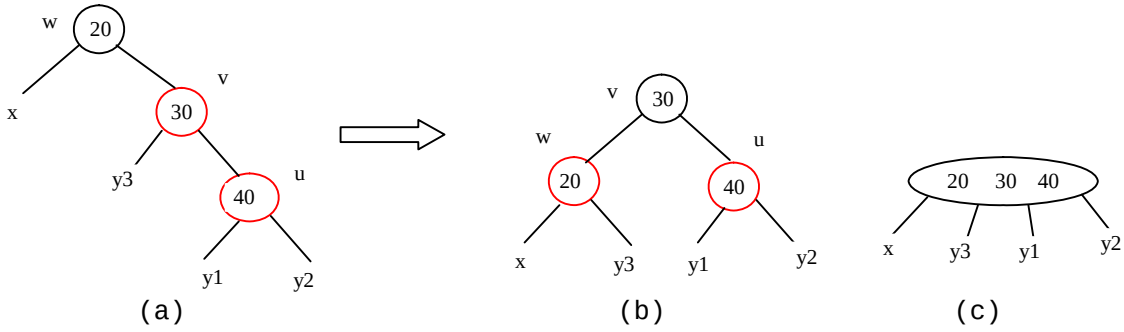



**Figure 10.10**
*Case 1.3: w < v < u*

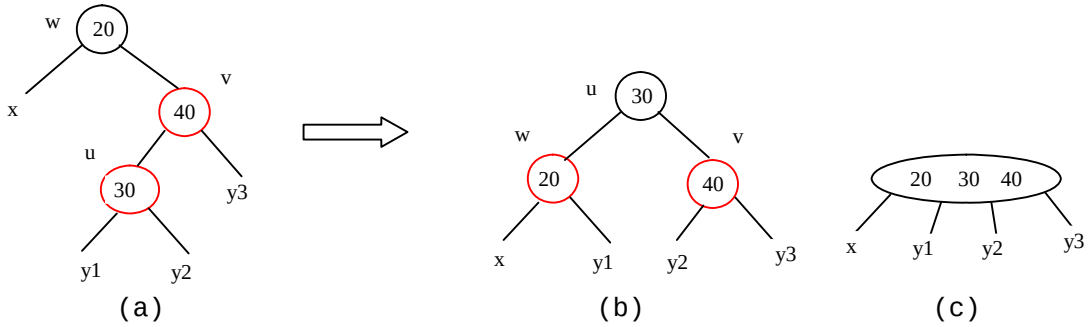



**Figure 10.11**
*Case 1.4: w < u < v*

Case 2: *x* is red. There are four possible configurations for *u*, *v*, *w*, and *x*, as shown in Figures 10.12(a), 10.12(b), 10.12(c), and 10.12(d). All of these configurations correspond to an overflow situation in the corresponding 4-node in a 2-4 tree, as shown in Figure 12.12(a). A splitting operation is performed to fix the overflow problem in a 2-4 tree. We perform an equivalent recoloring operation to fix the problem in a red-black tree. Color *w* and *u* red and color two children of *w* black. Assume *u* is a left child of *v*, as shown in Figure 12.11(a). After recoloring, the nodes are shown in Figure 12.12(c). Now *w* is red, if *w*'s parent is black, the double red problem is fixed. Otherwise, a new double red problem occurs at node *w*. We need to continue the same process to eliminate the double red problem at *w*, recursively.

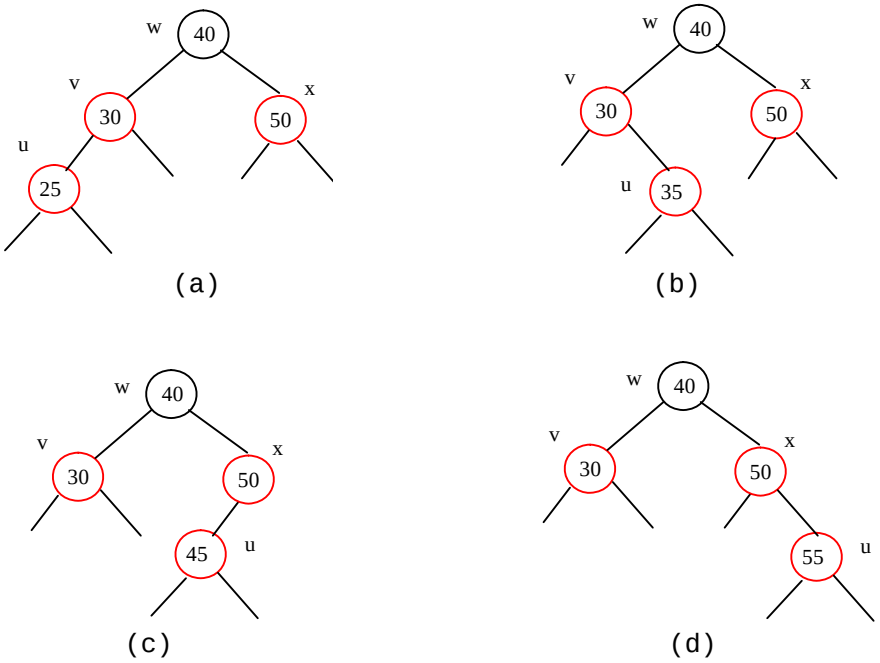



**Figure 10.12**
*Case 2 has four possible configurations.*

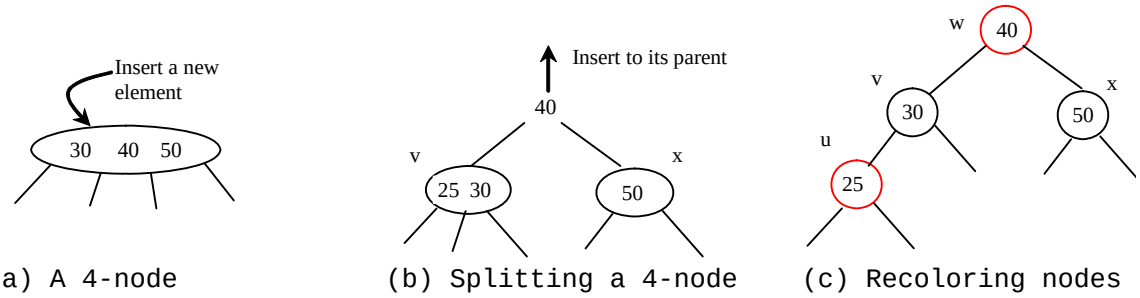



**Figure 10.13**
*Splitting a 4-node corresponds to recoloring of the nodes in the red-black tree.*

A more detailed algorithm for inserting an element is described in Listing 10.1.

**Listing 10.1 Inserting an Element to a Red-Black Tree**

```
***PD: Please add line numbers in the following code***
<Side Remark line 1: insert to tree>
<Side Remark line 2: invoke super.insert>
<Side Remark line 4: duplicate element>
<Side Remark line 6: ensure color and depth>

<Side Remark line 13: ensure color and depth>
<Side Remark line 14: get path>
<Side Remark line 15: node index>
<Side Remark line 16: get u, v>
```

```
<Side Remark line 20: u is root?>
<Side Remark line 22: double red problem>

<Side Remark line 27: fix double red>
<Side Remark line 29: get w>
<Side Remark line 32: get x>
<Side Remark line 36: Case 1>
<Side Remark line 38: Case 1.1>
<Side Remark line 41: Case 1.2>
<Side Remark line 44: Case 1.3>
<Side Remark line 47: Case 1.4>
<Side Remark line 50: Case 2>
<Side Remark line 51: recoloring>
<Side Remark line 54: w is root?>
<Side Remark line 57: propagate upward>
<Side Remark line 61: fix new double red>
```

```
public boolean insert(E e) {
  boolean successful = super.insert(e);
  if (!successful)
    return false; // e is already in the tree
  else {
    ensureRBTree(e);
  }

  return true; // e is inserted
}

/** Ensure that the tree is a red-black tree */
private void ensureRBTree(E e) {
  Get the path that leads to element e from the root.
  int i = path.size() – 1; // Index to the current node in the path
  Get u, v from the path. u is the node that contains e and v
    is the parent of u.
  Color u red;

  if (u == root) // If e is inserted as the root, set root black
    u.setBlack();
  else if (v.isRed())
    fixDoubleRed(u, v, path, i); // Fix double red violation at u
}

/** Fix double red violation at node u */
private void fixDoubleRed(RBTreeNode<E> u, RBTreeNode<E> v,
    ArrayList<TreeNode<E>> path, int i) {
  Get w from the path. w is the grandparent of u.

  // Get v's sibling named x
  RBTreeNode<E> x = (w.left == v) ?
    (RBTreeNode<E>)(w.right) : (RBTreeNode<E>)(w.left);

  if (x == null || x.isBlack()) {
    // Case 1: v's sibling x is black
    if (w.left == v && v.left == u) {
```

```
      // Case 1.1: u < v < w, Restructure and recolor nodes
    }
    else if (w.left == v && v.right == u) {
      // Case 1.2: v < u < w, Restructure and recolor nodes
    }
    else if (w.right == v && v.right == u) {
      // Case 1.3: w < v < u, Restructure and recolor nodes
    }
    else {
      // Case 1.4: w < u < v, Restructure and recolor nodes
    }
  }
  else { // Case 2: v's sibling x is red
    Color w and u red
    Color two children of w black.

    if (w is root) {
      Set w black;
    }
    else if (the parent of w is red) {
      // Propagate along the path to fix new double red violation
      u = w;
      v = parent of w;
      fixDoubleRed(u, v, path, i - 2); // i – 2 propagates upward
    }
  }
}
```

### <Side Remark: insert(E, e)>

The insert(E e) method (lines 1-10) invokes the insert method in the BinaryTree class to create a new leaf node for the element (line 2). If the element is already in the tree, return false (line 4). Otherwise, invoke ensureRBTree(e) (line 6) to ensure that the tree satisfies the color and black depth property of the red-black tree.

### <Side Remark: ensureRBTree(E, e)>

The ensureRBTree(E e) method (lines 13-25) obtains the path that leads to e from the root (line 15), as shown in Figure 10.14. This path plays an important role to implementing the algorithm. From this path, you get nodes u and v (lines 16-17). If u is the root, color u black (lines 20-21). If v is red, a double black problem occurs at node u. Invoke fixDoubleRed to fix the problem.

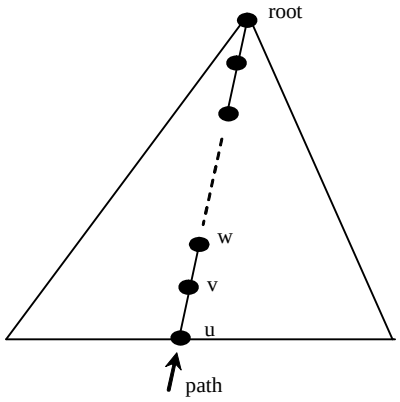


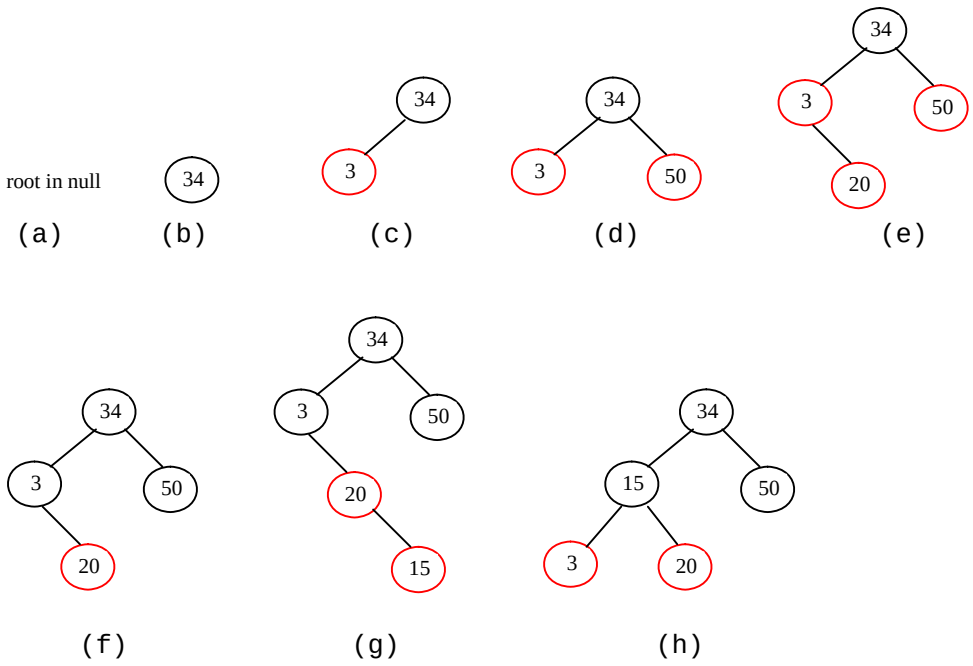
**Figure 10.14**
*The path consists of the nodes from u to the root.*

### *<Side Remark: fixDoubleRed>*

The fixDoubleRed method (lines 27-63) fixes the double red problem. It first obtains w (the parent of v) from the path (line 29) and x (the sibling of v) (lines 32-33). If x is empty or a black node, restructure and recolor three nodes u, v, and w to eliminate the problem (lines 35-49). If x is a red node, recolor the nodes u, v, w and x (lines 51-52). If w is the root, color w black (lines 54-56). If the parent of w is red, the double red problem reappears at w. Invoke fixDoubleRed with new u and v to fix the problem. Note that now i – 2 points to the new u in the path. This adjustment is necessary to locate the new nodes w and parent of w along the path.

### *<Side Remark: insertion example>*

Figure 10.15 shows the steps of inserting elements. When inserting 20 into the tree in (d), Case 2 applies to recolor 3 and 50 to black. When inserting 15 into the tree in (e), Case 1.3 applies to restructure and recolor nodes 15, 20, and 3. When inserting 16 into the tree in (f), Case 2 applies to recolor nodes 3 and 20 to black and nodes 15 and 16 to red. When inserting 27 into the tree in (h), Case 2 applies to recolor nodes 16 and 25 to black and nodes 20 and 27 to red. Now new red double problem occurs at node 20. Apply Case 1.2 to restructure and recolor nodes. The new tree is shown in (i).



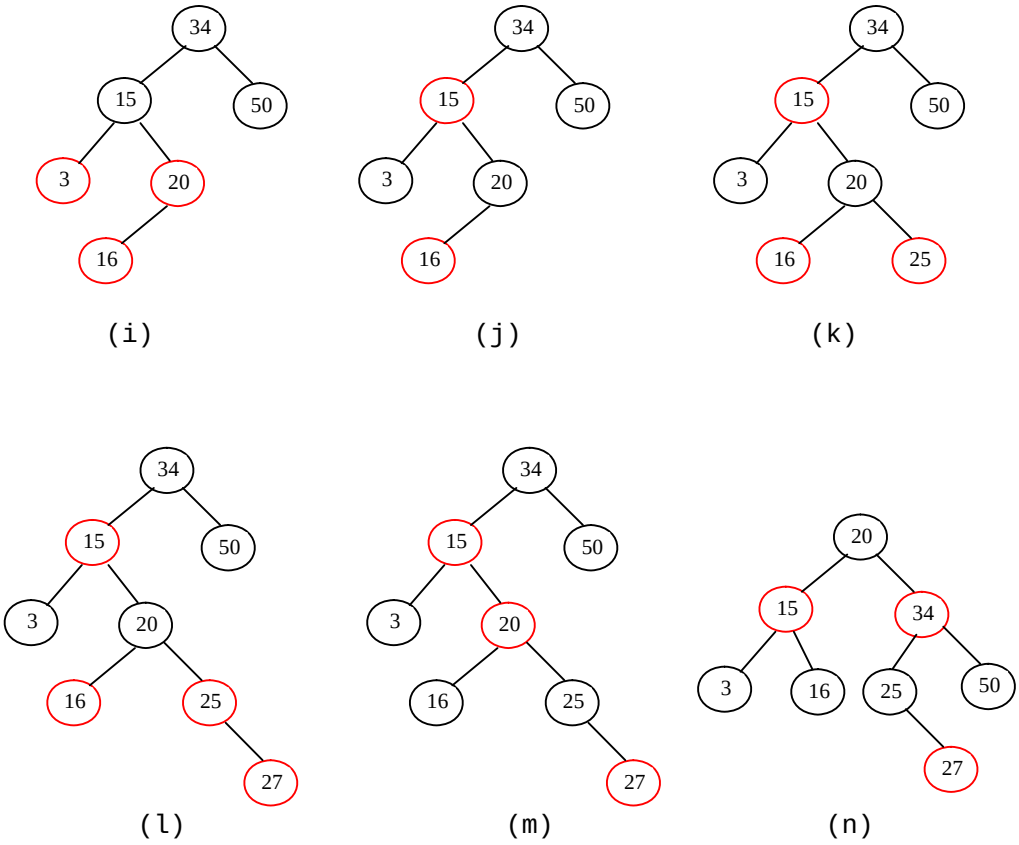
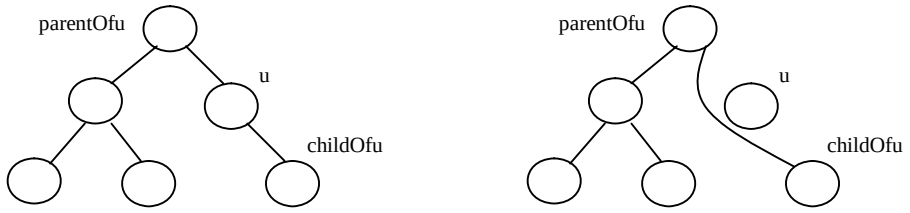



**Figure 10.15**
*Inserting into a red-black tree: (a) initial empty tree; (b) inserting 34; (c) inserting 3; (d) inserting 50; (e) inserting 20 causes a double red; (f) after recoloring (Case 2); (g) inserting 15 causes a double red; (h) after restructuring and recoloring (Case 1.4); (i) inserting 16 causes a double red; (j) after recoloring (Case 2); (k) inserting 25; (l) inserting 27 causes a double red at 27; (m) a double red at 20 reappears after recoloring (Case 2); (n) after restructuring and recoloring (Case 1.2).*

## 10.5 Overriding the delete Method

To delete an element from a red-black tree, first search the element in the tree to locate the node that contains the element. If the element is not in the tree, the method returns false. Let *u* be the node that contains the element. If *u* is an internal node with both left and right children, find the rightmost node in the left subtree of *u*. Replace the element in *u* with the element in the rightmost node. Now we will only consider deleting external nodes.

Let *u* be an external node to be deleted. Since *u* is an external node, it has at most one child, denoted by *childOfu*. *childOfu* may be null. Let *parentOfu* denote the parent of *u*, as shown in Figure 10.16(a). Delete *u* by connecting *childOfu* with *parentOfu*, as shown in Figure 10.15(b).




(a) Before deleting u (b) After deleting u

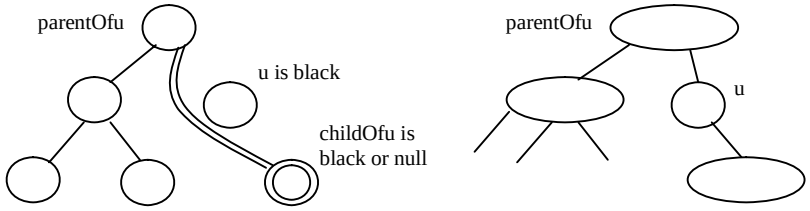
**Figure 10.16**
*u is an external node and childOfu may be null.*

Consider the following case:

- If *u* is red, we are done.
- If *u* is black and *childOfu* is red. Color *childOfu* black to maintain the black height for *childOfu*.

***<Side Remark: double black>***

- Otherwise, assign *childOfu* a fictitious *double black*, as shown in Figure 10.17(a).




**(a)** **(b)**

**Figure 10.17**
*(a) childOfu is denoted double black. (b) u corresponds to an empty node in a 2-4 tree.*

A double black in a red-black tree corresponds to an empty node for u (i.e., underflow situation) in the corresponding 2-4 tree, as shown in Figure 10.17(b). To fix the double black problem, we will perform equivalent transfer and fusion operations. Consider three cases:

***<Side Remark: Case 1>***

**Case 1**: The sibling y of *childOfu* is black and has a red child. Such case has four possible configurations, as shown in Figures 10.16(a), 10.17(a), 10.18(a), and 10.19(a). Note that the dashed circle denotes that the node is either red or black. To eliminate the double black problem, restructure and recolor the nodes, as shown in Figures 10.16(b), 10.17(b), 10.18(b), and 10.19(b).

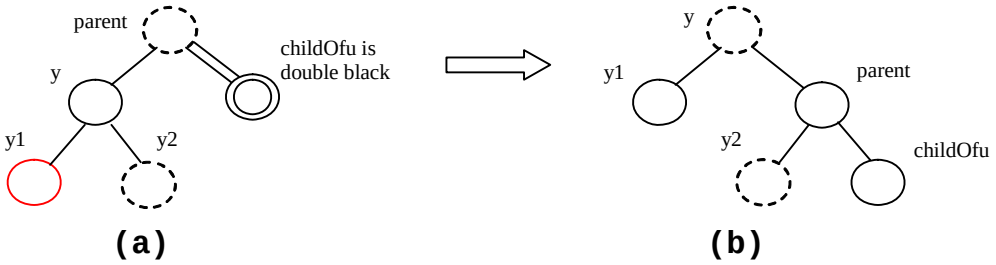



**Figure 10.17**
*Case 1.1: the sibling y of childOfu is black and y1 is red.*

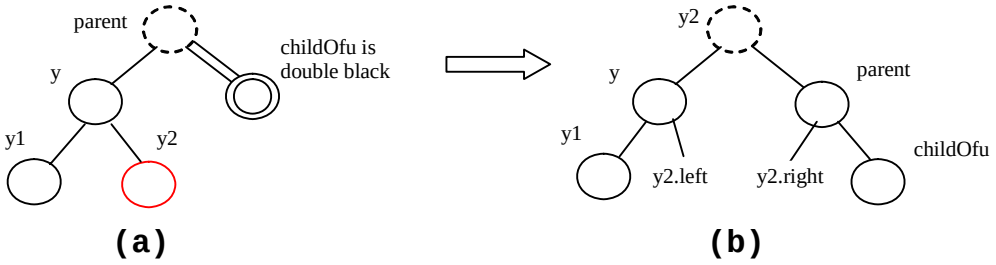



**Figure 10.18**
*Case 1.2: the sibling y of childOfu is black and y2 is red.*

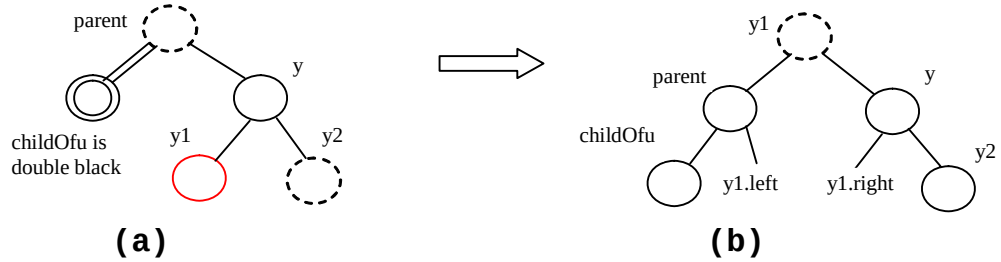



**Figure 10.19**
*Case 1.3: the sibling y of childOfu is black and y1 is red.*

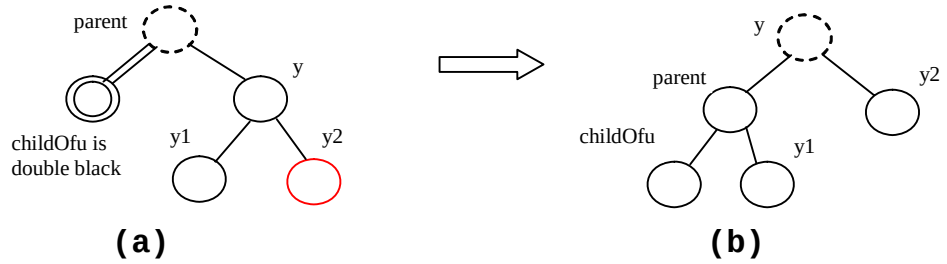
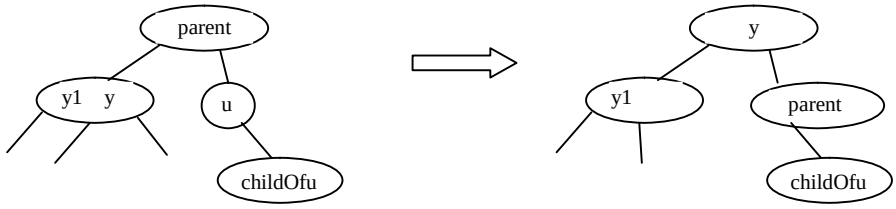


**Figure 10.20**
*Case 1.4: the sibling y of childOfu is black and y2 is red.*

Note

***<Side Remark: transfer operation>***

Case 1 corresponds to a *transfer* operation in the 2-4 tree. For example, the corresponding 2-4 tree for Figure 10.17(a) is shown in Figure 10.21(a) and it is transformed into 10.21(b) through a transfer operation.



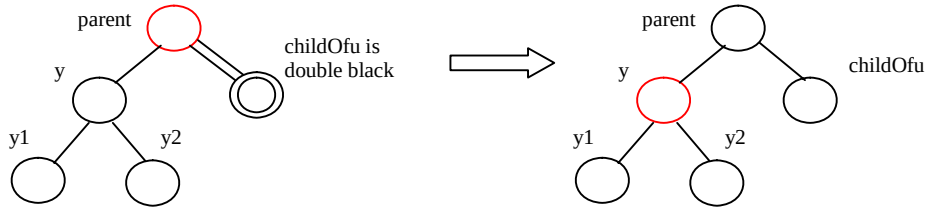

**(a)** **(b)**

**Figure 10.21**
*Case 1 corresponds to a transfer operation in the corresponding 2-4 tree.*

******END of NOTE***

***<Side Remark: Case 2>***
***<Side Remark: propagate>***

**Case 2:** The sibling y of *childOfu* is black and its children are black or null. In this case, change y's color to red. If parent is red, change it to black, we are done, as shown in Figure 10.22. If parent is black, we denote parent double black, as shown in Figure 10.23. The double black problem *propagates* to the parent node.




**(a)** **(b)**

**Figure 10.22**
*Case 2: Recoloring eliminates the double black problem if parent is red.*

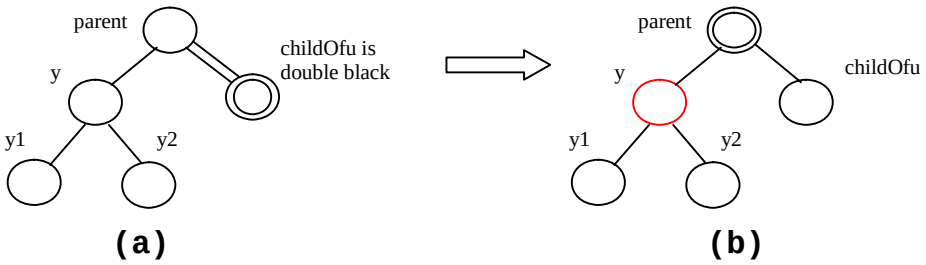



**Figure 10.23**
*Case 2: Recoloring propagates the double black problem if parent is black.*

Note

***<Side Remark: left childOfu>***
Figures 10.21 and 10.22 show that childOfu is a right child of parent. If childOfu is a left child of parent, recoloring is performed identically.

Note

***<Side Remark: fusion operation>***
Case 2 corresponds to a *fusion* operation in the 2-4 tree. For example, the corresponding 2-4 tree for Figure 10.22(a) is shown in Figure 10.24(a) and it is transformed into 10.24(b) through a fusion operation.

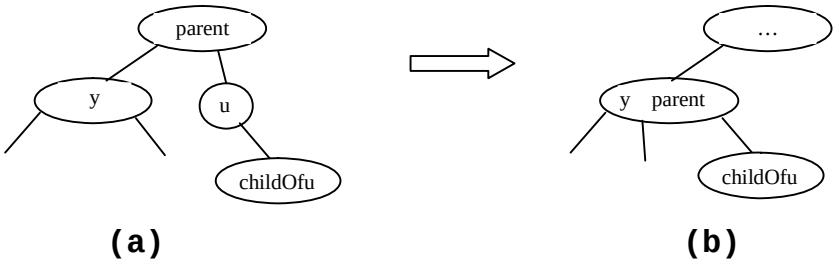



**Figure 10.24**
*Case 2 corresponds to a fusion operation in the corresponding 2-4 tree.*

******END of NOTE***

***<Side Remark: Case 3>***
***<Side Remark: adjustment>***
**Case 3:** The sibling y of *childOfu* is red. In this case, perform an *adjustment* operation. If y is a left child of parent, let y2 be the left child of parent and parent be the right child of y, as shown in Figure

10.25. If y is a right child of parent, let y1 be the right child of parent and parent be the left child of y, as shown in Figure 10.26. In both cases, color y black and parent red. childOfu is still a fictitious double black node. After the adjustment, the sibling of childOfu is now black, either Case 1 or Case 2 applies. If Case 1 applies, one time restructure and recoloring operation eliminates the double black problem. If Case 2 applies, the double black problem cannot reappear, since parent is now red. Therefore, one time application of Case 1 or Case 2 will complete Case 3.

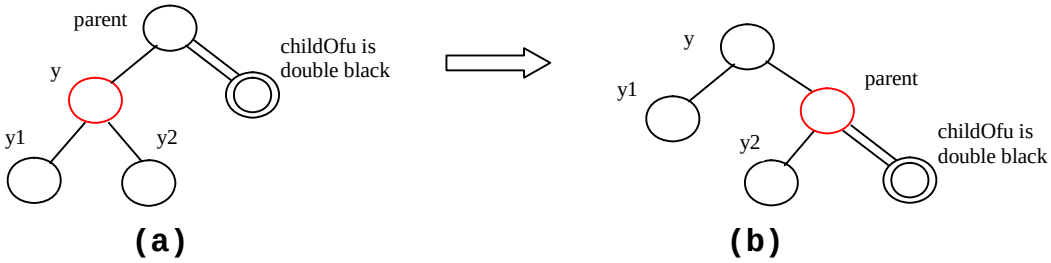



**Figure 10.25**
*Case 3.1: y is a left red child of parent.*

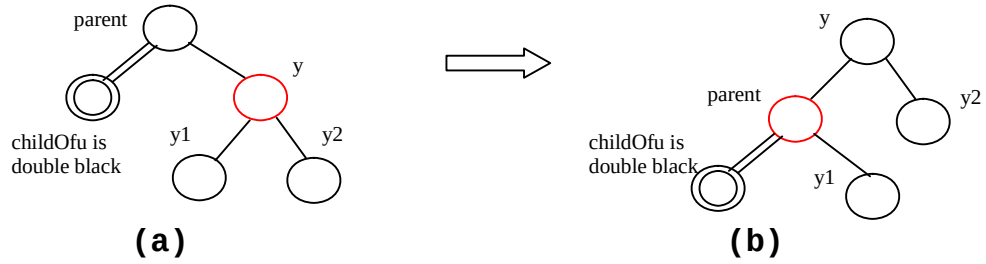



**Figure 10.26**
*Case 3.2: y is a right red child of parent.*

Note

***<Side Remark: non-unique transform of 3-node>***

Case 3 results from the fact that a 3-node may by transformed in two ways to a red-black tree, as shown in Figure 10.28.

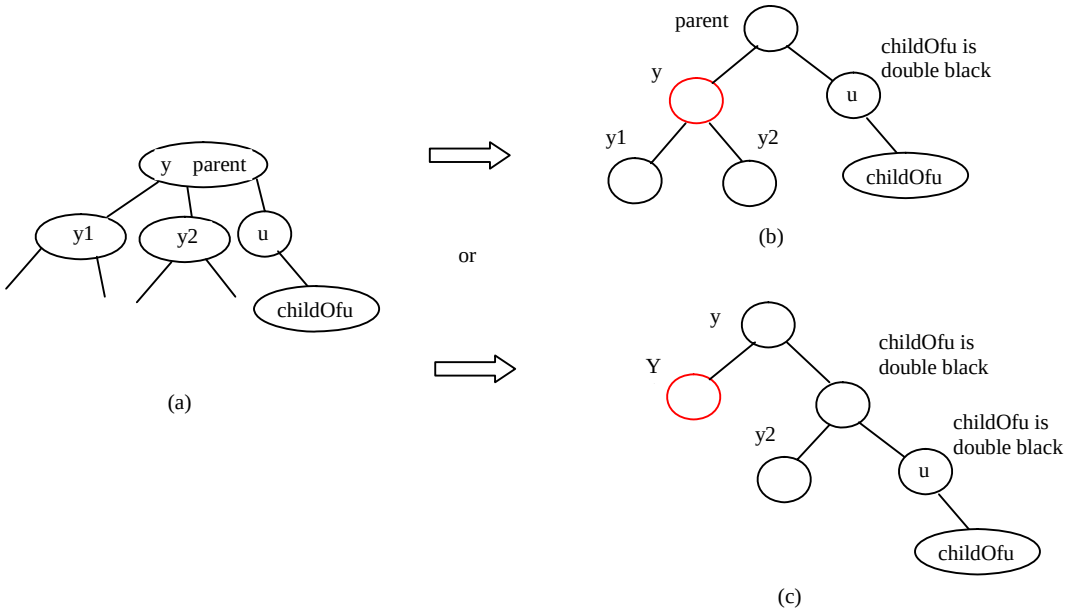



**Figure 10.28**
*A 3-node may be transformed in two ways to red-black tree nodes.*

Based on the foregoing discussion, a more detailed algorithm for deleting an element can be described in Listing 10.2.

**Listing 10.2 Deleting an Element from a Red-Black Tree**

```
***PD: Please add line numbers in the following code***
<Side Remark line 1: delete e from tree>
<Side Remark line 2: locate the node>
<Side Remark line 4: element not found>
<Side Remark line 6: internal element?>
<Side Remark line 7: rightmost node>
<Side Remark line 12: path to external node>
<Side Remark line 15: delete the node>
<Side Remark line 17: one element deleted>
<Side Remark line 18: deletion successful>

<Side Remark line 22: delete a node>
<Side Remark line 23: u>
<Side Remark line 24: parentOfu, grandparentOfu>
<Side Remark line 23: childOfu>
<Side Remark line 26: delete u>
<Side Remark line 30: done>
<Side Remark line 32: set childOfu black>
<Side Remark line 35: fix double black>

<Side Remark line 39: fix double black>
<Side Remark line 42: y, y1, y2>
<Side Remark line 47: process Case 1.1>
<Side Remark line 51: process Case 1.3>
<Side Remark line 57: process Case 1.2>
```

<Side Remark line 61: process Case 1.4>
<Side Remark line 66: process Case 2>
<Side Remark line 77: propagate double black>
<Side Remark line 83: process Case 3.1>
<Side Remark line 88: process Case 3.2>
<Side Remark line 95: fix double black>

```
public boolean delete(E e) {
  Locate the node to be deleted
  if (the node is not found)
    return false;

  if (the node is an internal node) {
    Find the rightmost node in the subtree of the node;
    Replace the element in the node with the one in rightmost;
    The rightmost node is the node to be deleted now;
  }

  Obtain the path from the root to the node to be deleted;

  // Delete the last node in the path and propagate if needed
  deleteLastNodeInPath(path);

  size--; // After one element deleted
  return true; // Element deleted
}

/** Delete the last node from the path. */
public void deleteLastNodeInPath(ArrayList<TreeNode<E>> path) {
  Get the last node u in the path;
  Get parentOfu and grandparentOfu in the path;
  Get childOfu from u;
  Delete node u. Connect childOfu with parentOfu

  // Recolor the nodes and fix double black if needed
  if (childOfu == root || u.isRed())
    return; // Done if childOfu is root or if u is red
  else if (childOfu != null && childOfu.isRed())
    childOfu.setBlack(); // Set it black, done
  else // u is black, childOfu is null or black
    // Fix double black on parentOfu
    fixDoubleBlack(grandparentOfu, parentOfu, childOfu, path, i);
}

/** Fix the double black problem at node parent */
private void fixDoubleBlack(
    RBTreeNode<E> grandparent, RBTreeNode<E> parent,
    RBTreeNode<E> db, ArrayList<TreeNode<E>> path, int i) {
  Obtain y, y1, and y2

  if (y.isBlack() && y1 != null && y1.isRed()) {
    if (parent.right == db) {
      // Case 1.1: y is a left black sibling and y1 is red
      Restructure and recolor parent, y, and y1 to fix the problem;
    }
```

```
    else {
      // Case 1.3: y is a right black sibling and y1 is red
      Restructure and recolor parent, y1, and y to fix the problem;
    }
  }
  else if (y.isBlack() && y2 != null && y2.isRed()) {
    if (parent.right == db) {
      // Case 1.2: y is a left black sibling and y2 is red
      Restructure and recolor parent, y2, and y to fix the problem;
    }
    else {
      // Case 1.4: y is a right black sibling and y2 is red
      Restructure and recolor parent, y, and y2 to fix the problem;
    }
  }
  else if (y.isBlack()) {
    // Case 2: y is black and y's children are black or null
    Recolor y to red;

    if (parent.isRed())
      parent.setBlack(); // Done
    else if (parent != root) {
      // Propagate double black to the parent node
      // Fix new appearance of double black recursively
      db = parent;
      parent = grandparent;
      grandparent =
        (i >= 3) ? (RBTreeNode<E>)(path.get(i - 3)) : null;
      fixDoubleBlack(grandparent, parent, db, path, i - 1);
    }
  }
  else if (y.isRed()) {
    if (parent.right == db) {
      // Case 3.1: y is a left red child of parent
      parent.left = y2;
      y.right = parent;
    }
    else {
      // Case 3.2: y is a right red child of parent
      parent.right = y.left;
      y.left = parent;
    }

    parent.setRed(); // Color parent red
    y.setBlack(); // Color y black
    connectNewParent(grandparent, parent, y); // y is new parent
    fixDoubleBlack(y, parent, db, path, i - 1);
  }
}
```

### *<Side Remark: delete(E, e)>*

The delete(E e) method (lines 1-18) locates the node that contains e (line 2). If the node does not exist, return false (lines 3-4). If the node is an internal node, find the right most node in its left subtree and replace the element in the node with the element in the right most node (lines 6-9). Now the node to be deleted is an external node. Obtain

the path from the root to the node (line 11). Invoke deleteLastNodeInPath(path) to delete the last node in the path and ensure that the tree is still a red-black tree (line 14).

### *<Side Remark: deleteLastNodeInPath(path)>*

The deleteLastNodeInPath method (lines 21-35) obtains the last node u, parentOfu, grandparendOfu, and childOfu (lines 22-25). If childOfu is the root or u is red, the tree is fine (lines 28-29). If childOfu is red, color it black (lines 30-31). We are done. Otherwise, u is black and childOfu is null or black. Invoke fixDoubleBlack to eliminate the double black problem (line 34).

### *<Side Remark: fixDoubleBlack>*

The fixDoubleBlack method (lines 38-95) eliminates the double black problem. Obtain y, y1, and y2 (line 41). y is the sibling of the double black node. y1 and y2 are the left and right children of y. Consider three cases:

1. If y is black and one of the children of y is red, the double black problem can be fixed by one-time restructuring and recoloring in Case 1 (lines 43-62).
2. If y is black and the children of y is null or black, change y to red. If parent of y is black, denote parent to be the new double black node and invoke fixDoubleBlack recursively (line 76).
3. If y is red, adjust the nodes to make parent as child of y (lines 83, 88) and color parent red, and y black (lines 91-92). Make y the new parent (line 93). Recursively invoke fixDoubleBlack on the same double black node with a different color for parent (line 94).

### *<Side Remark: deletion example>*

Figure 10.28 shows the steps of deleting elements. To delete 50 from the tree in Figure 10.28(a), apply Case 1.2, as shown in Figure 10.28(b). After restructuring and recoloring, the new tree is shown in Figure 10.28(c).

When deleting 20 in Figure 10.28(c), 20 is an internal node and it is replaced by 16, as shown in Figure 10.28(d). Now Case 2 applies to deleting the rightmost node, as shown in Figure 10.28(e). Recolor the nodes results in a new tree, as shown in Figure 10.28(f).

When deleting 15, connect node 3 with node 20 and color node 3 black, as shown in Figure 10.28(g). We are done.

After deleting 25, the new tree is shown in Figure 10.28(j). Now delete 16. Apply Case 2, as shown in Figure 10.28(k). The new tree is shown in Figure 10.29(l).

After deleting 34, the new tree is shown in Figure 10.28(m).

After deleting 27, the new tree is shown in Figure 10.28(n).

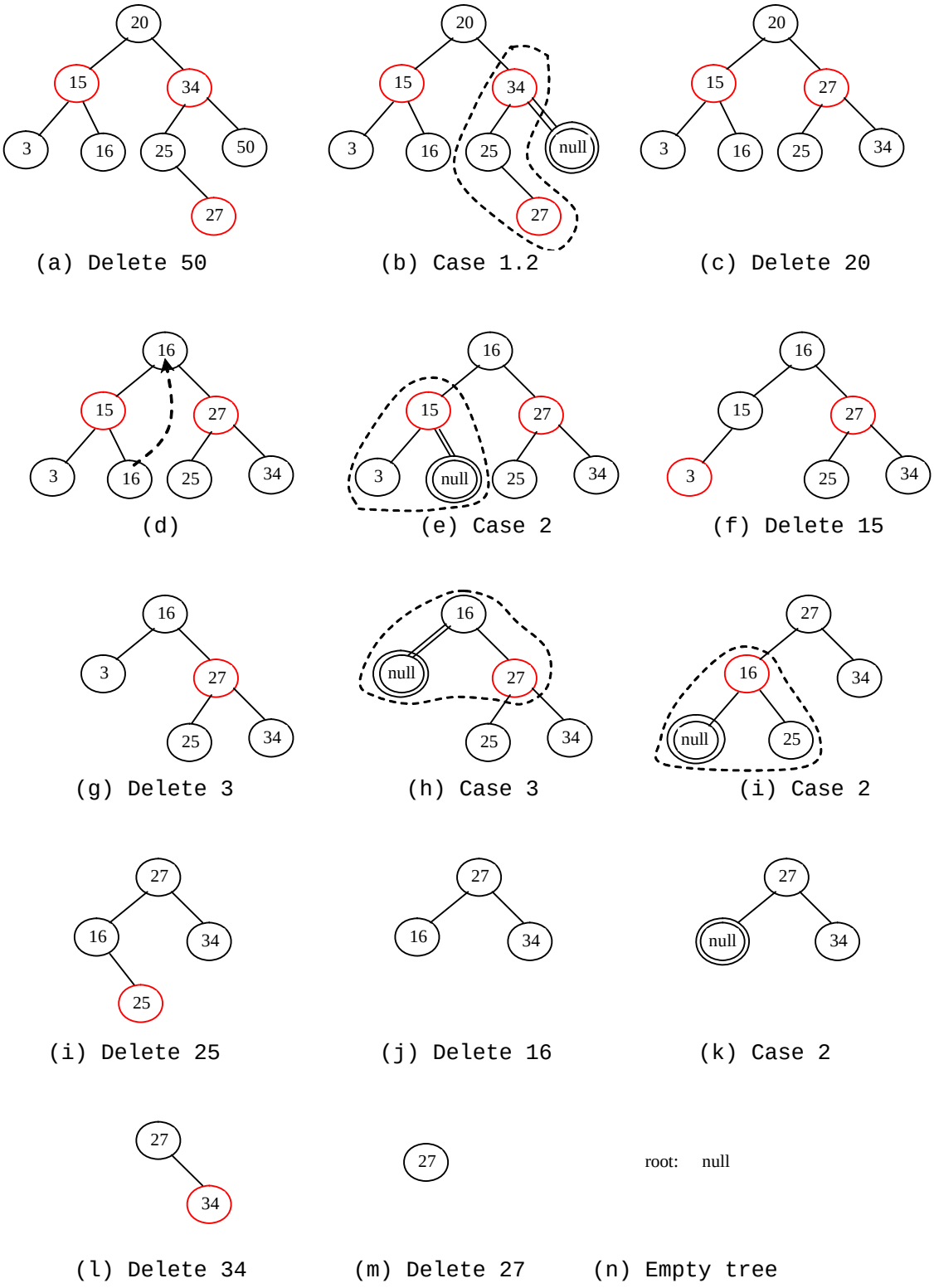



**Figure 10.28**
*Delete elements from a red-black tree.*

## 10.6 Implementing RBTree Class

Listing 10.3 gives a complete implementation for the RBTree class.

**Listing 10.3 BRTree.java**

***PD: Please add line numbers in the following code***
<Side Remark line 1: delete e from tree>
<Side Remark line 2: locate the node>
<Side Remark line 4: element not found>
<Side Remark line 6: internal element?>
<Side Remark line 7: rightmost node>
<Side Remark line 12: path to external node>
<Side Remark line 15: delete the node>
<Side Remark line 17: one element deleted>
<Side Remark line 18: deletion successful>

<Side Remark line 22: delete a node>
<Side Remark line 23: u>
<Side Remark line 24: parentOfu, grandparentOfu>
<Side Remark line 23: childOfu>
<Side Remark line 26: delete u>
<Side Remark line 30: done>
<Side Remark line 32: set childOfu black>
<Side Remark line 35: fix double black>

<Side Remark line 39: fix double black>
<Side Remark line 42: y, y1, y2>
<Side Remark line 47: process Case 1.1>
<Side Remark line 51: process Case 1.3>
<Side Remark line 57: process Case 1.2>
<Side Remark line 61: process Case 1.4>
<Side Remark line 66: process Case 2>
<Side Remark line 77: propagate double black>
<Side Remark line 83: process Case 3.1>
<Side Remark line 88: process Case 3.2>
<Side Remark line 95: fix double black>

```
import java.util.ArrayList;

public class RBTree<E extends Comparable<E>> extends BinaryTree<E> {
  /** Create a default RB tree */
  public RBTree() {
  }

  /** Create an RB tree from an array of elements */
  public RBTree(E[] elements) {
    super(elements);
  }

  /** Override createNewNode to create an RBTreeNode */
  protected RBTreeNode<E> createNewNode(E e) {
    return new RBTreeNode<E>(e);
  }
```

```
  /** Override the insert method to balance the tree if necessary */
  public boolean insert(E e) {
    boolean successful = super.insert(e);
    if (!successful)
      return false; // e is already in the tree
    else {
      ensureRBTree(e);
    }

    return true; // e is inserted
  }

  /** Ensure that the tree is a red-black tree */
  private void ensureRBTree(E e) {
    // Get the path that leads to element e from the root
    ArrayList<TreeNode<E>> path = path(e);

    int i = path.size() - 1; // Index to the current node in the path

    // u is the last node in the path. u contains element e
    RBTreeNode<E> u = (RBTreeNode<E>)(path.get(i));

    // v is the parent of of u, if exists
    RBTreeNode<E> v = (u == root) ? null :
      (RBTreeNode<E>)(path.get(i - 1));

    u.setRed(); // It is OK to set u red

    if (u == root) // If e is inserted as the root, set root black
      u.setBlack();
    else if (v.isRed())
      fixDoubleRed(u, v, path, i); // Fix double red violation at u
  }

  /** Fix double red violation at node u */
  private void fixDoubleRed(RBTreeNode<E> u, RBTreeNode<E> v,
      ArrayList<TreeNode<E>> path, int i) {
    // w is the grandparent of u
    RBTreeNode<E> w = (RBTreeNode<E>)(path.get(i - 2));
    RBTreeNode<E> parentOfw = (w == root) ? null :
      (RBTreeNode<E>)path.get(i - 3);

    // Get v's sibling named x
    RBTreeNode<E> x = (w.left == v) ?
      (RBTreeNode<E>)(w.right) : (RBTreeNode<E>)(w.left);

    if (x == null || x.isBlack()) {
      // Case 1: v's sibling x is black
      if (w.left == v && v.left == u) {
        // Case 1.1: u < v < w, Restructure and recolor nodes
        restructureRecolor(u, v, w, w, parentOfw);

        w.left = v.right; // v.right is y3 in Figure 10.7
        v.right = w;
      }
      else if (w.left == v && v.right == u) {
```

```
        // Case 1.2: v < u < w, Restructure and recolor nodes
        restructureRecolor(v, u, w, w, parentOfw);
        v.right = u.left;
        w.left = u.right;
        u.left = v;
        u.right = w;
      }
      else if (w.right == v && v.right == u) {
        // Case 1.3: w < v < u, Restructure and recolor nodes
        restructureRecolor(w, v, u, w, parentOfw);
        w.right = v.left;
        v.left = w;
      }
      else {
        // Case 1.4: w < u < v, Restructure and recolor nodes
        restructureRecolor(w, u, v, w, parentOfw);
        w.right = u.left;
        v.left = u.right;
        u.left = w;
        u.right = v;
      }
    }
    else { // Case 2: v's sibling x is red
      // Recolor nodes
      w.setRed();
      u.setRed();
      ((RBTreeNode<E>)(w.left)).setBlack();
      ((RBTreeNode<E>)(w.right)).setBlack();

      if (w == root) {
        w.setBlack();
      }
      else if (((RBTreeNode<E>)parentOfw).isRed()) {
        // Propagate along the path to fix new double red violation
        u = w;
        v = (RBTreeNode<E>)parentOfw;
        fixDoubleRed(u, v, path, i - 2); // i - 2 propagates upward
      }
    }
  }

  /** Connect b with parentOfw and recolor a, b, c for a < b < c */
  private void restructureRecolor(RBTreeNode<E> a, RBTreeNode<E> b,
      RBTreeNode<E> c, RBTreeNode<E> w, RBTreeNode<E> parentOfw) {
    if (parentOfw == null)
      root = b;
    else if (parentOfw.left == w)
      parentOfw.left = b;
    else
      parentOfw.right = b;

    b.setBlack(); // b becomes the root in the subtree
    a.setRed(); // a becomes the left child of b
    c.setRed(); // c becomes the right child of b
  }
```

```
  /** Delete an element from the RBTree.
   * Return true if the element is deleted successfully
   * Return false if the element is not in the tree */
  public boolean delete(E e) {
    // Locate the node to be deleted
    TreeNode<E> current = root;
    while (current != null) {
      if (e.compareTo(current.element) < 0) {
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        current = current.right;
      }
      else
        break; // Element is in the tree pointed by current
    }

    if (current == null)
      return false; // Element is not in the tree

    java.util.ArrayList<TreeNode<E>> path;

    // current node is an internal node
    if (current.left != null && current.right != null) {
      // Locate the rightmost node in the left subtree of current
      TreeNode<E> rightMost = current.left;
      while (rightMost.right != null) {
        rightMost = rightMost.right; // Keep going to the right
      }

      path = path(rightMost.element); // Get path before replacement

      // Replace the element in current by the element in rightMost
      current.element = rightMost.element;
    }
    else
      path = path(e); // Get path to current node

    // Delete the last node in the path and propagate if needed
    deleteLastNodeInPath(path);

    size--; // After one element deleted
    return true; // Element deleted
  }

  /** Delete the last node from the path. */
  public void deleteLastNodeInPath(ArrayList<TreeNode<E>> path) {
    int i = path.size() - 1; // Index to the node in the path
    // u is the last node in the path
    RBTreeNode<E> u = (RBTreeNode<E>)(path.get(i));
    RBTreeNode<E> parentOfu = (u == root) ? null :
      (RBTreeNode<E>)(path.get(i - 1));
    RBTreeNode<E> grandparentOfu = (parentOfu == null ||
      parentOfu == root) ? null :
      (RBTreeNode<E>)(path.get(i - 2));
    RBTreeNode<E> childOfu = (u.left == null) ?
```

```
      (RBTreeNode<E>)(u.right) : (RBTreeNode<E>)(u.left);

    // Delete node u. Connect childOfu with parentOfu
    connectNewParent(parentOfu, u, childOfu);

    // Recolor the nodes and fix double black if needed
    if (childOfu == root || u.isRed())
      return; // Done if childOfu is root or if u is red
    else if (childOfu != null && childOfu.isRed())
      childOfu.setBlack(); // Set it black, done
    else // u is black, childOfu is null or black
      // Fix double black on parentOfu
      fixDoubleBlack(grandparentOfu, parentOfu, childOfu, path, i);
  }

  /** Fix the double black problem at node parent */
  private void fixDoubleBlack(
      RBTreeNode<E> grandparent, RBTreeNode<E> parent,
      RBTreeNode<E> db, ArrayList<TreeNode<E>> path, int i) {
    // Obtain y, y1, and y2
    RBTreeNode<E> y = (parent.right == db) ?
      (RBTreeNode<E>)(parent.left) : (RBTreeNode<E>)(parent.right);
    RBTreeNode<E> y1 = (RBTreeNode<E>)(y.left);
    RBTreeNode<E> y2 = (RBTreeNode<E>)(y.right);

    if (y.isBlack() && y1 != null && y1.isRed()) {
      if (parent.right == db) {
        // Case 1.1: y is a left black sibling and y1 is red
        connectNewParent(grandparent, parent, y);
        recolor(parent, y, y1); // Adjust colors

        // Adjust child links
        parent.left = y.right;
        y.right = parent;
      }
      else {
        // Case 1.3: y is a right black sibling and y1 is red
        connectNewParent(grandparent, parent, y1);
        recolor(parent, y1, y); // Adjust colors

        // Adjust child links
        parent.right = y1.left;
        y.left = y1.right;
        y1.left = parent;
        y1.right = y;
      }
    }
    else if (y.isBlack() && y2 != null && y2.isRed()) {
      if (parent.right == db) {
        // Case 1.2: y is a left black sibling and y2 is red
        connectNewParent(grandparent, parent, y2);
        recolor(parent, y2, y); // Adjust colors

        // Adjust child links
        y.right = y2.left;
        parent.left = y2.right;
```

```
        y2.left = y;
        y2.right = parent;
      }
      else {
        // Case 1.4: y is a right black sibling and y2 is red
        connectNewParent(grandparent, parent, y);
        recolor(parent, y, y2); // Adjust colors

        // Adjust child links
        y.left = parent;
        parent.right = y1;
      }
    }
    else if (y.isBlack()) {
      // Case 2: y is black and y's children are black or null
      y.setRed(); // Change y to red
      if (parent.isRed())
        parent.setBlack(); // Done
      else if (parent != root) {
        // Propagate double black to the parent node
        // Fix new appearance of double black recursively
        db = parent;
        parent = grandparent;
        grandparent =
          (i >= 3) ? (RBTreeNode<E>)(path.get(i - 3)) : null;
        fixDoubleBlack(grandparent, parent, db, path, i - 1);
      }
    }
    else { // y.isRed()
      if (parent.right == db) {
        // Case 3.1: y is a left red child of parent
        parent.left = y2;
        y.right = parent;
      }
      else {
        // Case 3.2: y is a right red child of parent
        parent.right = y.left;
        y.left = parent;
      }

      parent.setRed(); // Color parent red
      y.setBlack(); // Color y black
      connectNewParent(grandparent, parent, y); // y is new parent
      fixDoubleBlack(y, parent, db, path, i - 1);
    }
  }

  /** Recolor parent, newParent, and c. Case 1 removal */
  private void recolor(RBTreeNode<E> parent,
      RBTreeNode<E> newParent, RBTreeNode<E> c) {
    // Retain the parent's color for newParent
    if (parent.isRed())
      newParent.setRed();
    else
      newParent.setBlack();
```

```
    // c and parent become the children of newParent, set them black
    parent.setBlack();
    c.setBlack();
  }

  /** Connect newParent with grandParent */
  private void connectNewParent(RBTreeNode<E> grandparent,
      RBTreeNode<E> parent, RBTreeNode<E> newParent) {
    if (parent == root) {
      root = newParent;
      if (root != null)
        newParent.setBlack();
    }
    else if (grandparent.left == parent)
      grandparent.left = newParent;
    else
      grandparent.right = newParent;
  }

  /** Preorder traversal from a subtree */
  protected void preorder(TreeNode<E> root) {
    if (root == null) return;
    System.out.print(root.element +
      (((RBTreeNode<E>)root).isRed() ? " (red) " : " (black) "));
    preorder(root.left);
    preorder(root.right);
  }

  /** RBTreeNode is TreeNode plus color indicator */
  protected static class RBTreeNode<E extends Comparable<E>> extends
      BinaryTree.TreeNode<E> {
    private boolean red = true; // Indicate node color

    public RBTreeNode(E e) {
      super(e);
    }

    public boolean isRed() {
      return red;
    }

    public boolean isBlack() {
      return!red;
    }

    public void setBlack() {
      red = false;
    }

    public void setRed() {
      red = true;
    }

    int blackHeight;
  }
}
```

### *<Side Remark: constructors>*

The RBTree class extends BinaryTree. Like the BinaryTree class, the RBTree class has a no-arg constructor that constructs an empty RBTree (lines 7-8) and a constructor that creates an initial RBTree from an array of elements (lines 11-13).

### *<Side Remark: createNewNode()>*

The createNewNode() method defined in the BinaryTree class creates a TreeNode. This method is overridden to return an RBTreeNode (lines 16-18). This method is invoked in the insert method in BinaryTree to create a node.

### *<Side Remark: insert>*

The insert method in RBTree is overridden in lines 21-30. The method first invokes the insert method in BinaryTree, and then invokes ensureRBTree(e) (line 26) to ensure that tree is still a red-black tree after inserting a new element.

### *<Side Remark: ensureRBTree>*

The ensureRBTree(E e) method first obtains the path of nodes that lead to element e from the root (line 35). It obtains u and v (the parent of u) from the path. If u is the root, color u black (lines 48-49). If v is red, invoke fixDoubleRed to fix the double red on both u and v (lines 50-51).

### *<Side Remark: fixDoubleRed>*

The fixDoubleRed(u, v, path, i) method fixes the double red violation at node u. The method first obtains w (the grandparent of u from the path) (line 58), parentOfw if exists (lines 59-60), and x (the sibling of v) (lines 63-64). If x is null or black, consider four subcases to fix the double red problem (lines 68-96). If x is null or black, consider four subcases to fix the double red problem (lines 68-96). If x is red, color w and u red and color w's two children black (lines 100-103). If w is the root, color w black (lines 105-107). Otherwise, propagate along the path to fix the new double red violation (lines 110-112).

### *<Side Remark: delete>*

The delete(E e) method in RBTree is overridden in lines 135-175. The method locates the node that contains e (lines 137-147). If the node is null, no element is found (lines 149-150). The method considers two cases:

- If the node is internal, find the rightmost node in its left subtree (lines 157-160). Obtain a path from the root to the rightmost node (line 162), and replace the element in the node with the element in the rightmost node (line 165).

- If the node is external, obtain the path from the root to the node (line 168).

The last node in the path is the node to be deleted. Invoke deleteLastNodeInPath(path) to delete it and ensure the tree is a red-black after the node is deleted (line 171).

### *<Side Remark: deleteLastNodeInPath>*

The deleteLastNodeInPath(path) method first obtains u, parentOfu, grandparendOfu, and childOfu. u is the last node in the path. Connect childOfu as a child of parentOfu (line 191). This in effect deletes u from the tree. Consider three cases:

- If childOfu is the root or childOfu is red, we are done (lines 194-195).
- Otherwise, if childOfu is red, color it (lines 196-197).
- Otherwise, invoke fixDoubleBlack to fix the double black problem on childOfu (line 200).

***<Side Remark: fixDoubleBlack>***

The fixDoubleBlack method first obtains y, y1, and y2. y is the sibling of the first double black node, and y1 and y2 are the left and right children of y. Consider three cases:

- If y is black and y1 or y2 is red, fix the double black problem for Case 1 (lines 214-256).

- Otherwise, if y is black, fix the double black problem for Case 2 by recoloring the nodes. If parent is black and not a root, propagate double black to parent and recursively invoke fixDoubleBlack (lines 265-269).

- Otherwise, y is red. In this case, adjust the nodes to make parent the child of y (lines 276-281). Invoke fixDoubleBlack with the adjusted nodes (line 287) to fix the double black problem.

***<Side Remark: preorder>***

The preorder(TreeNode<E> root) method is overridden to display the node colors (lines 320-326).

## 10.7 Testing the RBTree Class

Listing 10.4 gives a test program. The program creates an RBTree initialized with an array of integers 34, 3, and 50 (lines 6-7), inserts elements in lines 10-22, and deletes elements in lines 25-46.

**Listing 10.4 TestRBTree.java**

******PD: Please add line numbers in the following code******
******Layout: Please layout exactly. Don't skip the space. This is true for all source code in the book. Thanks, AU.***
***<Side Remark line 7: create an RBTree>***
***<Side Remark line 10: insert 20>***
***<Side Remark line 13: insert 15>***
***<Side Remark line 16: insert 16>***
***<Side Remark line 19: insert 25>***
***<Side Remark line 22: insert 27>***
***<Side Remark line 25: delete 50>***
***<Side Remark line 28: delete 20>***
***<Side Remark line 31: delete 15>***
***<Side Remark line 34: delete 3>***
***<Side Remark line 37: delete 25>***
***<Side Remark line 40: delete 16>***
***<Side Remark line 43: delete 34>***

***<Side Remark line 46: delete 27>***

```
public class TestRBTree {
  public static void main(String[] args) {
    // Create an RB tree
    RBTree<Integer> tree =
      new RBTree<Integer>(new Integer[]{34, 3, 50});
    printTree(tree);

    tree.insert(20);
    printTree(tree);

    tree.insert(15);
    printTree(tree);

    tree.insert(16);
    printTree(tree);

    tree.insert(25);
    printTree(tree);

    tree.insert(27);
    printTree(tree);

    tree.delete(50);
    printTree(tree);

    tree.delete(20);
    printTree(tree);

    tree.delete(15);
    printTree(tree);

    tree.delete(3);
    printTree(tree);

    tree.delete(25);
    printTree(tree);

    tree.delete(16);
    printTree(tree);

    tree.delete(34);
    printTree(tree);

    tree.delete(27);
    printTree(tree);
  }

  public static void printTree(BinaryTree tree) {
    // Traverse tree
    System.out.print("\nInorder (sorted): ");
    tree.inorder();
    System.out.print("\nPostorder: ");
    tree.postorder();
    System.out.print("\nPreorder: ");
```

```
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());
    System.out.println();
  }
}
```

## <Output>

```
Inorder (sorted): 3 34 50
Postorder: 3 50 34
Preorder: 34 (black) 3 (red) 50 (red)
The number of nodes is 3

Inorder (sorted): 3 20 34 50
Postorder: 20 3 50 34
Preorder: 34 (black) 3 (black) 20 (red) 50 (black)
The number of nodes is 4

Inorder (sorted): 3 15 20 34 50
Postorder: 3 20 15 50 34
Preorder: 34 (black) 15 (black) 3 (red) 20 (red) 50 (black)
The number of nodes is 5

Inorder (sorted): 3 15 16 20 34 50
Postorder: 3 16 20 15 50 34
Preorder: 34 (black) 15 (red) 3 (black) 20 (black) 16 (red) 50 (black)
The number of nodes is 6

Inorder (sorted): 3 15 16 20 25 34 50
Postorder: 3 16 25 20 15 50 34
Preorder: 34 (black) 15 (red) 3 (black) 20 (black) 16 (red) 25 (red)
  50 (black)
The number of nodes is 7

Inorder (sorted): 3 15 16 20 25 27 34 50
Postorder: 3 16 15 27 25 50 34 20
Preorder: 20 (black) 15 (red) 3 (black) 16 (black) 34 (red) 25 (black)
  27 (red) 50 (black)
The number of nodes is 8

Inorder (sorted): 3 15 16 20 25 27 34
Postorder: 3 16 15 25 34 27 20
Preorder: 20 (black) 15 (red) 3 (black) 16 (black) 27 (red)
  25 (black) 34 (black)
The number of nodes is 7

Inorder (sorted): 3 15 16 25 27 34
Postorder: 3 15 25 34 27 16
Preorder: 16 (black) 15 (black) 3 (red) 27 (red) 25 (black) 34 (black)
The number of nodes is 6

Inorder (sorted): 3 16 25 27 34
Postorder: 3 25 34 27 16
Preorder: 16 (black) 3 (black) 27 (red) 25 (black) 34 (black)
The number of nodes is 5
```

```
Inorder (sorted): 16 25 27 34
Postorder: 25 16 34 27
Preorder: 27 (black) 16 (black) 25 (red) 34 (black)
The number of nodes is 4

Inorder (sorted): 16 27 34
Postorder: 16 34 27
Preorder: 27 (black) 16 (black) 34 (black)
The number of nodes is 3

Inorder (sorted): 27 34
Postorder: 34 27
Preorder: 27 (black) 34 (red)
The number of nodes is 2

Inorder (sorted): 27
Postorder: 27
Preorder: 27 (black)
The number of nodes is 1

Inorder (sorted):
Postorder:
Preorder:
The number of nodes is 0
```
***<End Output>***

Figure 10.14 shows how the tree evolves as elements are added to the tree and Figure 10.28 shows how the tree evolves as elements are deleted from the tree.

## 10.8 Performance of the RBTree Class

***<Side Remark:*** $2\log n$ ***height>***

The search, insertion, and deletion time in a red-black tree is dependent on the height of the tree. A red-black tree corresponds to a 2-4 tree. When you convert a node in a 2-4 tree to red-black tree nodes, it produces one black node, zero, one, or two red nodes as its children, depending on whether the original node is a 2-node, 3-node, or 4-node. So, the height of a red-black tree is at most as twice as its corresponding 2-4 tree. Since the height of a 2-4 tree is $\log n$, the height of a red-black tree is $2\log n$.

***<Side Remark: red-black vs. AVL>***

A red-black tree has the same time complexity as an AVL tree, as shown in Table 10.1. In general, a red-black is more efficient than an AVL tree, because a red-black tree requires only one time restructuring of the nodes for insert and delete operations.

***<Side Remark: red-black vs. 2-4>***

A red-black tree has the same time complexity as a 2-4 tree, as shown in Table 10.1. In general, a red-black is more efficient than a 2-4 tree because of the following two reasons:

1. A red-black tree requires only one time restructuring of the nodes for insert and delete operations. However, a 2-4 tree may require

many splits for an insert operation and fusion for a delete operation.

2. A red-black tree is a binary tree. A binary tree can be implemented more efficiently than a 2-4 tree, because a node in a 2-4 tree has maximum three elements and fours children. Space is wasted for 2-nodes and 3-nodes in a 2-4 tree.

*Table 10.1*
Time Complexities for Methods in RBTree, AVLTree, and Tree234

| Mehtods | Red-Black Tree | AVL Tree | 2-3-4 Tree |
|---|---|---|---|
| search(e: E) | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| insert(e: E) | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| delete(e: E) | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| getSize() | $O(1)$ | $O(1)$ | $O(1)$ |
| isEmpty() | $O(1)$ | $O(1)$ | $O(1)$ |

Listing 10.5 gives an empirical test on the performance of AVL trees, 2-4 trees, and red-black trees.

**Listing 25.8 SetPerformanceTest.java**

***PD: Please add line numbers in the following code***
<Side Remark line 8: an AVL tree>
<Side Remark line 13: a 2-4 tree>
<Side Remark line 18: a red-black tree>
<Side Remark line 23: start time>
<Side Remark line 31: shuffle>
<Side Remark line 35: add to tree>
<Side Remark line 37: shuffle>
<Side Remark line 41: remove from container>
<Side Remark line 43: end time>
<Side Remark line 44: return elapse time>

```
public class TreePerformanceTest {
  public static void main(String[] args) {
    final int TEST_SIZE = 500000; // Tree size used in the test

    // Create an AVL tree
    Tree<Integer> tree1 = new AVLTree<Integer>();
    System.out.println("AVL tree time: " +
      getTime(tree1, TEST_SIZE) + " milliseconds");

    // Create a 2-4 tree
    Tree<Integer> tree2 = new Tree234<Integer>();
    System.out.println("2-4 tree time: "
      + getTime(tree2, TEST_SIZE) + " milliseconds");

    // Create a red-black tree
    Tree<Integer> tree3 = new RBTree<Integer>();
    System.out.println("RB tree time: "
      + getTime(tree3, TEST_SIZE) + " milliseconds");
  }
```

```
  public static long getTime(Tree<Integer> tree, int testSize) {
    long startTime = System.currentTimeMillis(); // Start time

    // Create a list to store distinct integers
    java.util.List<Integer> list = new java.util.ArrayList<Integer>();
    for (int i = 0; i < testSize; i++)
      list.add(i);

    java.util.Collections.shuffle(list); // Shuffle the list

    // Insert elements in the list to the tree
    for (int i = 0; i < testSize; i++)
      tree.insert(list.get(i));

    java.util.Collections.shuffle(list); // Shuffle the list

    // Delete elements in the list from the tree
    for (int i = 0; i < testSize; i++)
      tree.delete(list.get(i));

    // Return elapse time
    return System.currentTimeMillis() - startTime;
  }
}
```

***<Output>***

```
AVL tree time: 7609 milliseconds
2-4 tree time: 8594 milliseconds
RB tree time: 5515 milliseconds
```

***<End Output>***

The getTestTime method creates a list of distinct integers from 0 to testSize – 1 (lines 27-29), shuffles the list (line 31), adds the elements from the list to a tree (lines 34-35), shuffles the list again (line 37), removes the elements from the tree (lines 40-41), and finally returns the execution time (line 44).

The program creates an AVL (line 8), a 2-4 tree (line 13), and a red-black tree (line 18). The program obtains the execution time for adding and removing 500000 elements in the three trees.

***<Side Remark: red-black tree best>***
As you see, the red-black tree performs the best, followed by the AVL tree.

NOTE:
***<Side Remark: java.util.TreeSet>***
The java.util.TreeSet class in the Java API is implemented using a red-black tree. Each entry in the set is stored in the tree. Since search, insert, and delete methods in a red-black tree take $O(\log n)$ time, the get, add, remove, and contains methods in java.util.TreeSet take $O(\log n)$ time.

NOTE:
***<Side Remark: java.util.TreeMap>***

The java.util.TreeMap class in the Java API is implemented using a red-black tree. Each entry in the map is stored in the tree. The order of the entries is determined by their keys. Since search, insert, and delete methods in a red-black tree take $O(\log n)$ time, the get, put, remove, and containsKey methods in java.util.TreeMap take $O(\log n)$ time.

## Key Terms

***PD: Please place terms in two columns same as in intro6e.

- black depth
- double black violation
- double red violation
- external node
- red-black tree

## Chapter Summary

- A red-black tree is a binary search tree, which is derived from a *2-4 tree*. A red-black tree corresponds to a 2-4 tree. You can convert a red-black tree to a 2-4 tree or vice versa.
- In a red-black tree, each node is color red or black. The root is always black. Two adjacent nodes cannot be both red. All external nodes have the same black depth.
- Since a red-black tree is a binary search tree, the RBTree class extends the BinaryTree class.
- Searching an element in a 2-4 tree is similar to searching an element in a binary tree. The difference is that you have search an element within a node.
- To insert an element to a 2-4 tree, locate a leaf node in which the element will be inserted. If the leaf node is a 2-node or 3-node, simply insert the element into the node. If the node is a 4-node, split the node.
- The process of deleting an element from a 2-4 tree is similar to deleting an element from a binary tree. The difference is that you have to perform transfer or fusion operations for empty nodes.
- The height of an 2-4 tree is O(logn). So, the time complexity for the search, insert, and delete methods are O(logn).

## Review Questions

### *Sections 10.1-10.2*

10.1
What is a red-black tree? What is an external node? What is black-depth?

10.2
Describe the properties of a red-black tree.

10.3
How do you convert a red-black tree to a 2-4 tree? Is the conversion unique?

10.4
How do you convert a 2-4 tree to a red-black tree? Is the conversion unique?

### *Sections 10.3-10.5*

10.5
What are the data fields in RBTreeNode?

10.6
How do you insert an element into a red-black tree and how do you fix the double red problem?

10.7
How do you delete an element from a red-black tree and how do you fix the double black problem?

10.8
Show the change of the tree when inserting 1, 2, 3, 4, 10, 9, 7, 5, 8, 6 into the tree, in this order.

10.9
For the tree built in the preceding question, show the change of the tree after deleting 1, 2, 3, 4, 10, 9, 7, 5, 8, 6 from the tree in this order.

## Programming Exercises

10.1*
(*red-black tree to 2-4 tree*) Write a program that converts a red-black tree to a 2-4 tree.

10.2*
(*2-4 tree to red-black tree*) Write a program that converts a red-black tree to a 2-4 tree.

10.3***
(*red-black tree animation*) Write a Java applet that animates the red-black tree insert, delete, and search methods, as shown in Figure 10.7.

10.4**
(*Parent reference for RBTree*) Suppose that the TreeNode class defined in BinaryTree contains a reference to the node's parent, as shown in Exercise 7.17. Implement the RBTree class to support this change. Write a test program that adds numbers 1, 2, ..., 100 to the tree, and displays the paths for all leaf nodes.