WORKED EXAMPLE 17.2 Implementing a Red-Black Tree

Problem Statement Implement a red-black tree using the algorithm for adding and removing elements from Section 17.5. Read that section first if you have not done so already.

The Node Implementation

The nodes of the red-black tree need to store the “color”, which we represent as the cost of traversing the node:

```
static final int BLACK = 1;
static final int RED = 0;
private static final int NEGATIVE_RED = -1;
private static final int DOUBLE_BLACK = 2;

static class Node
{
   public Comparable data;
   public Node left;
   public Node right;
   public Node parent;
   public int color;
   . . .
}
```

The first two color constants and the `Node` class have package visibility. We will add a test class to the same package, which is discussed later in this worked example.

Nodes in a red-black tree also have a link to the parent. When adding or moving a node, it is important that the parent and child links are synchronized. Because this synchronization is tedious and error-prone, we provide several helper methods:

```
public class RedBlackTree
{
   . . .
   static class Node
   {
      . . .

      /**
         Sets the left child and updates its parent reference.
         @param child the new left child
      */
      public void setLeftChild(Node child)
      {
         left = child;
         if (child != null) { child.parent = this; }
      }

      /**
         Sets the right child and updates its parent reference.
         @param child the new right child
      */
      public void setRightChild(Node child)
      {
         right = child;
         if (child != null) { child.parent = this; }
      }
      }

      /**
         Updates the parent's and replacement node's links when a node is replaced.
         Also updates the root reference if the root is replaced.
         @param toBeReplaced the node that is to be replaced
         @param replacement the node that replaces that node
      */
      private void replaceWith(Node toBeReplaced, Node replacement)
      {
         if (toBeReplaced.parent == null)
         {
            replacement.parent = null;
            root = replacement;
         }
         else if (toBeReplaced == toBeReplaced.parent.left)
         {
            toBeReplaced.parent.setLeftChild(replacement);
         }
         else
         {
            toBeReplaced.parent.setRightChild(replacement);
         }
      }
   }
```

Insertion

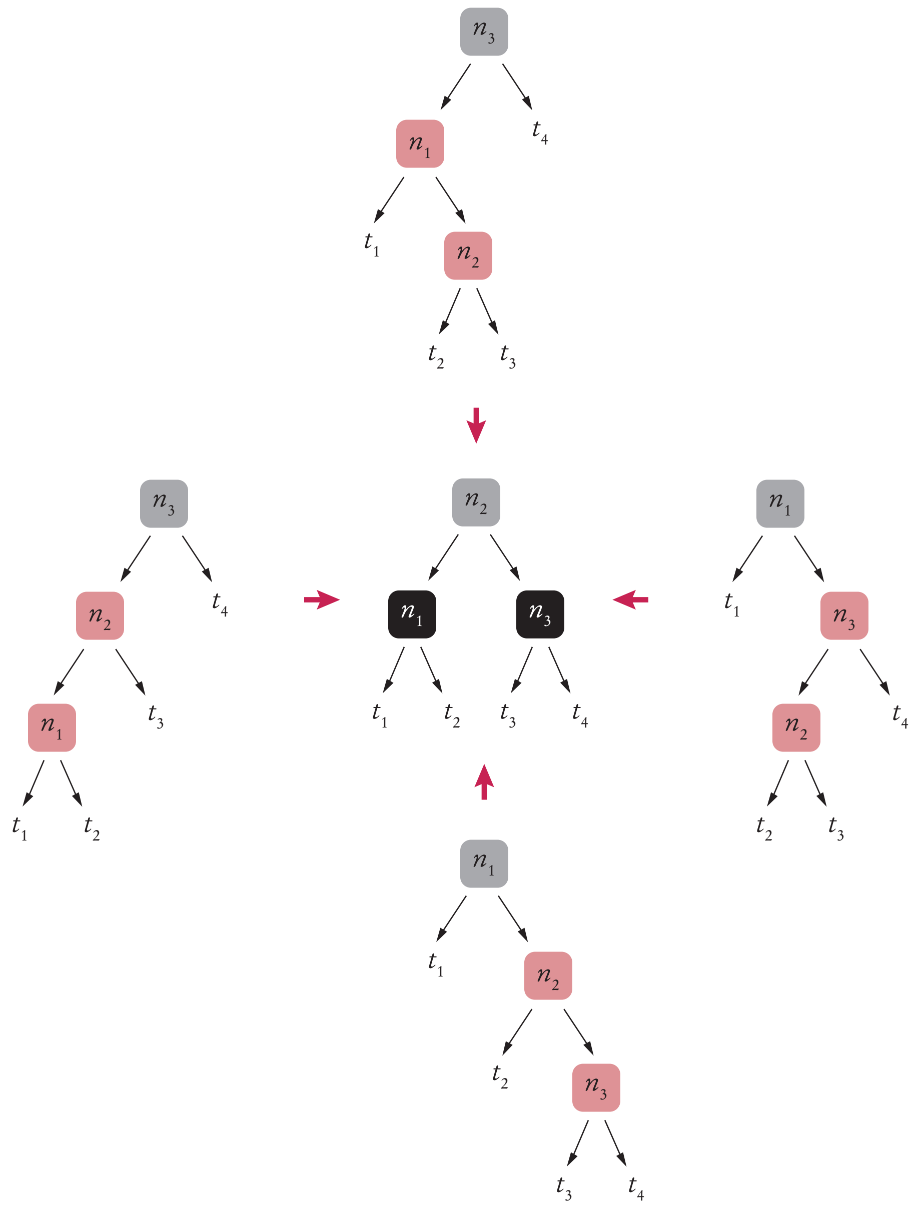
Insertion is handled as it is in a binary search tree. We insert a red node. Afterward, we call a method that fixes up the tree so it is a red-black tree again:

```
   public void add(Comparable obj)
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.left = null;
      newNode.right = null;
      if (root == null) { root = newNode; }
      else { root.addNode(newNode); }
      fixAfterAdd(newNode);
   }
```

If the inserted node is the root, it is turned black. Otherwise, we fix up any double-red violations:

```
   /**
      Restores the tree to a red-black tree after a node has been added.
      @param newNode the node that has been added
   */
   private void fixAfterAdd(Node newNode)
   {
      if (newNode.parent == null)
      {
         newNode.color = BLACK;
      }
      else
      {
         newNode.color = RED;
         if (newNode.parent.color == RED) { fixDoubleRed(newNode); }
      }
   }
```

The code for fixing up a double-red violation is quite long. Recall that there are four possible arrangements of the double red nodes:



In each case, we must sort the nodes and their children. Once we have the seven references `n1`, `n2`, `n3`, `t1`, `t2`, `t3`, and `t4`, the remainder of the procedure is straightforward. We build the replacement tree, change the reds to black, and subtract one from the color of the grandparent (which might be a double-black node when this method is called during node removal).

If we find that we introduced another double-red violation, we continue fixing it. Eventually, the violation is removed, or we reach the root, in which case the root is simply colored black:

```
/**
   Fixes a “double red” violation.
   @param child the child with a red parent
```

```
*/
private void fixDoubleRed(Node child)
{
   Node parent = child.parent;
   Node grandParent = parent.parent;
   if (grandParent == null) { parent.color = BLACK; return; }
   Node n1, n2, n3, t1, t2, t3, t4;
   if (parent == grandParent.left)
   {
      n3 = grandParent; t4 = grandParent.right;
      if (child == parent.left)
      {
         n1 = child; n2 = parent;
         t1 = child.left; t2 = child.right; t3 = parent.right;
      }
      else
      {
         n1 = parent; n2 = child;
         t1 = parent.left; t2 = child.left; t3 = child.right;
      }
   }
   else
   {
      n1 = grandParent; t1 = grandParent.left;
      if (child == parent.left)
      {
         n2 = child; n3 = parent;
         t2 = child.left; t3 = child.right; t4 = parent.right;
      }
      else
      {
         n2 = parent; n3 = child;
         t2 = parent.left; t3 = child.left; t4 = child.right;
      }
   }

   replaceWith(grandParent, n2);
   n1.setLeftChild(t1);
   n1.setRightChild(t2);
   n2.setLeftChild(n1);
   n2.setRightChild(n3);
   n3.setLeftChild(t3);
   n3.setRightChild(t4);
   n2.color = grandParent.color - 1;
   n1.color = BLACK;
   n3.color = BLACK;

   if (n2 == root)
   {
      root.color = BLACK;
   }
   else if (n2.color == RED && n2.parent.color == RED)
   {
      fixDoubleRed(n2);
   }
}
```

Removal

We remove a node in the same way as in a binary search tree. However, before removing it, we want to make sure that it is colored red. There are two cases for removal, removing an element with one child and removing the successor of an element with two children. Both branches must be modified:

```
public void remove(Comparable obj)
{
   // Find node to be removed

   Node toBeRemoved = root;
   boolean found = false;
   while (!found && toBeRemoved != null)
   {
      int d = toBeRemoved.data.compareTo(obj);
      if (d == 0) { found = true; }
      else
      {
         if (d > 0) { toBeRemoved = toBeRemoved.left; }
         else { toBeRemoved = toBeRemoved.right; }
      }
   }

   if (!found) { return; }

   // toBeRemoved contains obj

   // If one of the children is empty, use the other

   if (toBeRemoved.left == null || toBeRemoved.right == null)
   {
      Node newChild;
      if (toBeRemoved.left == null) { newChild = toBeRemoved.right; }
      else { newChild = toBeRemoved.left; }

      fixBeforeRemove(toBeRemoved);
      replaceWith(toBeRemoved, newChild);
      return;
   }

   // Neither subtree is empty

   // Find smallest element of the right subtree

   Node smallest = toBeRemoved.right;
   while (smallest.left != null)
   {
      smallest = smallest.left;
   }

   // smallest contains smallest child in right subtree

   // Move contents, unlink child

   toBeRemoved.data = smallest.data;
   fixBeforeRemove(smallest);
   replaceWith(smallest, smallest.right);
}
```

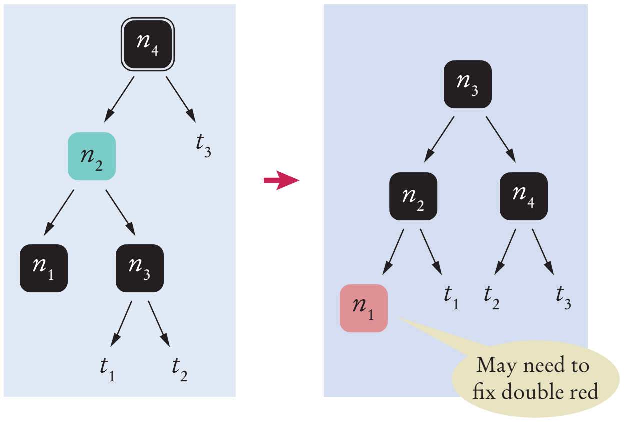
The `replaceWith` helper method, which was shown earlier, takes care of updating the parent, child, and root links. The `fixBeforeRemove` method has three cases. Removing a red leaf is safe. If a black node has a single child, that child must be red, and we can safely swap the colors. (We don't actually bother to color the node that is to be removed.) The case with a black leaf is the hardest. We need to initiate the "bubbling up" process:

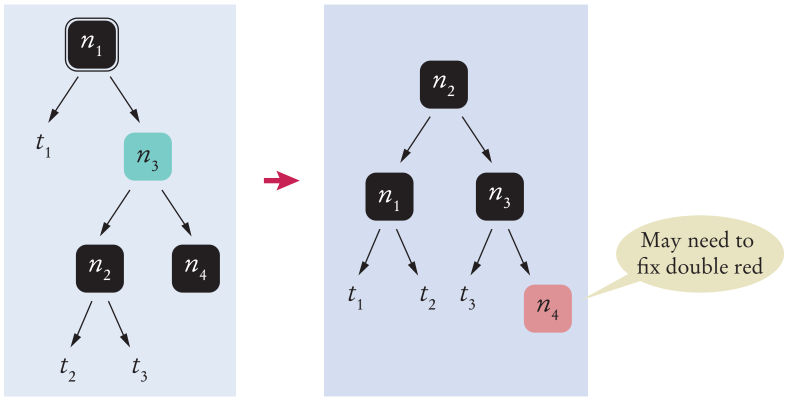
```
/**
   Fixes the tree so that it is a red-black tree after a node has been removed.
   @param toBeRemoved the node that is to be removed
*/
private void fixBeforeRemove(Node toBeRemoved)
{
   if (toBeRemoved.color == RED) { return; }

   if (toBeRemoved.left != null || toBeRemoved.right != null) // It is not a leaf
   {
      // Color the child black
      if (toBeRemoved.left == null) { toBeRemoved.right.color = BLACK; }
      else { toBeRemoved.left.color = BLACK; }
   }
   else { bubbleUp(toBeRemoved.parent); }
}
```

To bubble up, we move a "toll charge" from the children to the parent. This may result in a negative-red or double-red child, which we fix. If neither fix was successful, and the parent node is still double-black, we bubble up again until we reach the root. The root color can be safely changed to black.

```
/**
   Move a charge from two children of a parent.
   @param parent a node with two children, or null (in which case nothing is done)
*/
private void bubbleUp(Node parent)
{
   if (parent == null) { return; }
   parent.color++;
   parent.left.color--;
   parent.right.color--;

   if (bubbleUpFix(parent.left)) { return; }
   if (bubbleUpFix(parent.right)) { return; }

   if (parent.color == DOUBLE_BLACK)
   {
      if (parent.parent == null) { parent.color = BLACK; }
      else { bubbleUp(parent.parent); }
   }
}

/**
   Fixes a negative-red or double-red violation introduced by bubbling up.
   @param child the child to check for negative-red or double-red violations
   @return true if the tree was fixed
*/
private boolean bubbleUpFix(Node child)
{
   if (child.color == NEGATIVE_RED) { fixNegativeRed(child); return true; }
   else if (child.color == RED)
   {
      if (child.left != null && child.left.color == RED)
      {
         fixDoubleRed(child.left); return true;
      }
      if (child.right != null && child.right.color == RED)
      {
         fixDoubleRed(child.right); return true;
      }
   }
   return false;
}
```

We are left with the negative red removal. In the diagram in the book, we show only one of the two possible situations. In the code, we also need to handle the mirror image.





The implementation is not difficult, just long.

```
/**
   Fixes a “negative red” violation.
   @param negRed the negative red node
*/
private void fixNegativeRed(Node negRed)
{
   Node parent = negRed.parent;
   Node child;
   if (parent.left == negRed)
   {
      Node n1 = negRed.left;
      Node n2 = negRed;
```

```
         Node n3 = negRed.right;
         Node n4 = parent;
         Node t1 = n3.left;
         Node t2 = n3.right;
         Node t3 = n4.right;
         n1.color = RED;
         n2.color = BLACK;
         n4.color = BLACK;

         replaceWith(n4, n3);
         n3.setLeftChild(n2);
         n3.setRightChild(n4);
         n2.setLeftChild(n1);
         n2.setRightChild(t1);
         n4.setLeftChild(t2);
         n4.setRightChild(t3);

         child = n1;
      }
      else // Mirror image
      {
         Node n4 = negRed.right;
         Node n3 = negRed;
         Node n2 = negRed.left;
         Node n1 = parent;
         Node t3 = n2.right;
         Node t2 = n2.left;
         Node t1 = n1.left;
         n4.color = RED;
         n3.color = BLACK;
         n1.color = BLACK;

         replaceWith(n1, n2);
         n2.setRightChild(n3);
         n2.setLeftChild(n1);
         n3.setRightChild(n4);
         n3.setLeftChild(t3);
         n1.setRightChild(t2);
         n1.setLeftChild(t1);

         child = n4;
      }

      if (child.left != null && child.left.color == RED)
      {
         fixDoubleRed(child.left);
      }
      else if (child.right != null && child.right.color == RED)
      {
         fixDoubleRed(child.right);
      }
   }
```

Simple Tests

With such a complex implementation, it is extremely likely that some errors slipped in somewhere, and it is important to carry out thorough testing.

We can start with the test case used for the binary search tree from the book:

```
public static void testFromBook()
{
   RedBlackTree t = new RedBlackTree();
   t.add("D");
   t.add("B");
   t.add("A");
   t.add("C");
   t.add("F");
   t.add("E");
   t.add("I");
   t.add("G");
   t.add("H");
   t.add("J");
   t.remove("A"); // Removing leaf
   t.remove("B"); // Removing element with one child
   t.remove("F"); // Removing element with two children
   t.remove("D"); // Removing root
   assertEquals("C E G H I J ", t.toString());
}
```

The `toString` method is just like the `print` method of the binary search tree, but it returns the string instead of printing it.

If this test fails (which it did for the author at the first attempt), it is fairly easy to debug. If it passes, it gives some confidence. But there are so many different configurations that more thorough tests are required.

For a more exhaustive test, we can insert all permutations of the ten letters A – J and check that the resulting tree has the desired contents. Here, we use the permutation generator from Section 13.4.

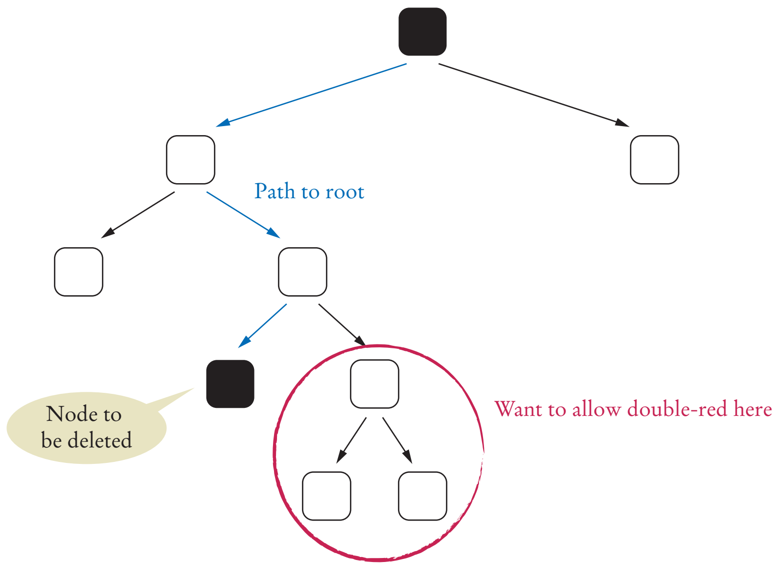
```
/**
   Inserts all permutations of a string into a red-black tree and checks that
   it contains the strings afterwards.
   @param letters a string of letters without repetition
*/
public static void insertionTest(String letters)
{
   PermutationGenerator gen = new PermutationGenerator(letters);
   for (String perm : gen.getPermutations())
   {
      RedBlackTree t = new RedBlackTree();
      for (int i = 0; i < perm.length(); i++)
      {
         String s = perm.substring(i, i + 1);
         t.add(s);
      }
      assertEquals(letters, t.toString().remove(" ", ""));
   }
}
```

This test runs through 10! = 3,628,800 permutations, which seems pretty exhaustive. But how do we really know that all possible configurations of red and black nodes have been covered? For example, it seems plausible that all four possible configurations of Figure 21 occur somewhere in these test cases, but how do we know for sure? We take up that question in the next section.

An Advanced Test

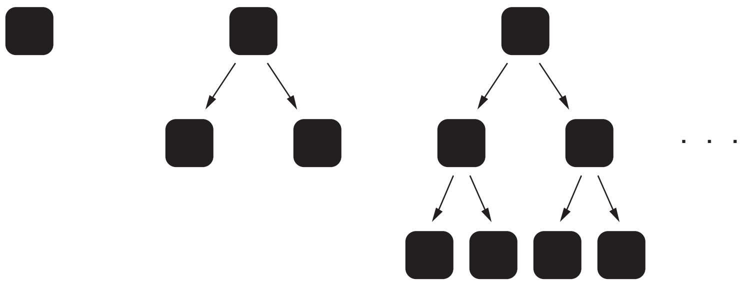
In the previous section, we reached the limits of what one can achieve with "black box" testing. For more exhaustive coverage, we need to manufacture red-black trees with all possible patterns of red and black nodes. At first, that seems hopeless. According to Exercise R17.19, there are 435,974,400 red-black trees with black height 2, far too many to generate and test.

Fortunately, we don't have to test them all. The algorithms for insertion and removal fix up nodes that form a direct path to the root. It is enough to fill this path, and its neighboring elements with all possible color combinations. Let us test the most complex case: removing a black leaf. We allow for two nodes between the leaf and the root.



Along the path to the root, we add siblings that can be red or black. We also add a couple of nodes to allow double-red violations. Each of the seven white nodes will be filled in with red or black, yielding 128 test cases. We also test all mirror images, for a total of 256 test cases.

Of course, if we fill in arbitrary combinations of red and black, the result may not be a red-black tree. First off, we add completely black subtrees



to each leaf so that the black height is constant (and equal to the black height of the node to be deleted). Then we remove trees with double-red violations. For the remaining trees, we fill in data values 1, 2, 3, so that we have a binary search tree. Then we remove the target node and check that the tree is still a proper red-black tree and that it contains the required values.

This seems like an ambitious undertaking, but it is better than the alternative—laboriously constructing a set of test cases by hand. It also provides good practice for working with trees.

In order to facilitate this style of testing, the `root` instance variable and the `Node` class of the `RedBlackTree` class are package-visible.

The following method produces the template for testing:

```
/**
   Makes a template for testing removal.
   @return a partially complete red black tree for the test.
   The node to be removed is black.
*/
private static RedBlackTree removalTestTemplate()
{
   RedBlackTree template = new RedBlackTree();

   /*
                      n7
                     /  \
                    n1   n8
                   /  \
                 n0    n3
                      /  \
                    n2*  n5
                         /\
                       n4  n6
   */

   RedBlackTree.Node[] n = new RedBlackTree.Node[9];
   for (int i = 0; i < n.length; i++) { n[i] = new RedBlackTree.Node(); }
   template.root = n[7];
   n[7].setLeftChild(n[1]);
   n[7].setRightChild(n[8]);
   n[1].setLeftChild(n[0]);
   n[1].setRightChild(n[3]);
   n[3].setLeftChild(n[2]);
   n[3].setRightChild(n[5]);
   n[5].setLeftChild(n[4]);
   n[5].setRightChild(n[6]);

   n[2].color = RedBlackTree.BLACK;

   return template;
}
```

Because each test changes the shape of the tree, we want to make a copy of the template in each test. The following recursive method makes a copy of a tree:

```
/**
   Copies all nodes of a red-black tree.
   @param n the root of a red-black tree
   @return the root node of a copy of the tree
*/
private static RedBlackTree.Node copy(RedBlackTree.Node n)
{
   if (n == null) { return null; }
   RedBlackTree.Node newNode = new RedBlackTree.Node();
   newNode.setLeftChild(copy(n.left));
   newNode.setRightChild(copy(n.right));
   newNode.data = n.data;
   newNode.color = n.color;
   return newNode;
}
```

To make a mirror image instead of a copy, just swap the left and right child:

```
/**
   Generates the mirror image of a red black tree.
   @param n the root of the tree to reflect
   @return the root of the mirror image of the tree
*/
private static RedBlackTree.Node mirror(RedBlackTree.Node n)
{
   if (n == null) { return null; }
   RedBlackTree.Node newNode = new RedBlackTree.Node();
   newNode.setLeftChild(mirror(n.right));
   newNode.setRightChild(mirror(n.left));
   newNode.data = n.data;
   newNode.color = n.color;
   return newNode;
}
```

We want to test all possible combinations of red and black nodes in the template. Each pattern of reds and blacks can be represented as a sequence of zeroes and ones, or a binary number between 0 and $2^n - 1$, where n is the number of nodes to be colored.

```
for (int k = 0; k < Math.pow(2, nodesToColor); k++)
{
   RedBlackTree.Node[] nodes = . . . // The nodes to be colored;

   // Color with the bit pattern of k
   int bits = k;
   for (RedBlackTree.Node n : nodes)
   {
      n.color = bits % 2;
      bits = bits / 2;
   }

   // Now run a test with this tree
   . . .
}
```

We need to have a helper method to get all nodes of a tree into an array. Here it is:

```
/**
   Gets all nodes of a tree in sorted order.
   @param t a red-black tree
   @return an array of all nodes in t
*/
private static RedBlackTree.Node[] getNodes(RedBlackTree t)
{
   RedBlackTree.Node[] nodes = new RedBlackTree.Node[count(t.root)];
   getNodes(t.root, nodes, 0);
   return nodes;
}

/**
   Gets all nodes of a subtree and fills them into an array.
   @param n the root of the subtree
   @param nodes the array into which to place the nodes
   @param start the offset at which to start placing the nodes
   @return the number of nodes placed
*/
private static int getNodes(RedBlackTree.Node n, RedBlackTree.Node[] nodes, int start)
{
      if (n == null) { return 0; }
      int leftFilled = getNodes(n.left, nodes, start);
      nodes[start + leftFilled] = n;
      int rightFilled = getNodes(n.right, nodes, start + leftFilled + 1);
      return leftFilled + 1 + rightFilled;
   }
```

Once the tree has been colored, we need to give it a constant black height. For each leaf, we compute the cost to the root:

```
   /**
      Computes the cost from a node to a root.
      @param n a node of a red-black tree
      @return the number of black nodes between n and the root
   */
   private static int costToRoot(RedBlackTree.Node n)
   {
      int c = 0;
      while (n != null) { c = c + n.color; n = n.parent; }
      return c;
   }
```

If that cost is less than the black height of the node to be removed, we add a full tree of black nodes to make up the difference. This method makes these trees:

```
   /**
      Makes a full tree of black nodes of a given depth.
      @param depth the desired depth
      @return the root node of a full black tree
   */
   private static RedBlackTree.Node fullTree(int depth)
   {
      if (depth <= 0) { return null; }
      RedBlackTree.Node r = new RedBlackTree.Node();
      r.color = RedBlackTree.BLACK;
      r.setLeftChild(fullTree(depth - 1));
      r.setRightChild(fullTree(depth - 1));
      return r;
   }
```

This loop adds the full trees to the nodes:

```
   int targetCost = costToRoot(toDelete);
   for (RedBlackTree.Node n : nodes)
   {
      int cost = targetCost - costToRoot(n);
      if (n.left == null) { n.setLeftChild(fullTree(cost)); }
      if (n.right == null) { n.setRightChild(fullTree(cost)); }
   }
```

Now we need to fill the tree with values. Because `getNodes` returns the nodes in sorted order, we just populate them with 0, 1, 2, and so on.

```
   /**
      Populates this tree with the values 0, 1, 2, . . . .
      @param t a red-black tree
      @return the number of nodes in t
   */
   private static int populate(RedBlackTree t)
   {
      RedBlackTree.Node[] nodes = getNodes(t);
      for (int i = 0; i < nodes.length; i++)
      {
         nodes[i].data = new Integer(i);
      }
      return nodes.length;
   }
```

The resulting tree has constant black height, but it might still not be a valid red-black tree because it might have double-red violations. We could test just that, but we need to have a general method that tests the red-black properties after the removal. We also want to verify that all the parent and child links are not corrupted. Because removal introduces colors other than red or black (e.g., double-black or negative-red), we want to check that those colors are no longer present after the operation has completed. Specifically, we need to check the following for each subtree with root `n`:

- The left and right subtree of `n` have the same black depth.
- `n` must be red or black.
- If `n` is red, its parent is not.
- If `n` has children, then their parent references must equal `n`.
- `n.parent` is `null` if and only if `n` is the root of the tree.
- The root is black.

Moreover, because fixing double-red and negative-red violations reorders nodes, we will check that the tree is still a binary search tree. This can be tested by visiting the tree in order.

Here are the integrity check methods:

```
/**
   Checks whether a red-black tree is valid and throws an exception if not.
   @param t the tree to test
*/
public static void checkRedBlack(RedBlackTree t)
{
   checkRedBlack(t.root, true);

   // Check that it's a BST
   RedBlackTree.Node[] nodes = getNodes(t);
   for (int i = 0; i < nodes.length - 1; i++)
   {
      if (nodes[i].data.compareTo(nodes[i + 1].data) > 0)
      {
         throw new IllegalStateException(
            nodes[i].data + " is larger than " + nodes[i + 1].data);
      }
   }
}

/**
   Checks that the tree with the given node is a red-black tree, and throws an
   exception if a structural error is found.
   @param n the root of the subtree to check
   @param isRoot true if this is the root of the tree
   @return the black depth of this subtree
*/
private static int checkRedBlack(RedBlackTree.Node n, boolean isRoot)
{
   if (n == null) { return 0; }
   int nleft = checkRedBlack(n.left, false);
   int nright = checkRedBlack(n.right, false);
   if (nleft != nright)
   {
      throw new IllegalStateException(
         "Left and right children of " + n.data
         + " have different black depths");
   }
   if (n.parent == null)
   {
      if (!isRoot)
      {
         throw new IllegalStateException(
            n.data + " is not root and has no parent");
      }
      if (n.color != RedBlackTree.BLACK)
      {
         throw new IllegalStateException("Root "
            + n.data + " is not black");
      }
   }
   else
   {
      if (isRoot)
      {
         throw new IllegalStateException(
            n.data + " is root and has a parent");
      }
      if (n.color == RedBlackTree.RED
         && n.parent.color == RedBlackTree.RED)
      {
         throw new IllegalStateException(
            "Parent of red " + n.data + " is red");
      }
   }
   if (n.left != null && n.left.parent != n)
   {
      throw new IllegalStateException(
         "Left child of " + n.data + " has bad parent link");
   }
   if (n.right != null && n.right.parent != n)
   {
      throw new IllegalStateException(
         "Right child of " + n.data + " has bad parent link");
   }
   if (n.color != RedBlackTree.RED && n.color != RedBlackTree.BLACK)
   {
      throw new IllegalStateException(
         n.data + " has color " + n.color);
   }
   return n.color + nleft;
}

public static void assertEquals(Object expected, Object actual)
{
   if (expected == null && actual != null ||
      !expected.equals(actual))
   {
      throw new AssertionError("Expected " + expected + " but found " + actual);
   }
}
```

Now we have all the pieces together. Here is the complete method for testing removal. Note that the outer loop switches between copying and mirroring, and the inner loop iterates over all red/black colorings.

```
/**
   Tests removal, given a template for a tree with a black node that
   is to be deleted. All other nodes should be given all possible combinations
   of red and black.
   @param t the template for the test cases
*/
public static void removalTest(RedBlackTree t)
{
   for (int m = 0; m <= 1; m++)
   {
      int nodesToColor = count(t.root) - 2; // We don't recolor the root or toDelete
      for (int k = 0; k < Math.pow(2, nodesToColor); k++)
      {
         RedBlackTree rb = new RedBlackTree();
         if (m == 0) { rb.root = copy(t.root); }
         else { rb.root = mirror(t.root); }

         RedBlackTree.Node[] nodes = getNodes(rb);
         RedBlackTree.Node toDelete = null;

         // Color with the bit pattern of k
         int bits = k;
         for (RedBlackTree.Node n : nodes)
         {
            if (n == rb.root)
            {
               n.color = RedBlackTree.BLACK;
            }
            else if (n.color == RedBlackTree.BLACK)
            {
               toDelete = n;
            }
            else
            {
               n.color = bits % 2;
               bits = bits / 2;
            }
         }

         // Add children to make equal costs to null
         int targetCost = costToRoot(toDelete);
         for (RedBlackTree.Node n : nodes)
         {
            int cost = targetCost - costToRoot(n);
            if (n.left == null) { n.setLeftChild(fullTree(cost)); }
            if (n.right == null) { n.setRightChild(fullTree(cost)); }
         }

         int filledSize = populate(rb);
         boolean good = true;
         try { checkRedBlack(rb); }
         catch (IllegalStateException ex) { good = false; }
         if (good)
         {
               Comparable d = toDelete.data;
               rb.remove(d);
               checkRedBlack(rb);
               for (Integer j = 0; j < filledSize; j++)
               {
                  if (!rb.find(j) && !d.equals(j))
                  {
                     throw new IllegalStateException(j + " deleted");
                  }
                  if (rb.find(d))
                  {
                     throw new IllegalStateException(d + " not deleted");
                  }
               }
            }
         }
      }
   }
```

In our `main` method, we run all three tests. The last line, which only happens if no exceptions have been thrown, proclaims that the tests passed.

```
   public static void main(String[] args)
   {
      testFromBook();
      insertionTest("ABCDEFGHIJ");
      removalTest(removalTestTemplate());
      System.out.println("All tests passed.");
   }
```

See `ch17/worked_example_2` in your code folder for the complete program.