Red Black Trees

The Red Black Tree is one of the most popular implementation of sets and dictionaries.

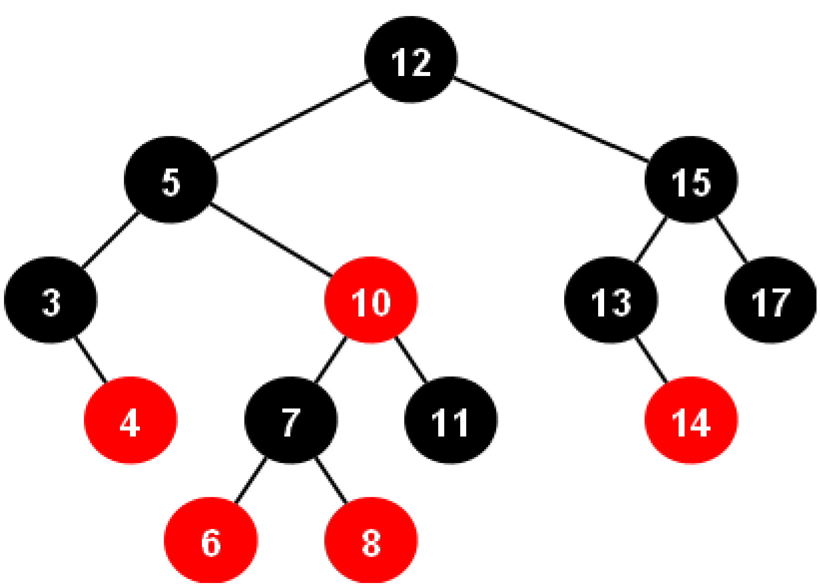
CONTENTS

Definition

A **red-black tree** is a binary search tree in which each node is colored red or black such that

- The root is black
- The children of a red node are black
- Every path from the root to a 0-node or a 1-node has the same number of black nodes.

Example:



Red black trees do not necessarily have minimum height, but they never get really bad. The height is never greater than 2 $\log_2(n)$, where n is the number of nodes.

Overview

- Red Black Trees are Cool
  - An online, interactive, animated applet that will teach you *how* they work, is here.
- Red Black Trees are Useful
  - Red Black trees are used in many real-world libraries as the foundations for sets and dictionaries.
  - They are used to implement the `TreeSet` and `TreeMap` classes in the Java Core API, as well as the Standard C++ sets and maps.

How They Work

Lookup

A red black tree is a BST. Lookup in an RBT is just lookup in a BST. The colors don't matter.

Insertion

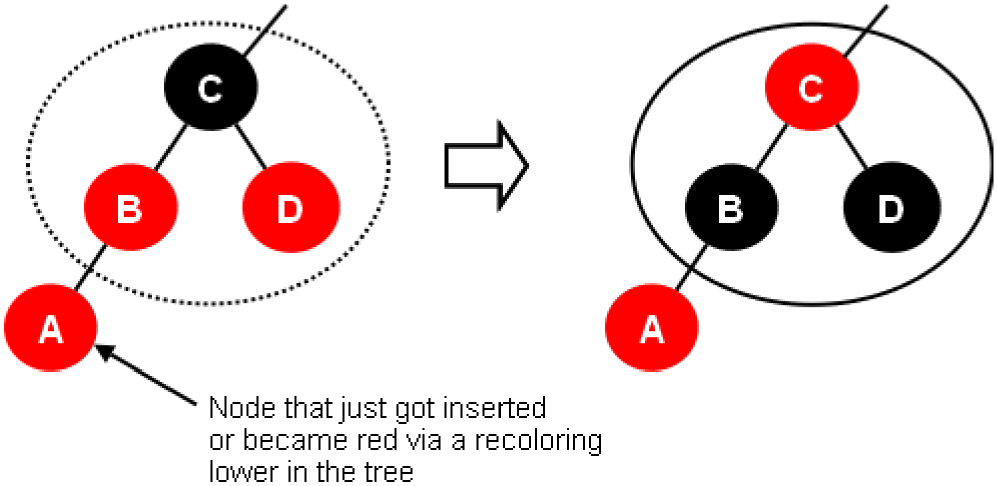
The algorithm has three steps:

1. Insert as you would into a BST, coloring the node red.
2. If the parent of the node you just inserted was red, you have a double-red problem which you must correct.
3. Color the root node black.

A double red problem is corrected with zero or more **recolorings** followed by zero or one **restructuring**.

Recoloring

Recolor whenever *the sibling of a red node's red parent is red*:

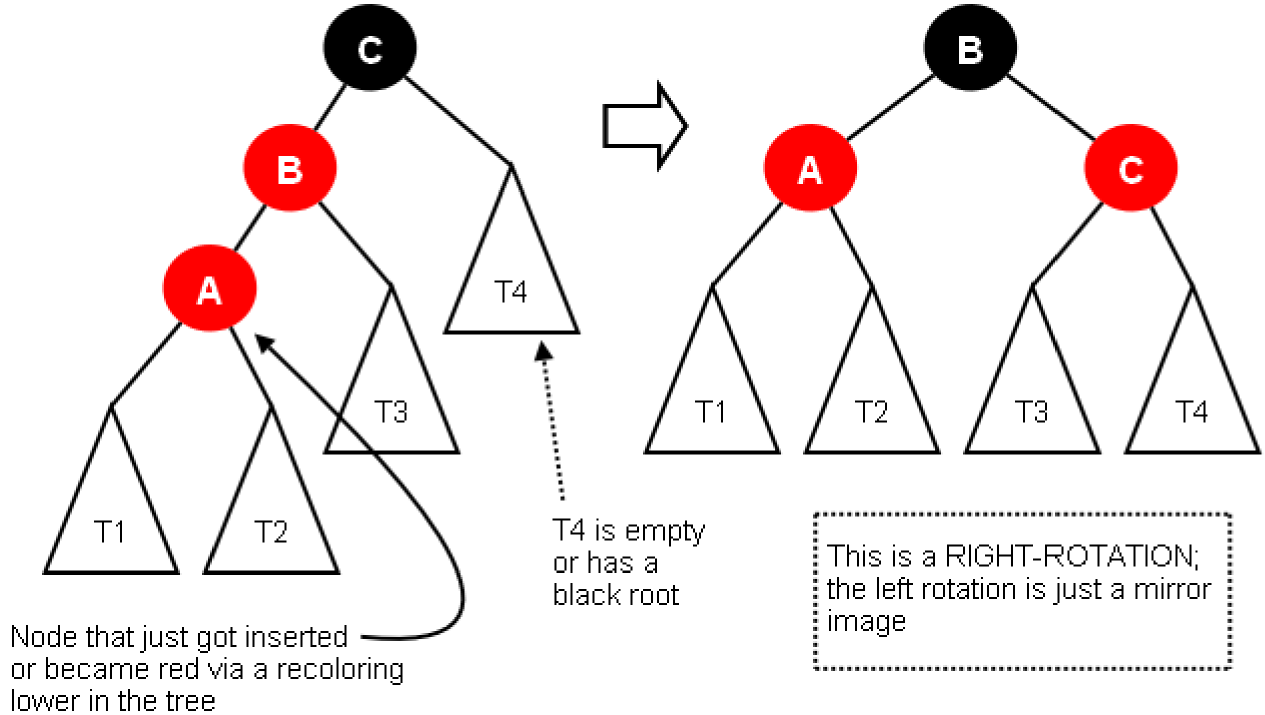


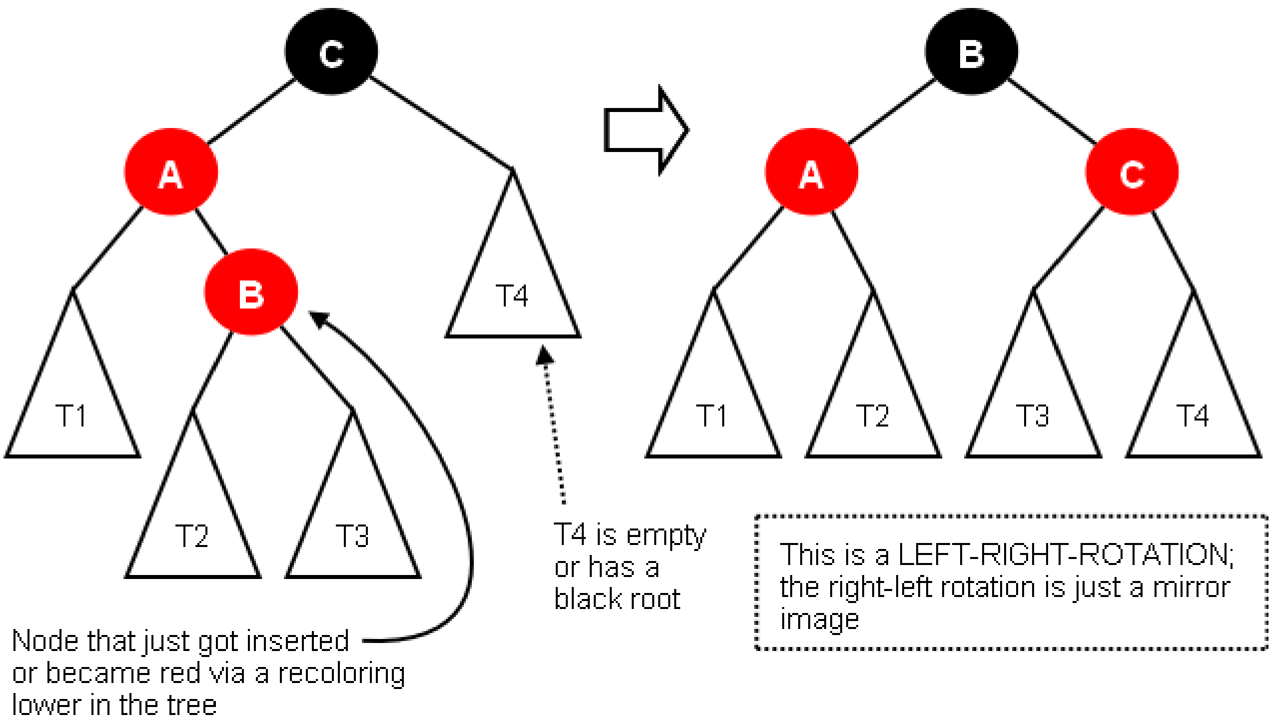
Restructuring

Restructure whenever *the red child's red parent's sibling is black or null*. There are four cases:

- Right
- Left
- Right-Left
- Left-Right

When you restructure, the root of the restructured subtree is colored black and its children are colored red.





An Example

Let's insert into an initially empty red-black tree, the following: 4 7 12 15 3 5 14 18 16 17. The tree takes shape like this:

1. (4B)
2. (4B (7R))

3. (4B (7R (12R))) - whoops, need to rotate left
   ((4R) 7B (12R))
4. ((4R) 7B (12R (15R))) - whoops, need to recolor
   ((4B) 7B (12B (15R)))
5. (((3R) 4B) 7B (12B (15R)))
6. (((3R) 4B (5R)) 7B (12B (15R)))
7. (((3R) 4B (5R)) 7B (12B ((14R) 15R))) -- whoops, need RL rotation
   (((3R) 4B (5R)) 7B ((12R) 14B (15R)))
8. (((3R) 4B (5R)) 7B ((12R) 14B (15R (18R)))) -- whoops, recolor
   (((3R) 4B (5R)) 7B ((12B) 14R (15B (18R))))
9. (((3R) 4B (5R)) 7B ((12B) 14R (15B ((16R) 18R)))) -- whoops, need RL rotation
   (((3R) 4B (5R)) 7B ((12B) 14R ((15R) 16B (18R))))
10. (((3R) 4B (5R)) 7B ((12B) 14R ((15R) 16B ((17R) 18R)))) -- whoops, need to recolor
    (((3R) 4B (5R)) 7B ((12B) 14R ((15B) 16R ((17R) 18B))))
    -- whoa, the recolor caused a problem higher up in the tree!
    -- we need to do a left rotation
    ((((3R) 4B (5R)) 7R (12B)) 14B ((15B) 16R ((17R) 18B)))

Deletion

The algorithm is in the code below. Have fun figuring it out.

An Implementation

Since Red Black Trees are just a kind of binary search tree, it makes sense to subclass:

RedBlackTree.java

```
import java.awt.Color;
import java.util.Comparator;

/**
 * A simple red-black tree class.
 */
public class RedBlackTree extends BinarySearchTree {

    /**
     * Constructs an empty RedBlackTree that can only accept Comparables as items.
     */
    public RedBlackTree() {
        this(null);
```

```
    }

    /**
     * Constructs an empty RedBlackTree that orders its items according to the given
     * comparator.
     */
    public RedBlackTree(Comparator c) {
        super(c);
    }

    /**
     * The nodes in a red-black tree store a color together with the actual data in
     * the node.

     */
    class Node extends LinkedBinaryTreeNode {
        Color color = Color.black;

        public Node(Object data) {
            super(data);
        }
    }

    /**
     * Adds a single data item to the tree. If there is already an item in the tree
     * that compares equal to the item being inserted, it is "overwritten" by the
     * new item. Overrides BinarySearchTree.add because the tree needs to be
     * adjusted after insertion.
     */
    public void add(Object data) {
        if (root == null) {
            root = new Node(data);
        }
        BinaryTreeNode n = root;
        while (true) {
            int comparisonResult = compare(data, n.getData());
            if (comparisonResult == 0) {
                n.setData(data);
                return;
            } else if (comparisonResult < 0) {
                if (n.getLeft() == null) {
                    n.setLeft(new Node(data));
                    adjustAfterInsertion((Node) n.getLeft());
                    break;
                }
                n = n.getLeft();
            } else { // comparisonResult > 0
                if (n.getRight() == null) {
                    n.setRight(new Node(data));
                    adjustAfterInsertion((Node) n.getRight());
                    break;
                }
                n = n.getRight();
            }
        }
    }

    /**
     * Removes the node containing the given value. Does nothing if there is no such
     * node.
     */
    public void remove(Object data) {
        Node node = (Node) nodeContaining(data);
        if (node == null) {
            // No such object, do nothing.
            return;
```

```
            return;
        } else if (node.getLeft() != null && node.getRight() != null) {
            // Node has two children, Copy predecessor data in.
            BinaryTreeNode predecessor = predecessor(node);
            node.setData(predecessor.getData());
            node = (Node) predecessor;
        }
        // At this point node has zero or one child
        Node pullUp = leftOf(node) == null ? rightOf(node) : leftOf(node);
        if (pullUp != null) {
            // Splice out node, and adjust if pullUp is a double black.
            if (node == root) {
                setRoot(pullUp);
            } else if (node.getParent().getLeft() == node) {
                node.getParent().setLeft(pullUp);
            } else {
                node.getParent().setRight(pullUp);
            }
            if (isBlack(node)) {
                adjustAfterRemoval(pullUp);
            }
        } else if (node == root) {
            // Nothing to pull up when deleting a root means we emptied the tree
            setRoot(null);
        } else {
            // The node being deleted acts as a double black sentinel
            if (isBlack(node)) {
                adjustAfterRemoval(node);
            }
            node.removeFromParent();
        }
    }

    /**
     * Classic algorithm for fixing up a tree after inserting a node.
     */
    private void adjustAfterInsertion(Node n) {
        // Step 1: color the node red
        setColor(n, Color.red);

        // Step 2: Correct double red problems, if they exist
        if (n != null && n != root && isRed(parentOf(n))) {

            // Step 2a (simplest): Recolor, and move up to see if more work
            // needed
            if (isRed(siblingOf(parentOf(n)))) {
                setColor(parentOf(n), Color.black);
                setColor(siblingOf(parentOf(n)), Color.black);
                setColor(grandparentOf(n), Color.red);
                adjustAfterInsertion(grandparentOf(n));
            }

            // Step 2b: Restructure for a parent who is the left child of the
            // grandparent. This will require a single right rotation if n is
            // also
            // a left child, or a left-right rotation otherwise.
            else if (parentOf(n) == leftOf(grandparentOf(n))) {
                if (n == rightOf(parentOf(n))) {
                    rotateLeft(n = parentOf(n));
                }
                setColor(parentOf(n), Color.black);
                setColor(grandparentOf(n), Color.red);
                rotateRight(grandparentOf(n));
            }

            // Step 2c: Restructure for a parent who is the right child of the
```

```
        // Step 2c: Restructure for a parent who is the right child of the
        // grandparent. This will require a single left rotation if n is
        // also
        // a right child, or a right-left rotation otherwise.
        else if (parentOf(n) == rightOf(grandparentOf(n))) {
            if (n == leftOf(parentOf(n))) {
                rotateRight(n = parentOf(n));
            }
            setColor(parentOf(n), Color.black);
            setColor(grandparentOf(n), Color.red);
            rotateLeft(grandparentOf(n));
        }
    }

    // Step 3: Color the root black
    setColor((Node) root, Color.black);
}

/**
 * Classic algorithm for fixing up a tree after removing a node; the parameter
 * to this method is the node that was pulled up to where the removed node was.
 */
private void adjustAfterRemoval(Node n) {
    while (n != root && isBlack(n)) {
        if (n == leftOf(parentOf(n))) {
            // Pulled up node is a left child
            Node sibling = rightOf(parentOf(n));
            if (isRed(sibling)) {
                setColor(sibling, Color.black);
                setColor(parentOf(n), Color.red);
                rotateLeft(parentOf(n));
                sibling = rightOf(parentOf(n));
            }
            if (isBlack(leftOf(sibling)) && isBlack(rightOf(sibling))) {
                setColor(sibling, Color.red);
                n = parentOf(n);
            } else {
                if (isBlack(rightOf(sibling))) {
                    setColor(leftOf(sibling), Color.black);
                    setColor(sibling, Color.red);
                    rotateRight(sibling);
                    sibling = rightOf(parentOf(n));
                }
                setColor(sibling, colorOf(parentOf(n)));
                setColor(parentOf(n), Color.black);
                setColor(rightOf(sibling), Color.black);
                rotateLeft(parentOf(n));
                n = (Node) root;
            }
        } else {
            // pulled up node is a right child
            Node sibling = leftOf(parentOf(n));
            if (isRed(sibling)) {
                setColor(sibling, Color.black);
                setColor(parentOf(n), Color.red);
                rotateRight(parentOf(n));
                sibling = leftOf(parentOf(n));
            }
            if (isBlack(leftOf(sibling)) && isBlack(rightOf(sibling))) {
                setColor(sibling, Color.red);
                n = parentOf(n);
            } else {
                if (isBlack(leftOf(sibling))) {
                    setColor(rightOf(sibling), Color.black);
                    setColor(sibling, Color.red);
                    rotateLeft(sibling);
```

```
                        rotateLeft(sibling);
                        sibling = leftOf(parentOf(n));
                    }
                    setColor(sibling, colorOf(parentOf(n)));
                    setColor(parentOf(n), Color.black);
                    setColor(leftOf(sibling), Color.black);
                    rotateRight(parentOf(n));
                    n = (Node) root;
                }
            }
        }
        setColor(n, Color.black);
    }

    // The following helpers dramatically simplify the code by getting
    // all the null pointer checking out of the adjustment methods.

    private Color colorOf(Node n) {
        return n == null ? Color.black : n.color;
    }

    private boolean isRed(Node n) {
        return n != null && colorOf(n) == Color.red;
    }

    private boolean isBlack(Node n) {
        return n == null || colorOf(n) == Color.black;
    }

    private void setColor(Node n, Color c) {
        if (n != null)
            n.color = c;
    }

    private Node parentOf(Node n) {
        return n == null ? null : (Node) n.getParent();
    }

    private Node grandparentOf(Node n) {
        return (n == null || n.getParent() == null) ? null : (Node) n.getParent().getParent();
    }

    private Node siblingOf(Node n) {
        return (n == null || n.getParent() == null) ? null
            : (n == n.getParent().getLeft()) ? (Node) n.getParent().getRight() : (Node) n.getParent().getLe
    }

    private Node leftOf(Node n) {
        return n == null ? null : (Node) n.getLeft();
    }

    private Node rightOf(Node n) {
        return n == null ? null : (Node) n.getRight();
    }
}
```

A Viewer

For fun, I made my own viewer application. No animation, though:

RedBlackTreeViewer.java

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
import javax.swing.border.BevelBorder;

/**
 * A little application that lets you interactively manipulate a binary search
 * tree.
 */
public class RedBlackTreeViewer extends JFrame {
    RedBlackTree tree = new RedBlackTree();

    JFrame frame = new JFrame("Red Black Tree Viewer");
    JTextField valueField = new JTextField(40);
    JPanel buttonPanel = new JPanel();
    BinaryTreePanel panel = new BinaryTreePanel(null, 40, 40);
    JScrollPane displayArea = new JScrollPane();
    JLabel messageLine = new JLabel();

    /**
     * An operation encapsulates a button and its action. The constructor will
     * create a button, add it to a button panel, and register itself as a listener
     * for the button. The listener first reads inputs from a textfield, then calls
     * a subclass-supplied method with those inputs, then displays the resulting
     * tree in the display area.
     */
    private abstract class Operation implements ActionListener {
        public Operation(String label) {
            JButton button = new JButton(label);
            buttonPanel.add(button);
            button.addActionListener(this);
        }

        public void actionPerformed(ActionEvent event) {
            String value = valueField.getText();
            messageLine.setText("");
            try {
                execute(value);
            } catch (Exception e) {
                e.printStackTrace();
            }
            // Update the picture and return the focus to the text field. Select
            // all the text in the textfield so it can easily be overwritten.
            panel.setTree(tree.getRoot());
            valueField.requestFocus();
            valueField.selectAll();

        }

        protected abstract void execute(String value);
    }
```

```
    /**
     * Constructs a viewer, laying out all the components in a very nice way, and
     * constructs and registers all the operation objects.
     */
    public RedBlackTreeViewer() {
        JPanel valuePanel = new JPanel();
        valuePanel.add(new JLabel("Value: "));
        valuePanel.add(valueField);

        JPanel controlPanel = new JPanel();
        controlPanel.setLayout(new GridLayout(0, 1));
        controlPanel.add(valuePanel);
        controlPanel.add(buttonPanel);

        // NOTE: Hardcoded preferred size! Fix this in the exercises.
        panel.setPreferredSize(new Dimension(2048, 2048));
        panel.setBackground(Color.white);
        panel.setBorder(new BevelBorder(BevelBorder.LOWERED));
        displayArea.setViewportView(panel);

        frame.setBackground(Color.lightGray);
        frame.getContentPane().add(controlPanel, "North");
        frame.getContentPane().add(displayArea, "Center");
        frame.getContentPane().add(messageLine, "South");
        frame.pack();

        new Operation("Add") {
            protected void execute(String value) {
                tree.add(value);
            }
        };
        new Operation("Add All") {
            protected void execute(String value) {
                for (String s : value.split("\\s+"))
                    tree.add(s);
            }
        };
        new Operation("Lookup") {
            protected void execute(String value) {
                messageLine.setText("The value \"" + value + "\" is " + (tree.contains(value) ? "" : "not ") +
            }
        };
        new Operation("Remove") {
            protected void execute(String value) {
                tree.remove(value);
            }
        };
    }

    /**
     * Makes an application whose main window is a RedBlackTreeViewer.
     */
    public static void main(String[] args) {
        RedBlackTreeViewer viewer = new RedBlackTreeViewer();
        viewer.frame.setSize(540, 480);
        viewer.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        viewer.frame.setVisible(true);
    }
}
```