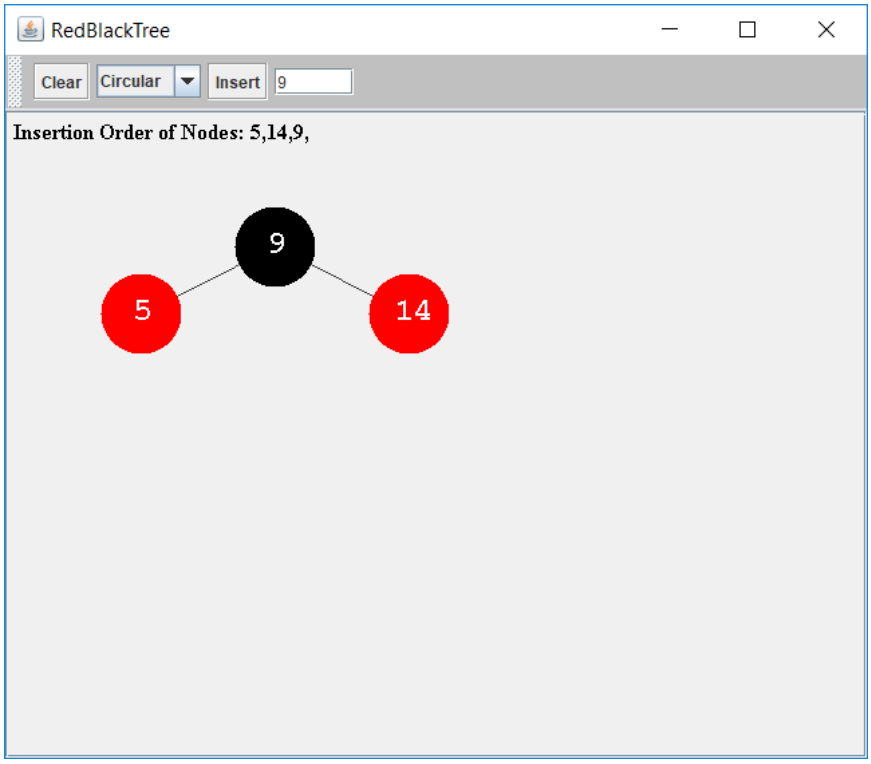
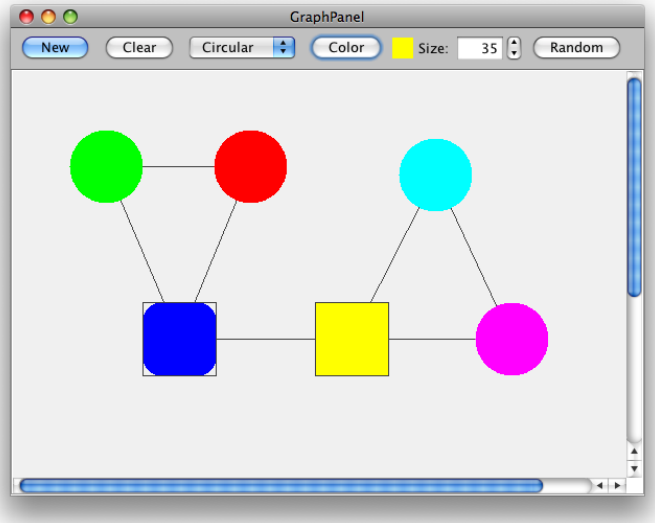
Java Swing Viewer for RedBlack Tree



The code is based on a class by John B. Matthews.

However, his class was concerned with drawing graphs (remember them? They consist of nodes connected by edges).



Objectives:

1. To integrate your RedBlackTree class without requiring major modifications to the code.
   Note: Your class must be named `RedBlackTree`.
   The only modifications that should be required are to change some of the instance field members from `private` to `protected`. Eclipse will alert you to any variables that the new class cannot access.
2. To show the insertion order, so that you can infer the rotations/recolourings that took place (to a certain extent; you should definitely be able to hand-trace through the various insertions, though, to get to the same resultant tree as displayed – assuming your code works, of course.)
3. To use a Graph data structure for display purposes.
4. To figure out the relative locations of each node/leaf for display purposes. *How can you traverse the binary-tree in such a way that you can infer where it should be placed?*

Inheritance

In order to achieve objective 1, I created a class called `BinarySearchTreeViewCapable` which inherits from `RedBlackTree`. I can add additional data (instance fields), methods, and override/overload methods in the superclass to give the extra functionality required without needing large-scale changes in the superclass.

Using a Graph

```
private List<DisplayNode> nodes = new ArrayList<DisplayNode>();

private List<Edge> edges = new ArrayList<Edge>();
```

As we know from our discussion on path-finding, a graph is a data structure with **nodes** and **edges** (each of which connect two nodes together).

In terms of displaying our data structure, this is a natural fit (note that graphs by themselves do not have any hierarchical structure).

As long as our node stores location information and the value/key, then we can use the parent reference to connect two nodes with an edge.

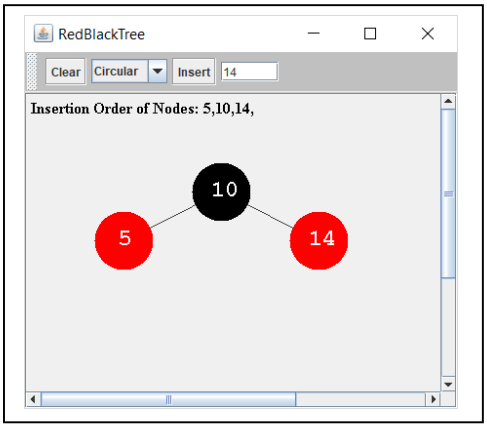
```
/**
 * An Edge is a pair of Nodes.
 */
private static class Edge {

    private DisplayNode n1;
    private DisplayNode n2;

    public Edge(DisplayNode n1, DisplayNode n2) {
        this.n1 = n1;
        this.n2 = n2;
    }

    public void draw(Graphics g) {
        Point p1 = n1.getLocation();
```

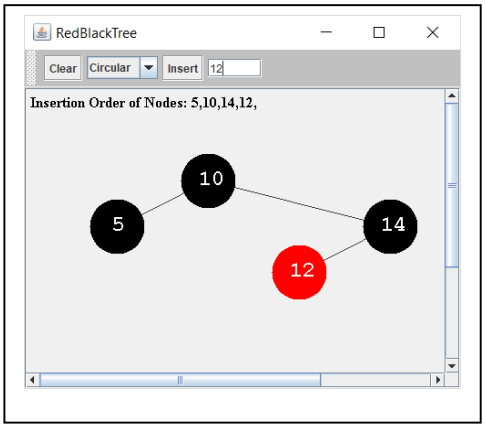
```
            Point p2 = n2.getLocation();
            g.setColor(Color.darkGray);
            g.drawLine(p1.x, p1.y, p2.x, p2.y);
        }
    }

    /**
     * A DisplayNode represents a node in a graph.
     * This was originally called Node but I changed it to
     * avoid confusion with the Node class in the Tree
     */
    private static class DisplayNode {

        private Point p; //has x,y coords to display at
        private int r;
        private Color color;
        private Kind kind; //circle/rounded/square
        private Integer nodeValue; //key/value stored in node



        /**
         * Draw this node.
         */
        public void draw(Graphics g) {
            g.setColor(this.color);
            if (this.kind == Kind.Circular) {
                g.fillOval(b.x, b.y, b.width, b.height);
            } else if (this.kind == Kind.Rounded) {
                g.fillRoundRect(b.x, b.y, b.width, b.height, r, r);
            } else if (this.kind == Kind.Square) {
                g.fillRect(b.x, b.y, b.width, b.height);
            }

            //Now "draw" the key/value on top of the node shape
```

Figuring out the positions of Nodes

This is trickier than it might first appear.

For example, consider Node(14) in each of the diagrams below.





It's at the same location (right of root) but, depending on the attached subtrees, its graphical position will vary.

It turns out that with a simple InOrderTraversal we can assign x,y position information to each of our DisplayNodes.

Here's the basic algorithm.

Every time [Insert] is clicked we:

1. Insert the node into the tree.
2. Call a method to recompute all the node positions.
3. Call `repaint()` which will force a redraw.

```
private class InsertAction extends AbstractAction {
  private JTextField insertVal;
    public InsertAction(String name, JTextField insertVal) {
        super(name);
        this.insertVal = insertVal;
    }

    public void actionPerformed(ActionEvent e) {

     Integer valToInsert = Integer.valueOf(insertVal.getText().trim());
     insertionsInOrder.add(valToInsert);

     myTree.insert(valToInsert);
     myTree.computeNodePositions();

        repaint();
    }
}
```

`myTree.computeNodePositions()` calls an InOrderTraversal. However (unlike the version in your class) this version also requires `depth` information to be passed down recursively, so that the Y display for each node can be computed. **Therefore I need an overloaded version of `RecInOrderTraversal()`.**

```
private void recInOrderTraversal(MyNode subTreeRoot, int depth)
{

   if(subTreeRoot == null) return;

   recInOrderTraversal((MyNode)subTreeRoot.left, depth + 1);
   processNode(subTreeRoot, depth);
   recInOrderTraversal((MyNode)subTreeRoot.right, depth + 1);
}
```

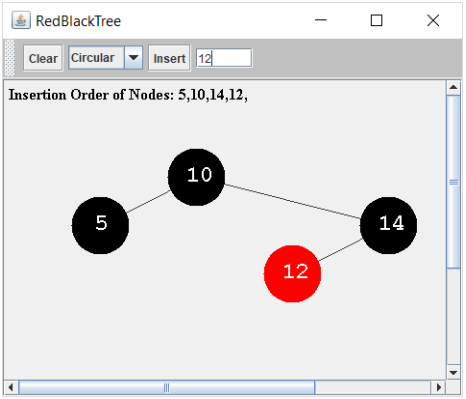
`processNode()` has the task of computing the `xpos` and `ypos` for the node. It also creates the entities required for the Graph data structure (A `DisplayNode` and an `Edge` associating the current `DisplayNode` with its parent).

Here are the first few lines:

```
/**
 * This performs the actual calculation of the relative x and y positions for
 * the node.
 * It creates a data structure necessary to display the node.
 * It also creates an edge using the current node and the parent node.
 * @param node
 * @param depth
 */
//@Override
private void processNode(MyNode node, int depth)
{
   Color nodeColour = (node.nodeColourRed) ? Color.red : Color.black;

   node.xpos = totalNodes++;
   node.ypos = depth;
   Point p = new Point(NodeConstants.X_OFFSET + node.xpos * NodeConstants.X_OFFSET,
                 NodeConstants.Y_OFFSET + node.ypos * NodeConstants.Y_OFFSET);
```

You'll see above that the `ypos` is related to the depth, as you might expect. **However, `xpos` is just calculated using a counter**. This actually make sense when you consider the nature of an *InOrderTraversal*. Look at the debug output below in conjunction with the screenshot and it should become clearer.

```
Node [value=5]  [Colour: Black] xpos: 0 ypos: 2
Node [value=10] [Colour: Black] xpos: 1 ypos: 1
Node [value=12] [Colour: Red]   xpos: 2 ypos: 3
Node [value=14] [Colour: Black] xpos: 3 ypos: 2
```



Appendix A: UML for RedBlackTree and GraphPanel Viewer