JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 2, March-April 2005

Generic Red-Black Tree and its C# Implementation

Dr. Richard Wiener, Editor-in-Chief, JOT**,** Associate Professor of Computer Science**,** University of Colorado at Colorado Springs

The focus of this installment of the Educator's Corner is on tree construction – red-black trees. Some of the material for this column is taken from Chapter 13 of the forthcoming book "Modern Software Development Using C#/.NET" by Richard Wiener. This book will be published by Thomson, Course Technology in the Fall, 2005.

A fascinating and important balanced binary search tree is the **red-black** tree. Rudolf Bayer invented this important tree structure in 1972, about 10 years after the introduction of AVL trees. He is also credited with the invention of the B-tree, a structure used extensively in database systems. Bayer referred to his red-black trees as "symmetric binary B-trees."

Red-black trees, as they are now known, like AVL trees, are "self-balancing". Whenever a new object is inserted or deleted, the resulting tree continues to satisfy the conditions required to be a red-black tree. The computational complexity for insertion or deletion can be shown to be $O(log_2 n)$, similar to an AVL tree.

Red black trees are important in practice. They are used as the basis for Java's implementation of its standard *SortedSet* collection class. There is no comparable collection in the C# standard collections so a C# implementation of red-black trees might be useful.

The rules that define a red-black tree are interesting because they are less constrained than the rather strict rules associated with an AVL tree. Each node is assigned a color of red or black. This can be accomplished using a field of type *bool* in class *Node* (true if the node is red and false if it is black).

The formal rules that define a red-black binary search tree are the following:

1. Every node is colored either red or black.
2. The root node is always black.
3. Every external node (null child of a leaf node) is black.
4. If a node is red, both of its children are black.
5. Every path from the root to an external node contains the same number of black nodes.

Cite this column as follows: Richard Wiener: "Generic Red-Black Tree and its C# Implementation", in *Journal of Object Technology*, vol. 4, no. 2, March-April 2005, pp. 59-80
http://www.jot.fm/issues/issue_2005_03/column6

It is rule 5 that leads to a balanced tree structure. Since red nodes may not have any red children, if the black height from root to every leaf node is the same, this implies that any two paths from root to leaf can differ only by a factor of two. Using a relatively simple proof of mathematical induction, Kenneth Berman and Jerome Paul show on page 659 of their book, *Algorithms: Sequential, Parallel, and Distributed*, Thomson, Course Technology, 2005, that the relationship between the maximum depth of a red-black tree and the number of internal nodes is, $depth = 2\ log_2(n + 1)$ where n is the number of internal nodes.

Let us consider some examples of red-black trees in Figure 1.

The first red-black tree has a black depth of 2 from the root to every leaf node. The second red-black tree has a black depth of 3 from the root to every leaf node. Each of these trees is generated by a search tree GUI application (to be the subject of a future column).

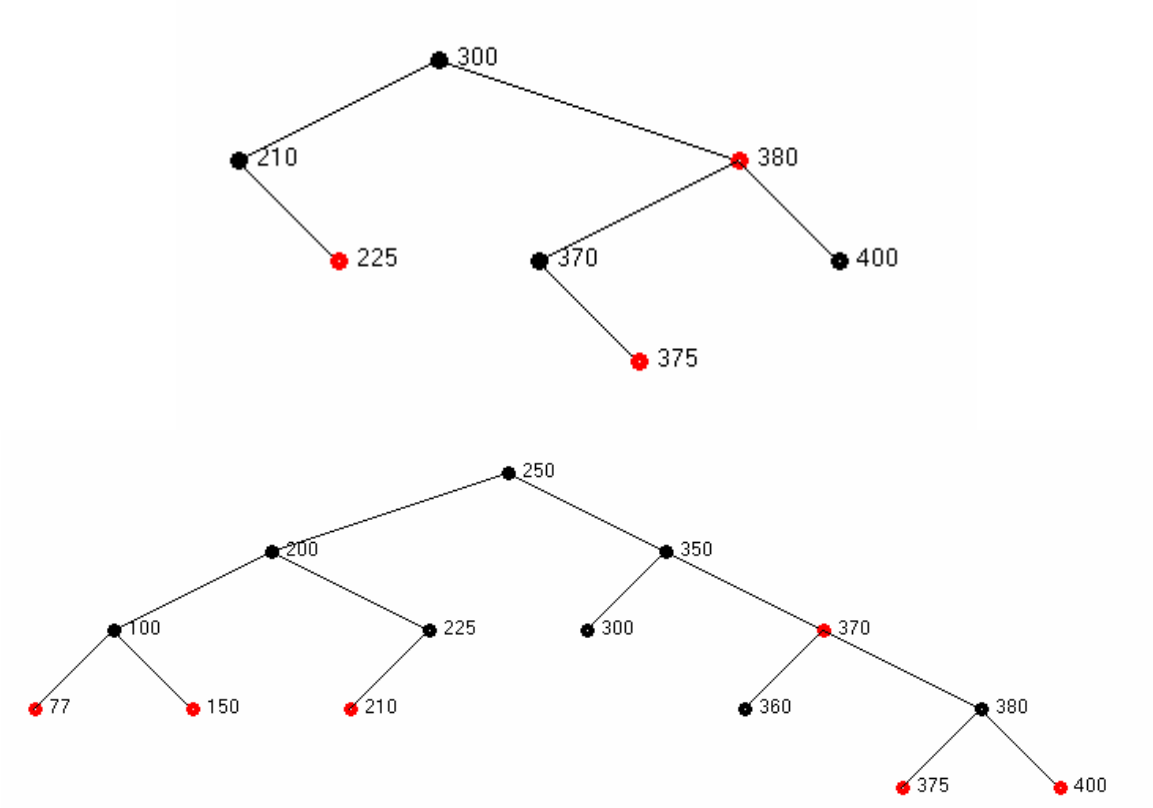


Figure 1 – Two Examples of Red-Black Trees

The algorithms for insertion and deletion involve node rotations as well as node re-coloring.

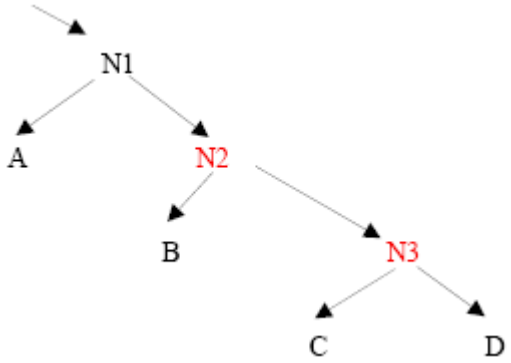
The algorithms with examples of insertion and deletion are presented first. Following this a complete implementation of class *RedBlackTree* is presented that includes all the implementation details for insertion and deletion.

Mechanism for insertion into a red-black tree

Perform an ordinary binary search tree insertion into the red-black tree. If the path from root to the new leaf node that contains the information being inserted passes through a node that contains two red children, re-color these nodes black and re-color the node with the two previously red children black, assuming that it is not the root node (if it is, leave the node black since the root node of a red-black tree must be black).

Color the new leaf node just inserted red.

If the steps above lead to a succession of two red nodes in moving from the root down the tree, a rotational correction is needed. There are four possible rotations that are possible. Two of these are illustrated in Figures 2(a), (b). The other two are mirror images.



Left rotate on N2 and color N1 red and N2 black.

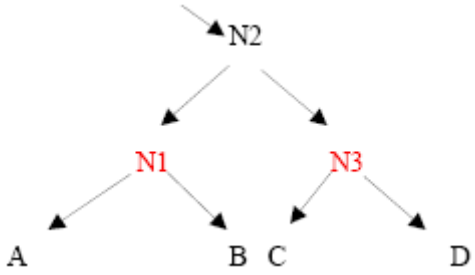


Figure 2a – Rotational and Recolor Correction

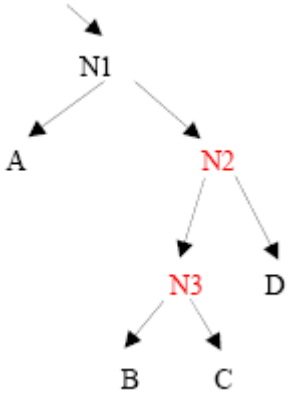


Figure 2b – Rotational and Recolor Correction

Right-rotate on N2 then left-rotate on N3 and perform appropriate re-coloring (details to be provided later).

Consider the sequence of insertions shown in Figure 3. After each insertion, the new red-black tree is displayed.

When node 50 is added, the search path passes through node 200 that contains two red nodes. This causes these nodes to be re-colored black.

After node 250 is added, the insertion of 275 causes the rotation and re-coloring shown in Figure 3d. After inserting 280, the final tree is shown in Figure 3e. No rotations are required.

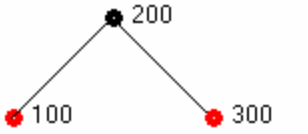


Figure 3a – Initial Red-Black Tree

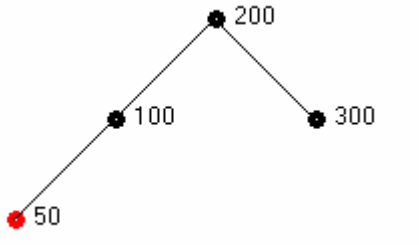


Figure 3b – Red-Black Tree After Inserting 50

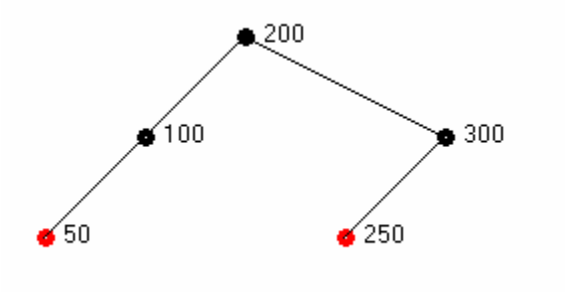


Figure 3c – Red-Black Tree After Inserting 250

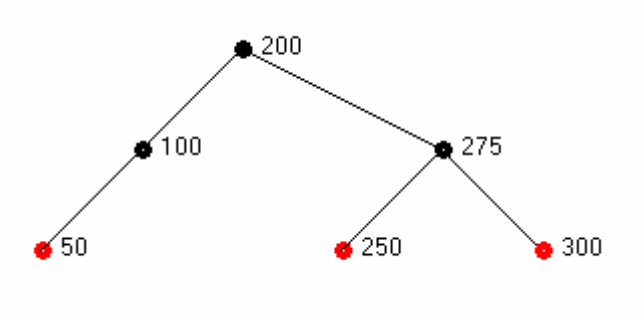


Figure 3d – Red-Black Tree After Inserting 275

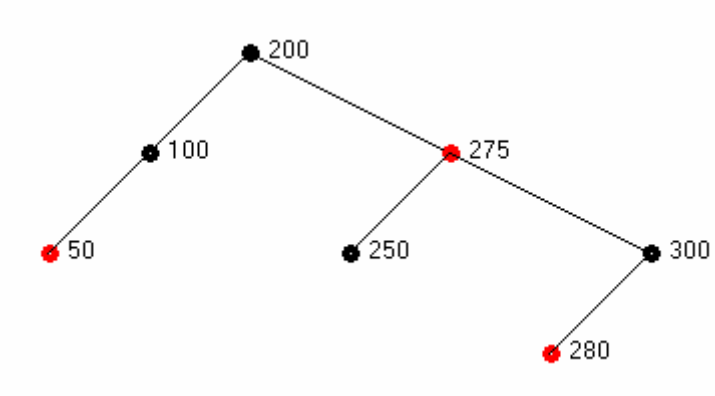


Figure 3e – Red-Black Tree After Inserting 280

Mechanism for deletion from a red-black tree

The algorithm for deletion is more complex than the algorithm for insertion. There are seven special cases to consider.

The first step is to perform an ordinary binary search tree deletion. In the case where the node being deleted has two children, we copy the value but not the color of the in-order successor. Then we delete the in-order successor, a node that can have at most one child.

If the node being deleted is colored red, no further corrections are needed. If the deleted node is black and it has a red right child, the red right child is re-colored to black and this serves to restore the tree to red-black status.

When the node being deleted node is black and it has no right child or a black right child, the following table indicates which of the seven cases needs to be utilized.

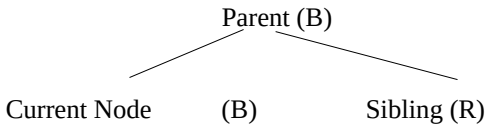
This table is an adaptation of Figure 21.8 in Berman and Paul's book cited above. The asterisk indicates any color. R indicates red and B indicates black.

Case	**1**	**2a**	**2b**	**3**	**4**	**3'**	**4'**
Parent	B	B	R	*	*	*	*
Sibling	R	B	B	B	B	B	B
Sibling (LC)	B	B	B	R	*	B	R
Sibling (RC)	B	B	B	B	R	R	*

From the starting state, any of the states can be reached. From state 1, a transition to either state 2b, state 3 or 4 occurs. When in state 2a, a transition back to state 2a, 2b, 3 or 4 occurs. When in state 3, a transition to state 4 occurs. When in state 3', a transition to state 4' occurs.

We consider states 1, 2a, 2b, 3 and 4 below. States 3' and 4' are mirror images of states 3 and 4. In each diagram, Current Node represents the right child of the actual node deleted (null in many cases). The R or B in parentheses indicate the node color.

State 1



Change color of Parent and Sibling and left rotate on Parent. Make transition to case 2b, 3 or 4.

State 2a

(Same diagram as state 1)

Change color of Sibling to red and then redefine Current Node as Parent. Compute new sibling and parent. Make transition back to state 2a or to case 2b, 3 or 4. Often Current Node gets moved up the tree in state 2a as transitions occur from 2a back to 2a.

State 2b

(Same diagram as state 1)

Change color of sibling to red and end.

State 3

(Same diagram as state 1)

Change color of sibling to red and its left child to black and then right rotate on sibling. Make transition to case 4.

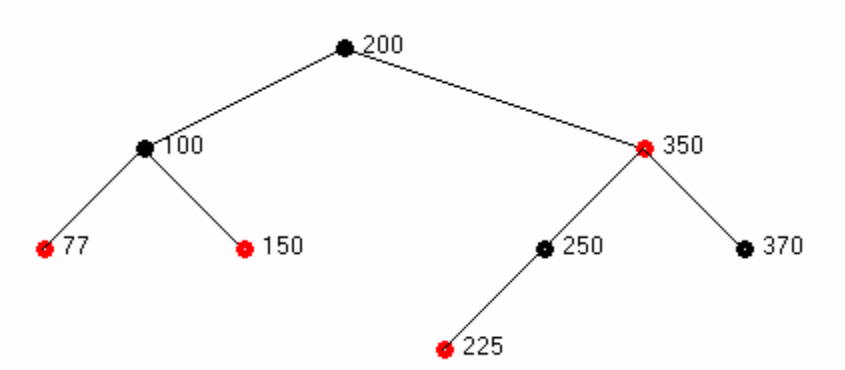
State 4

(Same diagram as state1)

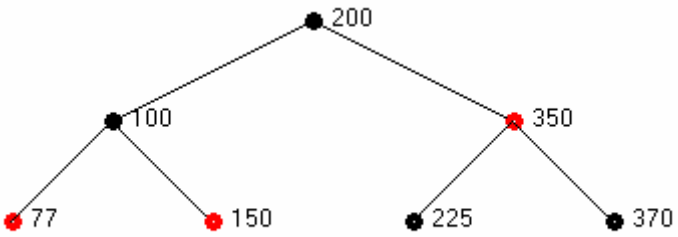
Change color of sibling's right child to black, make parent black, color sibling the color of parent and then left rotate on parent and end.

Several examples are presented that demonstrate some of the cases defined above.

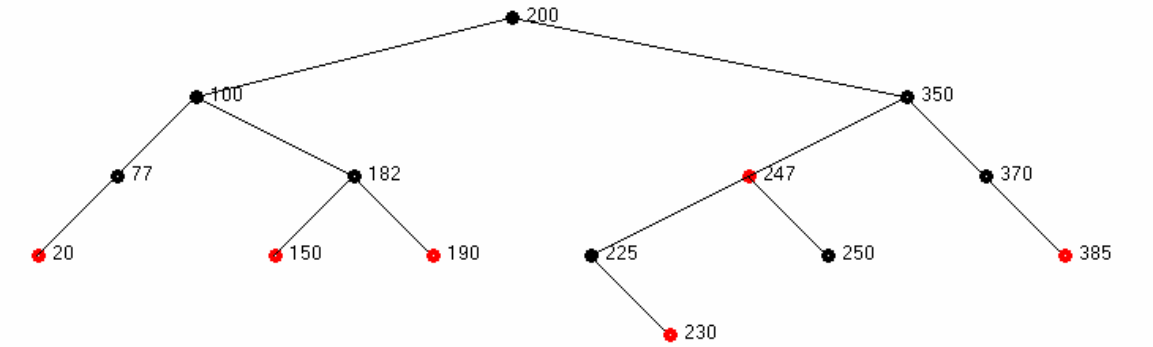
Consider the tree shown below. We wish to delete node 250. The right child of 250 is null so the Current Node is null. Its parent is 225 (after the deletion when the left child of 350 becomes 225) and its sibling is null. Since its parent is colored red, the system goes from the start state to state 2b.



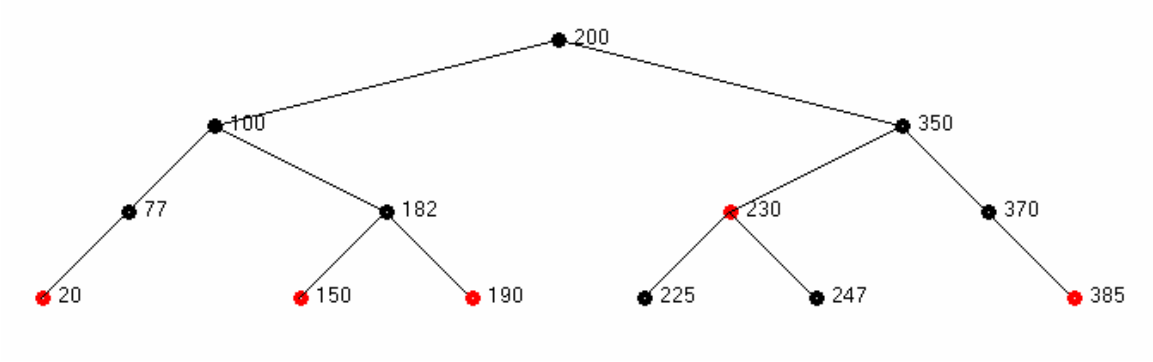
This leads to the result shown below in which the color of 225 is changed to black.



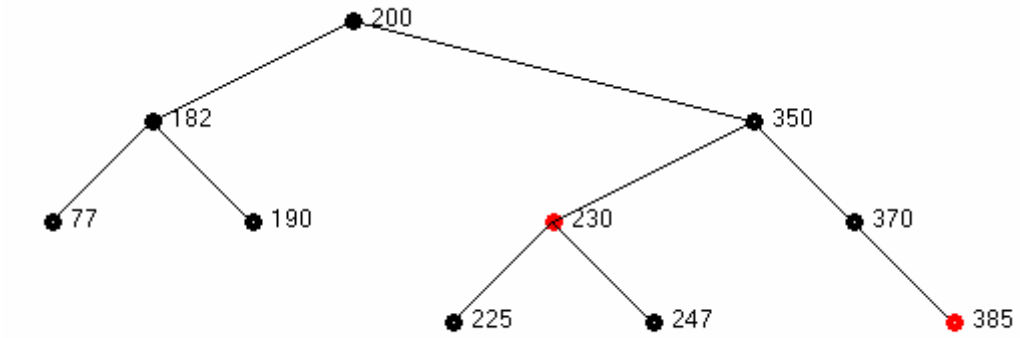
Consider the deletion of 250 from the following tree:



The table of states given above suggests that from the start state a transition to state 3' occurs. The Current Node is null, its parent is 247, its sibling is 225 and its sibling's right child is red. These are the conditions needed for state 3'. After the node re-coloring indicated under state 3' above, a left rotate is done on the sibling node 225. A transition to state 4' occurs. After more re-coloring, a right rotate on node 247 brings red node 230 to the left of 350. The tree resulting from states 3' and 4' is shown below.

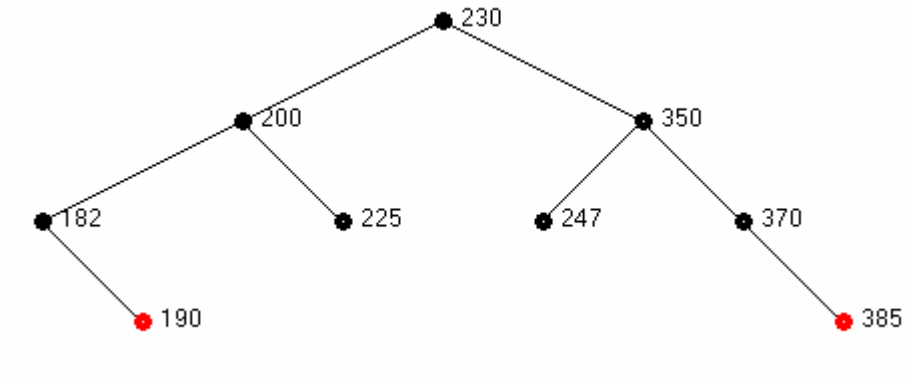


Consider the deletion of node 77 from the following red-black tree:



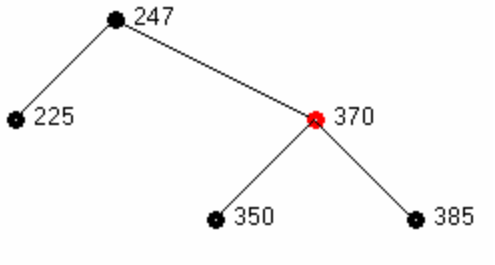
From the start state a transition is made to state 2a since the parent, node 182, is black, the sibling, 190 is black and the children of the sibling (null) are black.

From state 2a a transition is made to state 3 and then state 4. The final tree after these three states is shown below.



Unless one studies these transitions carefully, it simply looks like magic.

As the final example consider the deletion of node 225 from the following red-black tree:



From the start state a transition to state 1 occurs. This is because the sibling, node 370, is red and the parent, node 247 is black.

Two Alternative Designs of Red-Black Trees

The algorithm for insertion requires that the ancestors (parent, grand parent, great grand parent) of the inserted node be available. The implementation of deletion requires that the parent, sibling and grandparent nodes be available from the node being deleted. This leads to an important implementation question: should each node of a red-black tree have a link "pointing" to its parent (the root would link to a parent of null)? The alternative is to stay with the same structure as used in the implementation of the binary search tree and AVL tree in which each node is linked to its two children and not to its parent. The tradeoff is replacing more complexity in the basic operations of insertion and deletion with easier implementations of the various rotational and re-coloring "corrections" related to insertion and deletion.

The decision is to use the more complex node structure in which each node contains a link to its parent node since this leads to the most efficient implementation of the red-

black tree, particularly for deletion. An implementation that does not use links from child to parent requires a field, *stack*, and recursive *ConstructStack* method whose purpose is to determine the nodes in the path from root to any specified node. Because of the need to frequently invoke this method, the implementation for deletion is 100 times slower than an AVL tree with the same node values. This is unacceptable and leads to the focus on the version that contains links from child to parent nodes.

Comparing performance of binary search trees

The results of a simulation that compares the execution time required for inserting a sequence of values and later removing them from an unbalanced binary search tree, an AVL tree and a Red-Black tree are presented first. The two types of red-black tree implementations, with and without links to parent nodes, are included in the table even though only the implementation of the version with links to parent nodes will be presented. An ascending sequence of integers of size shown is used to construct each tree. Then all the values that were used to construct the tree are used in a sequence of deletions.

Tree Type	Number Nodes	Time for Insertion	Time for Deletion	ACE Value of Tree
Binary Search Tree	10,000	34.48 seconds		5000.5
Red-Black Tree (No links to parent nodes)	10,000	0.0312 seconds	2.844 seconds	12.88
Red-Black Tree (Links from child to parent nodes)	10,000	0.0156 seconds	0.0156 seconds	12.88
AVL Tree	10,000	0.0156 seconds	0.0156 seconds	12.36
Red-Black Tree (Links from child to parent nodes)	1,000,000	3.609	1.0625	19.33
AVL Tree	1,000,000	2.406 seconds	1.031	18.95

It is clear from the table above that the performance of the red-black tree with each node linking to its parent is superior to the red-black tree without such links to parent nodes and comparable to the performance of the AVL tree.

Implementation of class Node with upward links

The details of class *Node* are presented in Listing 1. It is a nested class (similar to Java's static inner class) in a generic class with parameter *T* (as seen later) so it does not require its own generic parameter.

Listing 1 – Class Node

```
private class Node {
     // Fields
     private T item; // Generic object held by each node
     private Node left, right, parent;  // Links to children and parent
     private bool red = true; // Color of node

     // Constructor
     public Node(T item, Node parent) {
          this.item = item;
          this.parent = parent;
          left = right = null;
     }

     // Properties
     public Node Left {
          get {
               return left;
          }
          set {
               left = value;
          }
     }

     public Node Right {
          get {
               return right;
          }
          set {
               right = value;
          }
     }

     public Node Parent {
          get {
               return parent;
          }
          set {
               parent = value;
          }
     }

     // Similar get/set properties for Item and Red
}
```

Tree rotation with upward links to parent nodes

The methods *LeftRotate* and *RightRotate* must take the *parent* link into account. Method *LeftRotate* is presented in Listing 2. Method *RightRotate* is the symmetrical opposite and not presented. The reference parameter capability of C# (*ref* parameter) is utilized in the two rotation methods. This allows explicit linking between the tree structure unaffected by the rotation and the new node that is passed back through the reference parameter.

Listing 2 – Revised methods LeftRotate

```
private void LeftRotate(ref Node node) {
     Node nodeParent = node.Parent;
     Node right = node.Right;
     Node temp = right.Left;
     right.Left = node;
     node.Parent = right;
     node.Right = temp;
     if (temp != null) {
          temp.Parent = node;
     }
     if (right != null) {
          right.Parent = nodeParent;
     }
      node = right;
}
```

After setting the right child of ref variable *node* to *temp*, the parent of *temp* is set to *node* (if *temp* is not null). This reflects the new parent of *temp* after the rotation is completed. The parent of *right* is set to *nodeParent*, a local variable that is immediately assigned as the first line in the method. This reflects the upward connection between the node being passed back (*right*) and its parent (not affected by the rotation). These two assignments using *Parent* ensure that the two links that are modified by the rotation are linked in both directions.

Implementation of insertion

The public method *Add* and its private support methods *InsertNode*, *GetNodesAbove* and *FixTreeAfterInsertion* implement the red-black algorithm for insertion.

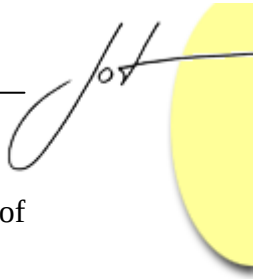
These methods are presented in Listing 3.

Listing 3 – Methods that support insertion in a red-black tree

```
public void Add(T item) {
     root = InsertNode(root, item, null);
     numberElements++;
     if (numberElements > 2) {
          Node parent, grandParent, greatGrandParent;
          GetNodesAbove(insertedNode, out parent,
                              out grandParent, out greatGrandParent);
          FixTreeAfterInsertion(insertedNode, parent,
                                   grandParent, greatGrandParent);
     }
}

private Node InsertNode(Node node, T item, Node parent) {
     if (node == null) {
          Node newNode = new Node(item, parent);
          if (numberElements > 0) {
               newNode.Red = true;
          } else {
               newNode.Red = false;
          }
          insertedNode = newNode;
```

```
            return newNode;
        } else if (item.CompareTo(node.Item) < 0) {
            node.Left = InsertNode(node.Left, item, node);
            return node;
        } else if (item.CompareTo(node.Item) > 0) {
            node.Right = InsertNode(node.Right, item, node);
            return node;
        } else {
            throw new InvalidOperationException(
                        "Cannot add duplicate object.");
        }
    }

    private void GetNodesAbove(Node curNode, out Node parent,
                               out Node grandParent,
                               out Node greatGrandParent) {
        parent = null;
        grandParent = null;
        greatGrandParent = null;
        if (curNode != null) {
            parent = curNode.Parent;
        }
        if (parent != null) {
            grandParent = parent.Parent;
        }
        if (grandParent != null) {
            greatGrandParent = grandParent.Parent;
        }
    }

    private void FixTreeAfterInsertion(Node child, Node parent,
                                       Node grandParent,
                                       Node greatGrandParent) {
        if (grandParent != null) {
            Node uncle = (grandParent.Right == parent) ?
                               grandParent.Left : grandParent.Right;
            if (uncle != null && parent.Red && uncle.Red) {
                uncle.Red = false;
                parent.Red = false;
                grandParent.Red = true;
                Node higher = null;
                Node stillHigher = null;
                if (greatGrandParent != null) {
                    higher = greatGrandParent.Parent;
                }
                if (higher != null) {
                    stillHigher = higher.Parent;
                }
                FixTreeAfterInsertion(grandParent,       greatGrandParent,
                                      higher, stillHigher);
            } else if (uncle == null || parent.Red && !uncle.Red) {
                if (grandParent.Right == parent &&
                        parent.Right == child) { // right-right case
                    parent.Red = false;
                    grandParent.Red = true;
                    if (greatGrandParent != null) {
                        if (greatGrandParent.Right == grandParent) {
                            LeftRotate(ref grandParent);
                            greatGrandParent.Right = grandParent;
                        } else {
                            LeftRotate(ref grandParent);
```

```
                        greatGrandParent.Left = grandParent;
                    }
                } else {
                    LeftRotate(ref root);
                }
            } else if (grandParent.Left == parent &&
                    parent.Left == child) { // left-left case
                parent.Red = false;
                grandParent.Red = true;
                if (greatGrandParent != null) {
                    if (greatGrandParent.Right == grandParent) {
                        RightRotate(ref grandParent);
                        greatGrandParent.Right = grandParent;
                    } else {
                        RightRotate(ref grandParent);
                        greatGrandParent.Left = grandParent;
                    }
                } else {
                    RightRotate(ref root);
                }
                } else if (grandParent.Right == parent &&
                        parent.Left == child) {// right-left case
                    child.Red = false;
                    grandParent.Red = true;
                    RightRotate(ref parent);
                    grandParent.Right = parent;
                    if (greatGrandParent != null) {
                        if (greatGrandParent.Right == grandParent) {
                            LeftRotate(ref grandParent);
                            greatGrandParent.Right = grandParent;
                        } else {
                            LeftRotate(ref grandParent);
                            greatGrandParent.Left = grandParent;
                        }
                    } else {
                        LeftRotate(ref root);
                    }
                } else if (grandParent.Left == parent &&
                            parent.Right == child) {// left-right case
                    child.Red = false;
                    grandParent.Red = true;
                    LeftRotate(ref parent);
                    grandParent.Left = parent;
                    if (greatGrandParent != null) {
                        if (greatGrandParent.Right == grandParent) {
                            RightRotate(ref grandParent);
                            greatGrandParent.Right = grandParent;
                        } else {
                            RightRotate(ref grandParent);
                            greatGrandParent.Left = grandParent;
                        }
                    } else {
                        RightRotate(ref root);
                    }
                }
            }
            if (root.Red) {
                root.Red = false;
            }
        }
    }
```



The recursive *InsertNode* contains a parameter, *parent*, which allows *node* at one level of recursion to be passed as *parent* to the next lower level of recursion.

The support method *GetNodesAbove* use the *out* parameter facility of C# to return relevant nodes above the current node (*curNode*). This can be easily accomplished because of the upward links in the node structure.

The important support method *FixTreeAfterInsertion* with four nodes as input implements the details of the red-black insertion algorithm as outlined and illustrated above. Care has been taken to use descriptive local variable and parameter names to make it easier for the code to be self-documenting

Implementation of deletion

The implementation of deletion is more complex than the implementation of insertion. This reflects the additional complexity of the algorithm for deletion which as you have seen earlier involves seven special cases and transitions from some of the cases to other cases. The remaining portion of class *RedBlack* that includes all the support for deletion is presented in Listing 4.

The generic class uses a constrained generic parameter *T* where *T* implements the *IComparable* interface.

Listing 4 – Remaining details of class RedBlackTree

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;

namespace SearchTrees {

    public class RedBlackTree<T> where T : IComparable {

        // Fields
        private Node root;
        private int numberElements;
        private Node insertedNode;
        private Node nodeBeingDeleted; // Set in DeleteNode
        private bool siblingToRight;   // Sibling of curNode
        private bool parentToRight;    // Of grand parent
        private bool nodeToDeleteRed;  // Color of deleted node

        // Commands
        public void Add(T item) {
            // Presented in previous listing
        }

       public void Remove(T item) {
           if (numberElements > 1) {
               root = DeleteNode(root, item, null);
               numberElements--;
               if (numberElements == 0) {
                   root = null;
```

```
            }
            Node curNode = null; // Right node being deleted
            if (nodeBeingDeleted.Right != null) {
                curNode = nodeBeingDeleted.Right;
            }
            Node parent, sibling, grandParent;
            if (curNode == null) {
                parent = nodeBeingDeleted.Parent;
            } else {
                parent = curNode.Parent;
            }
            GetParentGrandParentSibling(curNode, parent,
                                out sibling, out grandParent);

            if (curNode != null && curNode.Red) {
                curNode.Red = false;
            } else if (!nodeToDeleteRed && !nodeBeingDeleted.Red) {
                FixTreeAfterDeletion(curNode, parent,
                                        sibling, grandParent);
            }
            root.Red = false;
        }
    }

    // Queries
    private Node InsertNode(Node node, T item, Node parent) {
        // Presented in previous listing
    }

    private void RightRotate(ref Node node) {
        // Presented earlier
    }

    private void LeftRotate(ref Node node) {
        // Presented earlier
    }

    private void GetNodesAbove(Node curNode, out Node parent,
                                out Node grandParent,
                                out Node greatGrandParent) {
        // Presented in previous listing
    }

    private void GetParentGrandParentSibling(Node curNode,
                        Node parent,
                        out Node sibling, out Node grandParent) {
        sibling = null;
        grandParent = null;

        if (parent != null) {
            if (parent.Right == curNode) {
                siblingToRight = false;
                sibling = parent.Left;
            }
            if (parent.Left == curNode) {
                siblingToRight = true;
                sibling = parent.Right;
            }
        }
        if (parent != null) {
            grandParent = parent.Parent;
```

```
        }
        if (grandParent != null) {
            if (grandParent.Right == parent) {
                parentToRight = true;
            }
            if (grandParent.Left == parent) {
                parentToRight = false;
            }
        }
    }

    private void FixTreeAfterInsertion(Node child, Node parent,
                        Node grandParent,
                        Node greatGrandParent) {
        // Presented in previous listing
    }

    private Node DeleteNode(Node node, T item, Node parent) {
          private Node DeleteNode(Node node, T item, Node parent) {
              if (node == null) {
                  throw new InvalidOperationException(
                                  "item not in search tree.");
              }
              if (item.CompareTo(node.Item) < 0) {
                  node.Left = DeleteNode(node.Left, item, node);
              } else if (item.CompareTo(node.Item) > 0) {
                  node.Right = DeleteNode(node.Right, item, node);
              } else if (item.CompareTo(node.Item) == 0) {
                  // Item found
                  nodeToDeleteRed = node.Red;
                  nodeBeingDeleted = node;
                  if (node.Left == null) {
                      // No children or only a right child
                      node = node.Right;
                      if (node != null) {
                          node.Parent = parent;
                      }
                  } else if (node.Right == null) {
                      // Only a left child
                      node = node.Left;
                      node.Parent = parent;
                  } else { // Two children
                      // Deletes using the leftmost node of the
                      // right subtree
                      T replaceWithValue = LeftMost(node.Right);
                      node.Right =
                          DeleteLeftMost(node.Right, node);
                      node.Item = replaceWithValue;
                  }
              }
              return node;
          }
    }

    private Node DeleteLeftMost(Node node, Node parent) {
          if (node.Left == null) {
              nodeBeingDeleted = node;
              if (node.Right != null) {
                  node.Parent = parent;
              }
               return node.Right;
```

```
        } else {
            node.Left = DeleteLeftMost(node.Left, node);
            return node;
        }
    }

    private T LeftMost(Node node) {
        if (node.Left == null) {
            return node.Item;
        } else {
            return LeftMost(node.Left);
        }
    }

    private void FixTreeAfterDeletion(Node curNode, Node parent,
                                Node sibling, Node grandParent) {
        Node siblingLeftChild = null;
        Node siblingRightChild = null;
        if (sibling != null && sibling.Left != null) {
            siblingLeftChild = sibling.Left;
        }
        if (sibling != null && sibling.Right != null) {
            siblingRightChild = sibling.Right;
        }
        bool siblingRed = (sibling != null && sibling.Red);
        bool siblingLeftRed = (siblingLeftChild != null
                                        && siblingLeftChild.Red);
        bool siblingRightRed = (siblingRightChild != null &&
                                            siblingRightChild.Red);

        if (parent != null && !parent.Red && siblingRed &&
                            !siblingLeftRed && !siblingRightRed) {
            Case1(curNode, parent, sibling, grandParent);
        } else if (parent != null && !parent.Red &&
              !siblingRed && !siblingLeftRed && !siblingRightRed) {
            Case2A(curNode, parent, sibling, grandParent);
        } else if (parent != null && parent.Red &&
              !siblingRed && !siblingLeftRed && !siblingRightRed) {
            Case2B(curNode, parent, sibling, grandParent);
        } else if (siblingToRight && !siblingRed &&
                            siblingLeftRed && !siblingRightRed) {
            Case3(curNode, parent, sibling, grandParent);
        } else if (!siblingToRight &&
              !siblingRed && !siblingLeftRed && siblingRightRed) {
            Case3P(curNode, parent, sibling, grandParent);
        } else if (siblingToRight && !siblingRed &&
                                            siblingRightRed) {
            Case4(curNode, parent, sibling, grandParent);
        } else if (!siblingToRight && !siblingRed &&
                                                siblingLeftRed) {
            Case4P(curNode, parent, sibling, grandParent);
        }
    }

    private void Case1(Node curNode, Node parent,
                        Node sibling, Node grandParent) {
        if (siblingToRight) {
            parent.Red = !parent.Red;
            sibling.Red = !sibling.Red;
            if (grandParent != null) {
                if (parentToRight) {
```

```
                LeftRotate(ref parent);
                grandParent.Right = parent;
            } else if (!parentToRight) {
                LeftRotate(ref parent);
                grandParent.Left = parent;
            }
        } else {
            LeftRotate(ref parent);
            root = parent;
        }
        grandParent = sibling;
        parent = parent.Left;
        parentToRight = false;
    } else if (!siblingToRight) {
        parent.Red = !parent.Red;
        sibling.Red = !sibling.Red;
        if (grandParent != null) {
            if (parentToRight) {
                RightRotate(ref parent);
                grandParent.Right = parent;
            } else if (!parentToRight) {
                RightRotate(ref parent);
                grandParent.Left = parent;
            }
        } else {
            RightRotate(ref parent);
            root = parent;
        }
        grandParent = sibling;
        parent = parent.Right;
        parentToRight = true;
    }

    if (parent.Right == curNode) {
        sibling = parent.Left;
        siblingToRight = false;
    } else if (parent.Left == curNode) {
        sibling = parent.Right;
        siblingToRight = true;
    }

    Node siblingLeftChild = null;
    Node siblingRightChild = null;
    if (sibling != null && sibling.Left != null) {
        siblingLeftChild = sibling.Left;
    }
    if (sibling != null && sibling.Right != null) {
        siblingRightChild = sibling.Right;
    }
    bool siblingRed = (sibling != null && sibling.Red);
    bool siblingLeftRed = (siblingLeftChild != null &&
                           siblingLeftChild.Red);
    bool siblingRightRed = (siblingRightChild != null &&
                            siblingRightChild.Red);
    if (parent.Red && !siblingRed && !siblingLeftRed &&
                      !siblingRightRed) {
        Case2B(curNode, parent, sibling, grandParent);
    } else if (siblingToRight && !siblingRed && siblingLeftRed
                              && !siblingRightRed) {
        Case3(curNode, parent, sibling, grandParent);
```

```
        } else if (!siblingToRight && !siblingRed &&
                   !siblingLeftRed && siblingRightRed) {
            Case3P(curNode, parent, sibling, grandParent);
        } else if (siblingToRight && !siblingRed &&
                   siblingRightRed) {
            Case4(curNode, parent, sibling, grandParent);
        } else if (!siblingToRight && !siblingRed &&
                                      siblingLeftRed) {
            Case4P(curNode, parent, sibling, grandParent);
        }
    }

    private void Case2A(Node curNode, Node parent,
                        Node sibling, Node grandParent) {
        if (sibling != null) {
            sibling.Red = !sibling.Red;
        }
        curNode = parent;
        if (curNode != root) {
            parent = curNode.Parent;
            GetParentGrandParentSibling(curNode, parent,
                              out sibling, out grandParent);
            Node siblingLeftChild = null;
            Node siblingRightChild = null;
            if (sibling != null && sibling.Left != null) {
                siblingLeftChild = sibling.Left;
            }
            if (sibling != null && sibling.Right != null) {
                siblingRightChild = sibling.Right;
            }
            bool siblingRed = (sibling != null && sibling.Red);
            bool siblingLeftRed = (siblingLeftChild != null &&
                                      siblingLeftChild.Red);
            bool siblingRightRed = (siblingRightChild != null &&
                                      siblingRightChild.Red);
            if (parent != null && !parent.Red && !siblingRed &&
                          !siblingLeftRed && !siblingRightRed) {
                Case2A(curNode, parent, sibling, grandParent);
            } else if (parent != null && parent.Red && !siblingRed
                       && !siblingLeftRed && !siblingRightRed) {
                Case2B(curNode, parent, sibling, grandParent);
            } else if (siblingToRight && !siblingRed &&
                          siblingLeftRed && !siblingRightRed) {
                Case3(curNode, parent, sibling, grandParent);
            } else if (!siblingToRight && !siblingRed &&
                          !siblingLeftRed && siblingRightRed) {
                Case3P(curNode, parent, sibling, grandParent);
            } else if (siblingToRight && !siblingRed &&
                             siblingRightRed) {
                Case4(curNode, parent, sibling, grandParent);
            } else if (!siblingToRight && !siblingRed &&
                          siblingLeftRed) {
                Case4P(curNode, parent, sibling, grandParent);
            }
        }
    }

    private void Case2B(Node curNode, Node parent,
                        Node sibling, Node grandParent) {
        if (sibling != null) {
            sibling.Red = !sibling.Red;
```

```
        }
        curNode = parent;
        curNode.Red = !curNode.Red;
    }

    private void Case3(Node curNode, Node parent,
                       Node sibling, Node grandParent) {
        if (parent.Left == curNode) {
            sibling.Red = true;
            sibling.Left.Red = false;
            RightRotate(ref sibling);
            parent.Right = sibling;
        }
        Case4(curNode, parent, sibling, grandParent);
    }

    private void Case3P(Node curNode, Node parent,
                       Node sibling, Node grandParent) {
        if (parent.Right == curNode) {
            sibling.Red = true;
            sibling.Right.Red = false;
            LeftRotate(ref sibling);
            parent.Left = sibling;
        }
        Case4P(curNode, parent, sibling, grandParent);
    }

    private void Case4(Node curNode, Node parent,
                           Node sibling, Node grandParent) {
        sibling.Red = parent.Red;
        sibling.Right.Red = false;
        parent.Red = false;
        if (grandParent != null) {
            if (parentToRight) {
                LeftRotate(ref parent);
                grandParent.Right = parent;
            } else {
                LeftRotate(ref parent);
                grandParent.Left = parent;
            }
        } else {
            LeftRotate(ref parent);
            root = parent;
        }
    }

    private void Case4P(Node curNode, Node parent,
                           Node sibling, Node grandParent) {
        sibling.Red = parent.Red;
        sibling.Left.Red = false;
        parent.Red = false;
        if (grandParent != null) {
            if (parentToRight) {
                RightRotate(ref parent);
                grandParent.Right = parent;
            } else {
                RightRotate(ref parent);
                grandParent.Left = parent;
            }
        } else {
            RightRotate(ref parent);
```

```
                root = parent;
            }
        }

        private class Node {
            // Details presented earlier
        }
    }
}
```

Care was again taken in choosing variable and parameter names so that the code would be self-documenting. The complexity of the code reflects the complexity of the algorithm for deletion.

C#'s requirement that *ref* or *out* parameters must be explicitly tagged when invoking a method is desirable. It makes the semantics of the method invocation clear without requiring the programmer to refer back to the method signature. There are many examples of such method invocations in Listing 4.

In a future column, a GUI application that renders a graphical depiction of search trees as values are inserted in or deleted from ordinary binary search trees, AVL trees and red-black trees is presented. This GUI application was used as a test-bed while developing the code presented in this paper.

About the author

Richard Wiener is Associate Professor of Computer Science at the University of Colorado at Colorado Springs. He is also the Editor-in-Chief of JOT and former Editor-in-Chief of the Journal of Object Oriented Programming. In addition to University work, Dr. Wiener has authored or co-authored 21 books and works actively as a consultant and software contractor whenever the possibility arises.

2. Customized Implementation of Tree Map. if you need to Implement Custom Objects as a keys in Tree Map , you Must Implement the Comparator Interface and you Must Overridden the Compare Method between the Two Keys . Then only it will works. Letâ€™s see Below Example for the Better Understanding â€¦.. we have Employee custom object , we put Employee Object as a key and their PF amount as a Values in Tree Map with and without implementing the Comparator Interface. TreeMap with and without Comparator Interface implementation. OutputÂ Tree Map Internally Implements the Red Black Tree Data-structure internally. so letâ€™s see Red Black Tree and How itâ€™s Insertion Operation will works. 3. Properties of the Red Black Tree. It must obey the Binary Search Tree Property. C Implementation of K-dimensional Tree. Play with libgfapi and its python bindings.. Your category. Your category.Â (20) Debugging/GDB (10) Device-mapper-multipath (4) docker (13) Fedora (7) Filesystems (31) glusterfs (29) gluster-block (1) git (1) Go Programming (2) Kernel (25) kubernetes (36) kube-chapters (2) kubernetes operators (1) KVM/ Qemu (51) Lex (10) libvirt (25) linux-general (32) LVM (6) misc/general (40) openshift (24) Ovirt (39) Ovirt/RHEV (23) Programming (24) C programming (18) Python (5). Red Hat (6) rook (2) shell-scripts (2) Uncategorized (2) virt-tools (28) Virtualization (39) Xen (6) Yacc (2) à´®à´²à´¯à´¾à´³à´, à´¬àµà´²àµ‹à´—àµâ€Œ (5). Search your topic here! Search for Red-Black trees are ordered binary trees with one extra attribute in each node: the color, which is either red or black. Like the Treap, and the AVL Tree, a Red-Black tree is a self-balancing tree that automatically keeps the treeâ€™s height as short as possible. Since search times on trees are dependent on the treeâ€™s height (the higher the tree, the more nodes to examine), keeping the height as short as possible increases performance of the tree.Â This article provides a Red-BlackTree implementation in the C# language. Background.Â Every child node contains a value, or a key, that determines its position in the tree relative to its parent. Since the root node is the top parent, all nodes are organized relative to the root node in branches.