Implementation of Rotations in AVL Trees
Using java.util

Nicholas J. De Lillo
Randall Swan
Nicholas J. De Lillo is Professor of Mathematics and Computer Science at Manhattan College where he has taught courses in computer science, computer engineering, and software engineering at both the undergraduate and graduate levels for over thirty years. In addition, Professor De Lillo regularly teaches courses in the masters program in computer science here at Pace. He is on also on the Editorial Board of Technical Reports.

Professor De Lillo is the author of numerous research papers and seven textbooks in mathematics and computer science. His texts include Advanced Calculus with Applications (1982); Computability with Pascal, co-authored with John S. Mallozzi (1984); A First Course in Computer Science with Ada (1993); Data Structures with C++, co-authored with John S. Mallozzi (1997); Object-Oriented Design in C++ Using the Standard Template Library (2002); Object-Oriented Design in Java Using java.util (2004); and Data Structures Using Java 5.0 (2006).

Professor De Lillo holds a B.S. in mathematics from Manhattan College, an M.A. in mathematics from Fordham University, and the Ph.D. in mathematics from New York University, where he was a student of Martin Davis's.

Randall Swan holds a masters degree in computer science from Pace University.
IMPLEMENTATION OF ROTATIONS IN AVL TREES
USING JAVA.UTIL

Nicholas J. De Lillo
Department of Mathematics and Computer Science
Manhattan College
Riverdale, New York 10471

Randall Swan
Ivan G. Seidenberg School of Computer Science
and Information Systems
Pace University
White Plains, New York 10606-1932

Abstract

This paper extends a number of results first presented in [1] regarding implementations in Java of unbalanced binary trees, with special emphasis on binary trees first studied by Adel'son-Velskii and Landis in 1962 [2] and now known as AVL trees. We begin by giving a definition of AVL trees, and give a formal object-oriented implementation of the Binary Search Tree abstract data type. In addition, we provide an implementation in Java of a number of rotation methods that convert the original binary search tree into an AVL tree.

1. Introduction: Preliminary Definitions.

Binary search trees are special forms of binary trees whose design is both simple and extremely efficient as a nonlinear data structure used for searching and sorting. In fact, if n Comparable objects are stored in a binary search tree, the efficiency of searching such a structure for a value ranges from an $O(n)$ worst case to an $O(\log n)$ average case. The worst case involves the construction of a binary search tree whose shape is “short and fat,” since such trees optimize the efficiency of searching and sorting. Indeed, such “short and fat” trees yield searches than are approximately $O(\log n)$. In case the original tree is not “short and fat,” we consider clockwise and counterclockwise rotations of the tree that transform it into a balanced binary search tree, also known as an AVL tree.

In this paper, we describe one way of defining “short and fat,” and then propose a method that converts binary search trees into an equivalent form satisfying this definition. The optimal binary search tree for purposes of searching and sorting is a balanced tree, namely a binary search tree with the additional property that the height of any node’s right subtree differs from the height of the same node’s left subtree by no more than one.
This is commonly referred to as an **AVL tree**, named after its principal researchers, the Russian mathematicians Adelson-Velskii and Landis. We may observe that an AVL tree satisfies our criterion for a "short and fat" tree.

We then face a very interesting problem. Suppose our current binary search tree is not an AVL tree. Is it possible to provide algorithms that can be used to convert the current binary search tree into an AVL tree? One way of solving this problem is to rearrange the nodes of the tree to achieve better balance, and one way of accomplishing this objective is by rotation operations on the original tree.

3. **Binary Search trees.**

\( T_r \) is a binary search tree (BST) if either \( T_r \) is empty, or if \( T_r \) is a nonempty binary tree satisfying each of

- for each node \( u \) of \( T_r \), the left subtree rooted at \( u \) is either empty, or contains nodes, each of whose \texttt{info} component has a value less than that of \( u \);
- for each node \( u \) of \( T_r \), the right subtree rooted at \( u \) is either empty, or contains nodes, each of whose \texttt{info} component has a value greater than that of \( u \);
- for each node \( u \) of \( T_r \), the left and right subtrees rooted at \( u \) are both binary search trees.

We assume throughout this discussion that the nodes of a binary search tree have \texttt{info} components that are members of a single \texttt{Comparable} type, whether predefined or defined explicitly by the programmer.

We may define the binary search tree abstract data type (ADT) as an abstract data type that supports the following list of admissible operations:

1. test whether the current tree is empty;
2. retrieve a value stored in the tree (and test whether that value is currently stored in the tree);
3. insert a new node into the tree so that the result retains the structure of a binary search tree;
4. remove an existing value from the tree so that the result remains a binary search tree;
5. traverse the nodes of the current tree.

The implementation of this ADT in Java results in the construction of a class, whose objects will have nodes, all of which will lie in a single subclass of the \texttt{Comparable} class.

Our design will involve first defining a new class of nodes that are suitable for binary search trees. This class is called \texttt{BSTNode}, and is coded as follows:
class BSTNode
{
    private BSTNode leftChild;
    private BSTNode rightChild;
    private BSTNode parent;
    private Comparable info;

    // Constructor. Initializes binary search tree with info,
    // and left and right children, and parent references.
    public BSTNode(Comparable value, BSTNode left, BSTNode right,
        BSTNode parent)
    {
        this.leftChild = left;
        this.rightChild = right;
        this.parent = parent;
        this.info = value;
    }

    // Accessors.
    // Returns the reference to one of the fields of the node.
    public BSTNode getLeft(){return leftChild;}
    public BSTNode getRight(){return rightChild;}
    public BSTNode getParent(){return parent;}
    public Comparable getInfo(){return info;}

    // Sets the reference to one of the fields of the current node.
    public void setLeft(BSTNode x){leftChild = x;}
    public void setRight(BSTNode x){rightChild = x;}
    public void setParent(BSTNode x){parent = x;}
    public void setInfo(Comparable x){info = x;}
}

A BSTNode has the structure of (Figure 1):

<table>
<thead>
<tr>
<th>info</th>
<th>leftChild</th>
<th>rightChild</th>
<th>parent</th>
</tr>
</thead>
</table>

(Figure 1)

The class BinSrchTree will have two constructors. The first is the empty constructor

public BinSrchTree()
{
    root = null;
} // terminates text of the empty constructor
which constructs an empty binary search tree. The second constructor generates a binary search tree with a single nonempty root node whose info component is the value of the parameter. This is coded as

```java
public BinSrchTree(Comparable value)
{
    root = new BSTNode(value, null, null, null);
} //terminates text of constructor
```

The only instance variable of BinSrchTree is the designated root of the tree, coded as

```java
private BSTNode root;
```

The isEmpty method tests whether a binary search tree is empty, returning true if so, and false if not.

```java
public boolean isEmpty()
{
    return root == null;
} // terminates text of isEmpty
```

The method makeEmpty removes all of the nodes of an already existing BinSrchTree object, if that object is not already empty:

```java
public BSTNode makeEmpty()
{
    return root = null;
} // terminates text of makeEmpty
```

Due to the nature of binary search trees, insertion and removal operations must be done in a manner that maintains the integrity of the tree. That is to say, after an insertion or removal operation is performed, the resulting tree must retain the structure of a binary search tree. Each of the insertion, removal, and retrieval methods exploit the fact that the only point of entry into the tree is through the root. Each of these operations involves a search for the value to be retrieved or removed from the tree, or a search for the proper position in the tree where the new node is to be inserted. Thus, the root of the BinSrchTree object plays a critical role in each of these operations. It is therefore important to code the getRoot method for use in binary search trees. The code is given as

```java
public BSTNode getRoot()
{
    if (isEmpty()) // Current tree is empty
        throw new BinaryTreeException("retrieval operation aborted - current tree is empty");
    else return root;
} // terminates text of getRoot."
```

---

1For the description of the BinaryTreeException class, see [1], p. 348.
The design of the insertion method is of particular interest. We choose to design a method that inserts values, one at a time, into the existing binary search tree, so as to maintain the integrity of the tree. This first involves a search for the proper location of the new node in the tree, and then executing the necessary insertion operations of the new node into the existing tree, provided that the new node does not already appear in the tree.

The insertion operation is performed by executing a pair of methods. The first of these is insert, which is public, and requires a single Comparable parameter. This parameter holds the value of the info component of the node to be inserted into the tree. The insert method then invokes the auxiliary private recursive method insertValue, which does most of the computational work of inserting the value in its proper position in the tree. The reason for recursion is that it is impossible to predict the height of the resulting tree in advance. Consequently, in the coding of insertValue, the parameter defined as root not only will refer to the root of the entire tree, but also to the root of the (left or right) subtree at any specific level of the recursion. Thus, the coding of insertValue may be given by

```java
private void insertValue(BSTNode root, Comparable value) {
    // If value to insert is less than the value at the
    // current root, go to the left subtree.
    if (value.compareTo(root.getInfo()) < 0) {
        // If at a leaf, insert new node.
        if (root.getLeft() == null) {
            BSTNode newNode = new BSTNode(value, null, null, null);
            root.setLeft(newNode);
        } else {
            // Not at a leaf, so proceed down the tree
            insertValue(root.getLeft(), value);
        }
    } else {
        // If value to insert is greater than the current root,
        // go to the right subtree.
        // If at a leaf, insert new node.
        if (root.getRight() == null) {
            BSTNode newNode = new BSTNode(value, null, null, null);
            root.setRight(newNode);
        } else {
            // Not at a leaf, so proceed down the tree.
            insertValue(root.getRight(), value);
        }
    }
    // terminates text of insertValue.
}
```

The formal code for insert invokes insertValue, and is given by

```java
public void insert(Comparable value) {
    insertValue(root, value);
    // terminates text of insert method.
}
```
It is important to note that the setLeft and setRight methods alluded to in the text of insertValue have to be recoded to accommodate an appropriate linkage involving the parent component of the new node.

Intuitively, each of the operations of insertion, retrieval, or removal involves a search of the current binary search tree. The efficiency of these operations depends on the shape of the tree. If the current tree contains $n$ and if it is "short and fat," the complexity of the search is approximately $O(\log n)$. On the other hand, if the tree is nearly linear, the complexity is approximately $O(n)$.

The coding of the instance method for retrieving a value (if it exists) in a binary search tree amounts to searching the tree for that value. The value sought is one of the parameters passed to the retrieval method. The remaining parameter is a reference to a node of the tree, which we may assume is the root of the tree. The reason for including this last parameter is to facilitate the formal coding of the recursive algorithm used for the retrieval.

Similar to insertion, the retrieval of a value (if such a value already exists in the tree) is done by the retrieve method, callable by the user and requiring a single Comparable parameter—the value to be retrieved. The retrieve method in turn invokes the private and recursive method retrieveValue, returning a boolean value and requiring two parameters—the root of the tree to be searched, and the value sought. If retrieveValue returns true, this indicates that the value sought was found in the tree; and if false is returned, this indicates that the value sought was not found in the tree.

The coding for retrieveValue is given by

```java
private boolean retrieveValue(BSTNode root, Comparable value) {
    if (root != null) {
        if (value.equals(root.getInfo())) { // successful search
            return true;
        } else { // search is unsuccessful so far
            if (value.compareTo(root.getInfo()) < 0) { // value too small
                // Resume search in left subtree
                return retrieveValue(root.getLeft(), value);
            } else if (value.compareTo(root.getInfo()) > 0) { // value too big
                // Resume search in right subtree
                return retrieveValue(root.getRight(), value);
            }
        }
    }
    // If execution resumes here, search is unsuccessful.
    // Report unsuccessful search.
    return false;
}
```

and the code for retrieve is
// Retrieve method, callable by user and invoking retrieveValue.
public boolean retrieve(Comparable value)
{
    return retrieveValue(root, value);
} // terminates text of retrieve method.

The implementation of the removal of an existing value is by far the most complicated operations defined for a binary search tree. This is primarily due to the fact that, after searching for and locating the value to be removed, the resulting tree must retain the structure of a binary search tree. As in the coding of the insertion and retrieval methods, the remove method will rely on the functionality of four private methods, some of which are recursive. The public method accessible to the user is called remove, and takes a Comparable parameter: the value to be removed from the current binary search tree. It is coded as

public void remove(Comparable value)
    // Removes the node of the binary search tree whose
    // info component matches the value of the parameter.
{
    root = removeValue(root, value);
} // terminates text of remove

Note that remove invokes another method called removeValue. The removeValue method is private, and, with the participation of three other auxiliary methods, performs the actual work of removing the indicated node and restructures the result so that it retains the structure of a binary search tree. One of the reasons for much of the programming effort that follows is that there are three possible cases of configurations of nodes, and each case requires a different form of removal. The three possible cases are listed as

(Case 1) the node to be removed is a leaf;
(Case 2) the node to be removed has exactly one non-null child;
(Case 3) the node to be removed has two non-null children.

(Case 1) This is the simplest case. Here the only action to be taken is to redirect the parent’s link away from the node to be removed to null.

(Case 2) Here, the node to be removed has exactly one non-null child. The algorithm strategy to apply here is to have the parent of the node to be removed redirect its reference from the present node to the non-null child of the present node.

(Case 3) The final case involves removing an interior node having non-null left and right children. To do so, we require an important auxiliary definition. The inorder successor (whenever it exists) of any node in a binary search tree is the node whose value immediately follows that of the current node in the inorder traversal of the tree. The strategy then is to swap the node to be removed with that containing its inorder successor. The resulting tree then places the node to be removed into a situation involving either (Case 1) or (Case 2).
The auxiliary method `removeValue` is recursive and involves another private method called `deleteNode`. The code for `removeValue` is then given by

```java
private BSTNode removeValue(BSTNode node, Comparable value) {
    // Recursive. Deletes node from BSTree object.
    BSTNode ptr;
    if (node == null)
        throw new BinaryTreeException("Exception - value not found");
    else
    {
        Comparable nodeValue = node.getInfo();
        if (value.equals(nodeValue))
            // node sought is the root of some subtree
            node = deleteNode(node); // delete the node
        else if (value.compareTo(nodeValue) < 0) // node too small
            // search the left subtree
            { ptr = rremoveValue(node.getLeft(), value);
              node.setLeft(ptr);
            }
        else // value too big - search the right subtree
            { ptr = removeValue(node.getRight(), value);
              node.setRight(ptr);
            }
    } // terminates text of outer else-clause
    return node;
} // terminates text of removeValue.
```

The task of `removeValue` is to locate the position of the node in the tree that is to be removed. The actual removal of the node is performed by `deleteNode`, invoked in the text of `removeValue` as

```java
node = deleteNode(node);
```

Thus, at this point, `node` refers to the node to be removed. If that node is a leaf, as in (Case 1), \( (\text{node.getLeft()} == \text{null}) \&\& (\text{node.getRight()} == \text{null}) \) is true, and hence return node executes, returning `null`. The value `null` returned indicates that a leaf node was removed from the tree.

Suppose now that the node to be removed has exactly one non-null child (as in (Case 2)). Suppose that non-null child is the left child. Thus \( \text{root.getLeft()} == \text{null} \) is false and \( \text{root.getRight()} == \text{null} \) is true. When `removeNode` executes in this case, `root.setLeft(ptr)` is returned – a reference to the left child of the node to be removed is returned. This return value indicates that the node to be removed has been bypassed; the new left child becomes the left child of the node that has just been removed.
The code for `deleteNode` is given by

```java
// Auxiliary private method used by removeValue.
private BSTNode deleteNode(BSTNode node)
{
    Comparable replacementValue;
    if((node.getLeft() == null) && (node.getRight() == null))
        return null;
    // test for left child
    else if(node.getLeft() == null)
        return node.getRight();
    // test for right child
    else if(node.getRight() == null)
        return node.getLeft();
    else
    {
        replacementValue = getSuccessor(node.getRight());
        node.setInfo(replacementValue);
        node.setRight(removeSUccessor(node.getRight()));
        return node;
    } // terminates text of last else-clause
} // terminates text of deleteNode
```

Finally, if neither the left child or the right child of the node to be removed is null, execution in `deleteNode` passes to the sequence

```java
replacementValue = getSuccessor(node.getRight());
node.setInfo(replacementValue);
node.setRight(removeSUccessor(node.getRight()));
return node;
```

Here we use the following key observation. Assume some fixed node in a binary search tree has two non-null children. The inorder successor of that node is the leftmost node in the right subtree rooted at that node.

The task performed by `getSuccessor` is to determine the value of the `info` component of the leftmost node in the right subtree rooted at the node to be removed. Its code is given by

```java
private Comparable getSuccessor(BSTNode node)
// Recursive
{
    if(node.getLeft() == null)
        return node.getInfo();
    else
    {
        return getSuccessor(node.getLeft());
    } // terminates text of getSuccessor
```

The role of `removeSuccessor` is to first copy the value of the inorder successor into the `info` component of the node that is to be removed, and then removing the leaf or the
node with exactly one non-null child holding the other copy. The code for removeSuccessor is given by

```java
private BSTNode removeSuccessor(BSTNode node) {
    if (node.getLeft() == null)
        return node.getRight();
    else
    { 
        node.setLeft(removeSuccessor(node.getLeft()));
        return node;
    }  // terminates text of last else-clause
    }  // terminates text of removeSuccessor
```

4. Rotations and Their Implementation.

In this section, we provide the necessary code for implementing rotation operations needed to convert ordinary binary search trees into AVL trees. Consider the example of the Integer-valued\(^2\) tree of (Figure 2) below:

(Figure 2)

\(^2\) Here Integer refers to the wrapper class associated with int.
Clearly, this tree is not balanced, but it can be converted into the AVL tree described in (Figure 3) by performing a single clockwise rotation. Also note that the height of the tree of (Figure 2) is 4, while that of the tree of (Figure 3) is 3.

(Figure 3)

This is an example of the result of applying a single clockwise rotation. In more general terms, suppose we have a subtree of a binary search tree as described in (Figure 4) below:

(Figure 4)

Here, \( P \) represents the original parent node of the child \( C \), \( C_1 \) represents the left subtree rooted at \( C \), \( C_2 \) is the right subtree rooted at \( C \), and \( C_3 \) is the right subtree rooted at \( P \). The result of applying a clockwise rotation is the tree described in (Figure 5).
Note that after the single clockwise rotation, the integrity of the binary search tree is preserved. Indeed, in either tree, any node (if such exists) in the subtree $C_1$ has a value less than $C$ (and $P$); any node (if such exists) in $C_2$ has a value greater than that of $C$ but less than that of $P$, and any node (if it exists) in $C_3$ has a value greater than that of $P$.

How may we implement a single clockwise rotation in Java? The following code defines a non-static method for performing a single clockwise rotation.

```java
// Performs a single clockwise rotation around a node.
public BSTNode rotateClockwise(BSTNode oldRoot) {
    BSTNode parent = oldRoot.getParent(); // hold onto parent
    BSTNode newRoot = oldRoot.getLeft(); // left child must exist
    try {
        BSTNode nrRight = newRoot.getRight(); // right child of newRoot
        oldRoot.setLeft(nrRight);
        newRoot.setRight(oldRoot);
        newRoot.setParent(parent); // reset parent node leading into subtree
        if (parent != null) { // have a normal subtree rotation
            // are we hanging off of the right subtree or the left subtree?
            if (oldRoot.equals(parent.getRight()))
                parent.setRight(newRoot);
            else if (oldRoot.equals(parent.getLeft()))
                parent.setLeft(newRoot);
        } else // parent == null: special case rotating about the true root
            root = newRoot; // important to reset root
    } // terminates text of try-block
    catch (Exception e) { // catches a null newRoot
        }
```
System.out.println(" FAILED rotateClockwise " + e);
}
return newRoot;
} // terminates text of rotateClockwise method.

Now consider the case of the Integer-valued tree of (Figure 6):

(Figure 6)

Again, this tree is not balanced. However, we can transform this tree into the AVL tree depicted in (Figure 7) by a single counterclockwise rotation. Also note that the height of the tree of (Figure 6) is 4, while that of (Figure 7) is 3.

(Figure 7)
In general, suppose we have a subtree of a binary search tree as described in (Figure 8):

(Figure 8)

Here P and C are as before, C₁ is the left subtree rooted at P, C₂ is the left subtree rooted at C, and C₃ is the right subtree rooted at C. Then a single counterclockwise rotation about P produces the binary search tree depicted in (Figure 9):

(Figure 9)

The following non-static method may be used to implement counterclockwise rotations.
// Performs a counterclockwise rotation about a node.
public BSTNode rotateCounterClockwise(BSTNode oldRoot)
{
    BSTNode parent = oldRoot.getParent(); // hold onto parent
    BSTNode newRoot = oldRoot.getRight(); // right child must exist
    try
    {
        BSTNode nrLeft = newRoot.getLeft(); // left child of newRoot
        oldRoot.setRight(nrLeft);
        newRoot.setLeft(oldRoot);
        newRoot.setParent(parent); // reset parent node leading into subtree
        if(parent != null) // is a normal subtree rotation
        {
            // are we hanging off of the right or the left subtree?
            if(oldRoot.equals(parent.getRight()))
                parent.setRight(newRoot);
            else // if(oldRoot.equals(parent.getLeft()))
                parent.setLeft(newRoot);
        }
        else // if(parent == null): special case rotating about the true
        // root
        root = newRoot; // important to reset root
    } // terminates text of try-block
    catch(Exception e) // catches a null newRoot
    {
        System.out.println(" FAILED rotateCounterClockwise :" + e);
    }
    return newRoot;
} // terminates text of rotateCounterClockwise method

There are situations in which a single rotation in either direction is not sufficient to
convert a binary search tree into an AVL tree. In such cases, a sequence of clockwise
and/or counterclockwise rotations about several different nodes of the existing tree may
be required to convert the tree into an AVL tree. For commercial purposes, it may well
be worth the effort required to accomplish the conversion.

5. Bibliography.

[1] De Lillo, Nicholas J., Data Structures Using Java 5.0, Stipes Publishing L.L.C.,
2006.

The Ivan G. Seidenberg
School of Computer Science and Information Systems
Pace University

Technical Report Series

EDITORIAL BOARD

Editor:
Allen Stix, Computer Science, Pace--Westchester

Associate Editors:
Constance A. Knapp, Information Systems, Pace--Westchester
Susan M. Merritt, Dean, The Ivan G. Seidenberg School of Computer Science and Information Systems --Pace

Members:
Howard S. Blum, Computer Science, Pace--New York
Mary F. Courtney, Computer Science, Pace--Westchester
Nicholas J. De Lillo, Mathematics and Computer Science, Manhattan College
Daniel Farkas, Information Systems, Pace--Westchester
Fred Grossman, Information Systems; Doctor of Professional Studies, Pace--New York and White Plains
Fran Goertzel Gustavson, Information Systems, Pace--Westchester
Joseph F. Malerba, Computer Science, Pace--Westchester
John S. Mallozzi, Computer Information Sciences, Iona College
John C. Molluzzo, Information Systems, Pace--New York
Pauline Mosley, Technology Systems, Pace--New York
Narayan S. Murthy, Computer Science, Pace--New York
Catherine Ricardo, Computer Information Sciences, Iona College
Judith E. Sullivan, CSIS Alumna, MS in CS from Pace--Westchester
Sylvester Tuohy, Computer Science, Pace--Westchester

The Ivan G. Seidenberg School of Computer Science and Information Systems, through the Technical Report Series, provides members of the community an opportunity to disseminate the results of their research by publishing monographs, working papers, and tutorials. Technical Reports is a place where scholarly striving is respected.

All preprints and recent reprints are requested and accepted. New manuscripts are read by two members of the editorial board; the editor decides upon publication. Authors, please note that production is Xerographic from your submission. Statements of policy and mission may be found in issues #29 (April 1990) and #34 (September 1990).

Please direct submissions as well as requests for single copies to:

Allen Stix
The Ivan G. Seidenberg School of Computer Science and Information Systems
Goldstein Academic Center
Pace University
861 Bedford Road
Pleasantville, NY 10570-2799