International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2022): 7.942

Analysing Efficiency and Time Complexity of AVL Tree Re-balancing during Value Insertion

Himansh Chitkara

Singapore International School, Mumbai, Maharashtra, India

Research Question: Modelling the efficiency of an AVL (Adelson-Velskii and Landis) Tree in re-balancing itself in terms of time complexity whilst inserting values into it.

Abstract: ***This research paper experimentally models the efficiency of an AVL (Adelson-Velskii and Landis) Tree in re-balancing itself during the insertion of values, focusing on its time complexity. The study demonstrates a logarithmic relationship between the time required for insertion and re-balancing operations in the AVL Tree. Through empirical analysis, the paper provides valuable insights into the performance characteristics of AVL Trees and their suitability for handling large data sets. The findings highlight the effectiveness of AVL Trees in maintaining balance and optimizing insertion operations, contributing to the understanding of efficient data structures.***

Keywords: Binary Tree, AVL Tree, Efficiency, Time Complexity, Re-balancing, Insertion Operations, Node, Balancing Algorithms

1. Introduction

Binary search trees are a fundamental data structure widely used in various applications due to their efficient organization and retrieval of data. This research paper focuses on the structure of binary search trees, specifically exploring one important type: the Adelson-Velskii. These type of trees are known for their ability to maintain balance, ensuring optimal performance for insertion operations. Through the course of this paper, the time complexity of both trees will be compared whilst inserting values into them in order to collect data to compare their re-balancing efficiencies.

The research question guiding this investigation is: Modelling the efficiency of an AVL (Adelson-Velskii and Landis) Tree in re-balancing itself in terms of time complexity whilst inserting values into it. By addressing this question, we aim to provide valuable insights into the performance characteristics of these tree structures, aiding in the understanding and selection of appropriate data structures for specific applications.

In the following sections, we will delve into the concepts of AVL Trees, exploring their respective properties, mechanisms for maintaining balance, and analysing their time complexity for insertion operations.

2. Background Information

2.1 Binary Search Trees and Time Complexities

Binary Search Trees

A binary search tree (BST) is a data structure organized in a manner such that it allows for data to be found in the quickest way possible. It is crucial to understand this concept as it is the basis of the algorithm in question.

The word binary means “made up of two parts” (Definition of BINARY, 2023). In the case of binary trees, it means that each item in a tree can point to a maximum of two other items: these items are known as children. All of these items are collectively known as nodes and each of these nodes contain a value inside of them. A BST is a special type of this tree that inserts nodes in a manner that makes it possible to efficiently search, insert, and delete each node of the tree (Introduction to Binary Search Tree - Data Structure and Algorithm Tutorials - GeeksforGeeks, 2020).

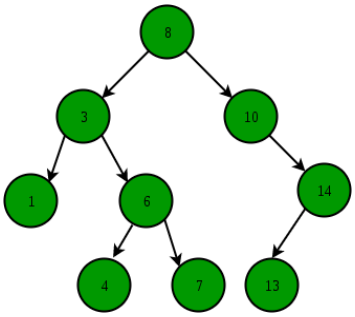

Figure 1: Binary Tree

As shown above, Figure 1 depicts a binary tree with several different nodes. Before moving forward, it is important to understand a few rules relating to the insertion of nodes into a BST. They have been listed below.

1) If the binary tree is empty, i.e. it contains no nodes inside of it, the first node to be created inside of it will be known as the *root node*. This can be easily identified by finding the node at the top of the binary tree. For instance, in Figure 1, the node containing the value 8 is the root node.
2) Values to left of the root node must are part of the left subtree and will always contain values lesser than the root node. Contrarily, values to the right of the root node are part of the right subtree and will always contain values greater than the parent node itself.
3) A node can have 0, 1, or 2 children. The left child of any node will always contain a value lesser than that of the parent node itself. On the other hand, the right child of any node will contain a value greater than that of the parent node itself.

4) For the purposes of this investigation, duplicate values cannot be entered into a binary tree as it will change the handling algorithms and may require different binary search algorithms compared to those mentioned in the research question (Binary Search Tree: Insertion | PrepInsta, n.d.).

Once a BST has been successfully created, it is possible for us to begin the "Search" process inside of it using the data structure to our advantage. In data structures, *searching* is a key process which allows for certain values to be found inside the data structure.

Time Complexities

In the field of Computer Science, *time complexity* is a term referring to the time it takes for a particular algorithm to successfully execute (Understanding Time Complexity With Simple Examples - GeeksforGeeks, 2017). Although there are many ways in which the time complexity of an algorithm can be represented, this essay will focus on the worst-case time complexity represented through the *Big-O Notation* in order to highlight the longest possible time it can take for an algorithm to execute (Big O Notation in Data Structure: An Introduction, Simplilearn, 2022).

Relating the above paragraph to BST, we find that a BST can provide us with a significant time advantage when dealing with large amounts of data through their search method as compared to linear searches which are executed in structures such as unordered arrays. This is because the Big-O Notation for an array is *O(N)* (Linear Search Vs Binary Search: Difference Between Linear Search & Binary Search | upGrad Blog, n.d.) where N refers to the number of elements present inside the array whereas it is $O(\log_2 N)$ (Bartakke, 2019) for a BST where N is the number of nodes in the data structure. When graphed, an obvious difference between the worst-case scenarios of each data structure is visible.

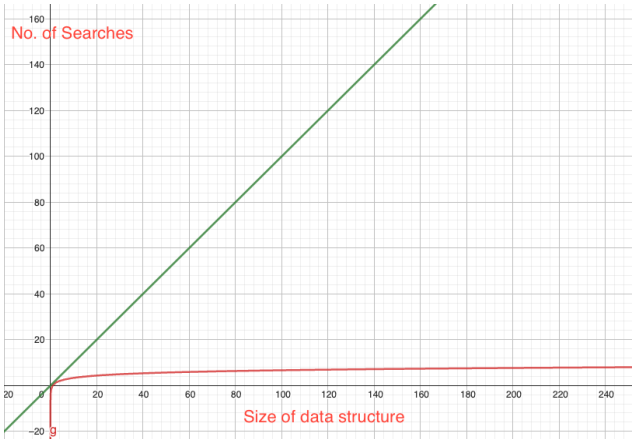

Figure 2: Worst-Case Scenarios for Linear Search in an Array (Green) and using a BST (Red)

As is clearly visible in Figure 2 above, as the size of the data structure increases, it is more beneficial to use a BST rather than a data structure such as an array to find an element as it requires a fewer number of searches implying that it would clearly take lesser time. However, this diagram assumes that the binary tree has been organized properly and is *balanced.* However, if we consider an unbalanced binary tree as shown in Figure 3 below, its worst-case time complexity would not be $O(\log_2 N)$, rather it would be *O(N)*.

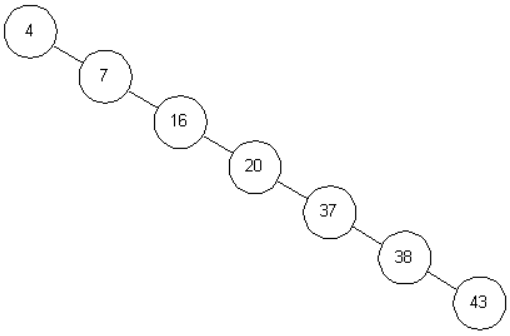

Figure 3: An unbalanced binary tree

It is crucial to understand the difference given in the above paragraph as it contains the answer to the question – Why are Binary Search Trees required? To avoid problems like the aforementioned, *balancing algorithms* are required to maintain the structure of the BST and ensure that the tree is created in the most optimal manner possible to reduce search times. This is also the function of the algorithm in question in this research paper, the AVL tree.

2.2 AVL (Adelson-Velskii and Landis) Trees:

AVL Trees are a type of BST that apply a special property called the *height-balance property* within themselves in order to balance the binary tree (AVL Trees Height-Balance Property, 2015). The height-balance property states that for the tree to be considered as a balanced tree, the height difference between the children of a particular node cannot be more than 1. Otherwise, the tree is considered to be unbalanced.

To explain the above property further, the term *height* inside an AVL tree refers to the length of the longest path from the tree's root to one of its leaves (Gautam, 2022). Please note that a leaf in a binary tree is a node that has no children.

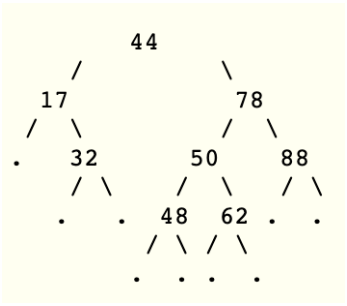

Figure 4: A balanced AVL tree.

In the diagram above, the height of the left subtree is 2 whereas the length of the right subtree is 3. As the difference between these two values is 1, the binary tree has been successfully balanced as it is meeting the condition which has been specified in paragraph 1. Although this may not seem like the best way to balance the tree, it is balanced well enough for a search algorithm to take place.

In a scenario where the AVL tree is not balanced, i.e., the height difference property is not met, a process called

rotation takes place inside the binary tree. Let us consider figure 4 below:

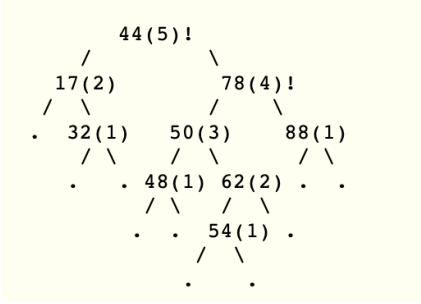

Figure 5: An unbalanced AVL Tree

First, let us understand why this AVL tree is unbalanced. As the height-difference between the right and left subtrees is 2, the height-balance property is not met leading to an imbalance. Hence, this AVL tree must be rearranged. Although it is beyond the scope of this research paper to explain how this principle works in the programs attached in *Appendices: 1,2,3*, it can be explained in theory.

From figure 5, we can clearly see that the right subtree has too many nodes inside it leading to an imbalance of the tree. Hence, to correct this imbalance, a rearrangement process known as *Rotation* takes place inside the AVL Tree (AVL Tree Data Structure - GeeksforGeeks, 2023). In simple terms, rotation is a process where nodes from the subtree with the greater height are picked and shifted to the subtree with a lesser height. Eventually, this shall balance the tree and it will allow us to run Search algorithms with utmost efficiency.

To do this, we must follow 3 steps:

1) Find out which side of the tree is unbalanced and locate the node closest to the root, i.e., the insertion point of the tree. Let's call this A.
2) Of A's two subtrees, locate the subtree with the greater height. Let us call the root of this subtree B.
3) Finally, locate B's two subtrees and select the one with the greater height. Label it C.

This can be seen in figure 6 below.

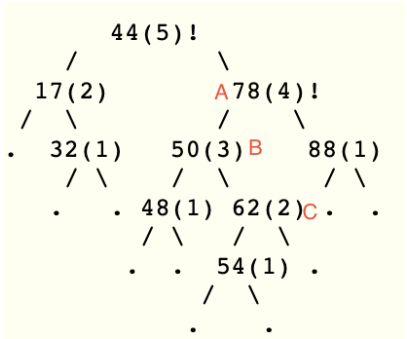

Figure 6

Now, if you observe carefully, all 3 nodes which we have marked are in a linear sequence inside the tree. As we have seen in figure 2 earlier, a linear sequence will increase the search time required inside the binary tree. Hence to break this sequence apart, we rearrange the nodes into the pattern shown in Figure 7.

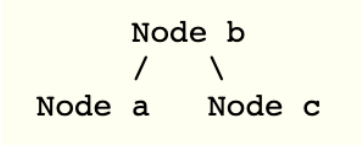

Figure 7: Rearranging the nodes

The final step is finding what a, b, c corresponds to. Ideally, a is the smallest value, b is the middle value and c is the largest. Hence considering our example above, a will correspond to Node B (contains 50), b will correspond to Node C (contains 62), c will correspond to Node A (contains 72). Now, as all of these nodes have their subtrees, once they have been rotated, they will be reconnected to their subtrees to form an ordered tree with reduced height. The same is shown in Figure 8 below.

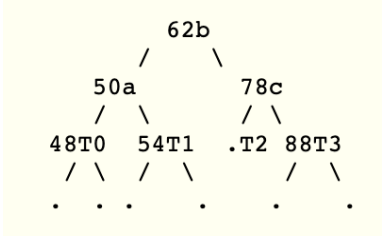

Figure 8: Ordered right sub-tree

3. Hypothesis and Methodology

Based on the theory in Section 2, the principle behind the working of the AVL tree algorithm has been explored in detail. Now, to model the relationship between the dataset size and the time it will take for the algorithm to run, we must remember that the AVL Tree has a theoretical worst-case efficiency of $O(\log_2 N)$. Although this doesn't take into account the insertion of values into the tree and rebalancing, which is being done in this experiment, as modern computers are extremely fast this shouldn't impact results vastly.

Hence, it can be predicted that the logarithmic relationship seen in Figure 2 of this essay will be seen in the results of this experiment too.

To carry out this experiment, the independent variable will be the size of the dataset and the dependent variable will be the time take to insert and re-balance the tree. Datasets will contain consecutive integers from 1 to N, where N is the size of the dataset incremented in 100s. The time will be measured in nanoseconds through the program in order to obtain the most precise reading possible.

4. Results, Graphs, and Analysis

As per the methodology above, the experiment has been conducted for multiple data set sizes and each reading has been collected and averaged over 10 times to obtain a reliable result.

Size	Average Time To Insert And Rebalance (AVL Tree) Nano Seconds (10 Trials Each)
100	525778
200	840316
300	1103392
400	1110556
500	1330456
600	1408253
700	1556087
800	1602452
900	1623423
1000	1652313

Figure 9: Result Readings

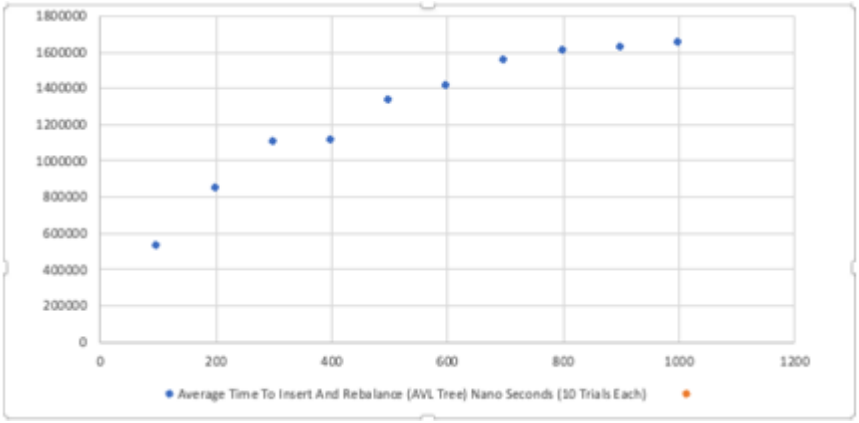

Figure 10: Graph of Results table in Figure 9. *Y-axis is average time in nanoseconds X-axis is the size of the dataset. (Please note these readings have been collected on a MacBook Air M1 2020 and can vary depending on the computer's performance)*

Based on the graph obtained above (drawn using Excel), we clearly see that there is a logarithmic relationship between the average time taken for insertion and the size of the dataset. This confirms my hypothesis in section 3.

I have also inputted these values into my GDC (Graphic Display Calculator) to find the relationship between the two values. The model in nanoseconds, expressed as a natural logarithm, is below:

$$y = -1.845 \times 10^6 + 510324.733 lnx$$

However, I would like to point out that the dataset of size 400 stands out as it doesn't fall within the expected trend of points. This could be due to recursion being used in the program obtained which can occasionally impact readings and lead to shorter times in a few cases. Furthermore, the model depicts the y-intercept as -0.001845 seconds which suggests there is a small error in the experiment readings as the time obtained can never be negative; however, as this is extremely minute, it can be ignored and the results can be considered largely accurate.

5. Conclusion

In conclusion, the experiment conducted in this research paper successfully achieved its objective of developing a model for inserting values into an AVL tree based on the theoretical foundation presented in Section 2. As observed in Section 4, the obtained results demonstrated a consistent and anticipated logarithmic relationship between the time required for insertion and re-balancing operations in AVL trees. Thus, it can be confidently concluded that this research paper effectively models the relationship between these variables, effectively fulfilling its intended purpose.

However, it is important to recognize the potential for further exploration and advancement of this experiment. Building upon the methodology employed in this study, it is feasible to create similar models for other binary search tree (BST) algorithms and conduct a comparative analysis of their efficiency in value insertion and re-balancing procedures. Such an investigation would be invaluable in determining the most optimal algorithm, thereby reducing processing time and enhancing performance in the industrial realm. Expanding the scope of this research to encompass a range of BST algorithms, such as the Red-Black Tree, Splay Tree, and B-Tree, would facilitate a more comprehensive evaluation of their insertion and re-balancing efficiency. Employing a consistent experimental approach as demonstrated in this paper, one could examine the time complexity associated with insertion operations and assess the effectiveness of re-balancing mechanisms employed by each algorithm.

Overall, I believe this research paper has significantly improved my understanding of binary trees and has provided me with a fantastic learning opportunity.

References

[1] *AVL Trees Height-Balance Property*. (2015, September 14). Computer Science Stack Exchange.https://cs.stackexchange.com/questions/47175/avl-trees-height-balance-property
[2] *Big O Notation in Data Structure: An Introduction | Simplilearn*. (2022, September 1). Simplilearn.com.https://www.simplilearn.com/big-o-notation-in-data-structure-article
[3] *Binary Search Tree: Insertion | PrepInsta*. (n.d.). PREP INSTA.https://prepinsta.com/data-structures/binary-search-tree-insertion/
[4] *Binary Search Trees*. (n.d.). Binary Search Trees.https://www.eecs.umich.edu/courses/eecs380/ALG/niemann/s_bin.htm
[5] *Definition of BINARY*. (2023, July 28). Binary Definition & Meaning - Merriam-Webster.https://www.merriam-webster.com/dictionary/binary
[6] Gautam, S. (n.d.). *Find Height or Maximum Depth of a Binary Tree*. Height (Maximum Depth) of a Binary Tree.https://www.enjoyalgorithms.com/blog/find-height-of-a-binary-tree
[7] *Introduction to Binary Search Tree - Data Structure and Algorithm Tutorials - GeeksforGeeks*. (2020, September 30). GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-binary-search-tree-data-structure-and-algorithm-tutorials/
[8] *Linear Search vs Binary Search: Difference Between Linear Search & Binary Search | upGrad blog*. (n.d.). upGrad Blog. https://www.upgrad.com/blog/linear-search-vs-binary-search/
[9] P. (n.d.). *AVL Tree Data Structure*. AVL Tree.https://www.enjoyalgorithms.com/blog/avl-tree-data-structure

Appendices:

Appendix 1: AvlTree.java

```
// AvlTree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x (unimplemented)
// boolean contains( x )  --> Return true if x is present
// boolean remove( x )    --> Return true if x was present
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// booleanisEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void printTree( )      --> Print tree in sorted order
// ******************ERRORS********************************
// Throws UnderflowException as appropriate

/**
 * Implements an AVL tree.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class AvlTree<AnyType extends Comparable<? super AnyType>>
{
    /**
     * Construct the tree.
     */
    public AvlTree( )
    {
        root = null;
    }

    /**
     * Insert into the tree; duplicates are ignored.
     * @param x the item to insert.
     */
    public void insert( AnyType x )
    {
        root = insert( x, root );
    }

    /**
```

```
     * Remove from the tree. Nothing is done if x is not found.
     * @param x the item to remove.
     */
    public void remove( AnyType x )
    {
        root = remove( x, root );
    }


    /**
     * Internal method to remove from a subtree.
     * @param x the item to remove.
     * @param t the node that roots the subtree.
     * @return the new root of the subtree.
     */
    private AvlNode<AnyType> remove( AnyType x, AvlNode<AnyType> t )
    {
        if( t == null )
            return t;   // Item not found; do nothing

        int compareResult = x.compareTo( t.element );

        if( compareResult< 0 )
t.left = remove( x, t.left );
        else if( compareResult> 0 )
t.right = remove( x, t.right );
        else if( t.left != null &&t.right != null ) // Two children
        {
t.element = findMin( t.right ).element;
t.right = remove( t.element, t.right );
        }
        else
            t = ( t.left != null ) ? t.left : t.right;
        return balance( t );
    }

    /**
     * Find the smallest item in the tree.
     * @return smallest item or null if empty.
     */
    public AnyTypefindMin( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( );
        return findMin( root ).element;
    }

    /**
     * Find the largest item in the tree.
     * @return the largest item of null if empty.
     */
    public AnyTypefindMax( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( );
        return findMax( root ).element;
    }

    /**
     * Find an item in the tree.
     * @param x the item to search for.
     * @return true if x is found.
     */
    public boolean contains( AnyType x )
```

```
    {
        return contains( x, root );
    }

    /**
     * Make the tree logically empty.
     */
    public void makeEmpty( )
    {
        root = null;
    }

    /**
     * Test if the tree is logically empty.
     * @return true if empty, false otherwise.
     */
    public booleanisEmpty( )
    {
        return root == null;
    }

    /**
     * Print the tree contents in sorted order.
     */
    public void printTree( )
    {
        if( isEmpty( ) )
System.out.println( "Empty tree" );
        else
printTree( root );
    }

    private static final int ALLOWED_IMBALANCE = 1;

    // Assume t is either balanced or within one of being balanced
    private AvlNode<AnyType> balance( AvlNode<AnyType> t )
    {
        if( t == null )
            return t;

        if( height( t.left ) - height( t.right ) > ALLOWED_IMBALANCE )
            if( height( t.left.left ) >= height( t.left.right ) )
                t = rotateWithLeftChild( t );
            else
                t = doubleWithLeftChild( t );
        else
        if( height( t.right ) - height( t.left ) > ALLOWED_IMBALANCE )
            if( height( t.right.right ) >= height( t.right.left ) )
                t = rotateWithRightChild( t );
            else
                t = doubleWithRightChild( t );

t.height = Math.max( height( t.left ), height( t.right ) ) + 1;
        return t;
    }

    public void checkBalance( )
    {
checkBalance( root );
    }

    private int checkBalance( AvlNode<AnyType> t )
    {
        if( t == null )
```

```
            return -1;

        if( t != null )
        {
            int hl = checkBalance( t.left );
            int hr = checkBalance( t.right );
            if( Math.abs( height( t.left ) - height( t.right ) ) > 1 ||
                    height( t.left ) != hl || height( t.right ) != hr )
System.out.println( "OOPS!!" );
        }

        return height( t );
    }


    /**
     * Internal method to insert into a subtree.
     * @param x the item to insert.
     * @param t the node that roots the subtree.
     * @return the new root of the subtree.
     */
    private AvlNode<AnyType> insert( AnyType x, AvlNode<AnyType> t )
    {
        if( t == null )
            return new AvlNode<>( x, null, null );

        int compareResult = x.compareTo( t.element );

        if( compareResult< 0 )
t.left = insert( x, t.left );
        else if( compareResult> 0 )
t.right = insert( x, t.right );
        else
            ;  // Duplicate; do nothing
        return balance( t );
    }

    /**
     * Internal method to find the smallest item in a subtree.
     * @param t the node that roots the tree.
     * @return node containing the smallest item.
     */
    private AvlNode<AnyType>findMin( AvlNode<AnyType> t )
    {
        if( t == null )
            return t;

        while( t.left != null )
            t = t.left;
        return t;
    }

    /**
     * Internal method to find the largest item in a subtree.
     * @param t the node that roots the tree.
     * @return node containing the largest item.
     */
    private AvlNode<AnyType>findMax( AvlNode<AnyType> t )
    {
        if( t == null )
            return t;

        while( t.right != null )
            t = t.right;
```

```
        return t;
    }

    /**
     * Internal method to find an item in a subtree.
     * @param x is item to search for.
     * @param t the node that roots the tree.
     * @return true if x is found in subtree.
     */
    private boolean contains( AnyType x, AvlNode<AnyType> t )
    {
        while( t != null )
        {
            int compareResult = x.compareTo( t.element );

            if( compareResult< 0 )
                t = t.left;
            else if( compareResult> 0 )
                t = t.right;
            else
                return true;    // Match
        }

        return false;   // No match
    }

    /**
     * Internal method to print a subtree in sorted order.
     * @param t the node that roots the tree.
     */
    private void printTree( AvlNode<AnyType> t )
    {
        if( t != null )
        {
printTree( t.left );
System.out.println( t.element );
printTree( t.right );
        }
    }

    /**
     * Return the height of node t, or -1, if null.
     */
    private int height( AvlNode<AnyType> t )
    {
        return t == null ? -1 : t.height;
    }

    /**
     * Rotate binary tree node with left child.
     * For AVL trees, this is a single rotation for case 1.
     * Update heights, then return new root.
     */
    private AvlNode<AnyType>rotateWithLeftChild( AvlNode<AnyType> k2 )
    {
AvlNode<AnyType> k1 = k2.left;
        k2.left = k1.right;
        k1.right = k2;
        k2.height = Math.max( height( k2.left ), height( k2.right ) ) + 1;
        k1.height = Math.max( height( k1.left ), k2.height ) + 1;
        return k1;
    }

    /**
```

```
     * Rotate binary tree node with right child.
     * For AVL trees, this is a single rotation for case 4.
     * Update heights, then return new root.
     */
    private AvlNode<AnyType>rotateWithRightChild( AvlNode<AnyType> k1 )
    {
AvlNode<AnyType> k2 = k1.right;
        k1.right = k2.left;
        k2.left = k1;
        k1.height = Math.max( height( k1.left ), height( k1.right ) ) + 1;
        k2.height = Math.max( height( k2.right ), k1.height ) + 1;
        return k2;
    }

    /**
     * Double rotate binary tree node: first left child
     * with its right child; then node k3 with new left child.
     * For AVL trees, this is a double rotation for case 2.
     * Update heights, then return new root.
     */
    private AvlNode<AnyType>doubleWithLeftChild( AvlNode<AnyType> k3 )
    {
        k3.left = rotateWithRightChild( k3.left );
        return rotateWithLeftChild( k3 );
    }

    /**
     * Double rotate binary tree node: first right child
     * with its left child; then node k1 with new right child.
     * For AVL trees, this is a double rotation for case 3.
     * Update heights, then return new root.
     */
    private AvlNode<AnyType>doubleWithRightChild( AvlNode<AnyType> k1 )
    {
        k1.right = rotateWithLeftChild( k1.right );
        return rotateWithRightChild( k1 );
    }

    private static class AvlNode<AnyType>
    {
            // Constructors
AvlNode( AnyTypetheElement )
        {
            this( theElement, null, null );
        }

AvlNode( AnyTypetheElement, AvlNode<AnyType>lt, AvlNode<AnyType> rt )
        {
            element  = theElement;
            left     = lt;
            right    = rt;
            height   = 0;
        }

AnyType           element;      // The data in the node
AvlNode<AnyType>  left;         // Left child
AvlNode<AnyType>  right;        // Right child
        int               height;       // Height
    }

      /** The tree root. */
    private AvlNode<AnyType> root;
```

```
        // Test program
    public static void main( String [ ] args )
    {
AvlTree<Integer> t = new AvlTree<>( );
        final int SMALL = 40;
        final int NUMS = 1000000;  // must be even
        final int GAP  =   37;

System.out.println( "Checking... (no more output means success)" );

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
        {
        //    System.out.println( "INSERT: " + i );
t.insert( i );
            if( NUMS < SMALL )
t.checkBalance( );
        }

        for( int i = 1; i < NUMS; i+= 2 )
        {
         //   System.out.println( "REMOVE: " + i );
t.remove( i );
            if( NUMS < SMALL )
t.checkBalance( );
        }
        if( NUMS < SMALL )
t.printTree( );
        if( t.findMin( ) != 2 || t.findMax( ) != NUMS - 2 )
System.out.println( "FindMin or FindMax error!" );

        for( int i = 2; i < NUMS; i+=2 )
             if( !t.contains( i ) )
System.out.println( "Find error1!" );

        for( int i = 1; i < NUMS; i+=2 )
        {
            if( t.contains( i ) )
System.out.println( "Find error2!" );
        }
    }
}
```

Appendix 2: AvlNode.java

```
    package DataStructures;

    // Basic node stored in AVL trees
    // Note that this class is not accessible outside
    // of package DataStructures

    class AvlNode
    {
            // Constructors
AvlNode( Comparable theElement )
        {
            this( theElement, null, null );
        }

AvlNode( Comparable theElement, AvlNodelt, AvlNode rt )
        {
            element  = theElement;
            left     = lt;
            right    = rt;
            height   = 0;
        }
```

```
            // Friendly data; accessible by other package routines
        Comparable element;      // The data in the node
AvlNode    left;          // Left child
AvlNode    right;         // Right child
        int        height;       // Height
    }
```

Appendix 3: Program used to conduct the experiment

```
int set = 100; // Change and re-run program

for (int trial = 1; trial <= 10; trial++) {
        AvlTree avl = new AvlTree();
        RedBlackTree rb = new RedBlackTree(Double.MIN_VALUE);

        long startAVL = System.nanoTime();
        for (double i = 1; i <= set; i++)
                avl.insert(i);
        long endAVL = System.nanoTime();

        long startRB = System.nanoTime();
        for (double i = 1; i <= set; i++)
                rb.insert(i);
        long endRB = System.nanoTime();

        System.out.println("AVL Trial " + trial + ": " + (endAVL - startAVL));
        System.out.println("RB Trial " + trial + ": " + (endRB - startRB));
}
```